

Data-Structures Archive

Singly Linked List:

1. Introduction to Linked List
2. Linked List vs Array
3. Linked List Insertion
4. Linked List Deletion (Deleting a given key)
5. Linked List Deletion (Deleting a key at given position)
6. A Programmer's approach of looking at Array vs. Linked List
7. Find Length of a Linked List (Iterative and Recursive)
8. Search an element in a Linked List (Iterative and Recursive)
9. How to write C functions that modify head pointer of a Linked List?
10. Swap nodes in a linked list without swapping data
11. Write a function to get Nth node in a Linked List
12. Print the middle of a given linked list
13. Nth node from the end of a Linked List
14. Write a function to delete a Linked List
15. Write a function that counts the number of times a given int occurs in a Linked List
16. Reverse a linked list
17. Detect loop in a linked list
18. Merge two sorted linked lists
19. Generic Linked List in C
20. Given a linked list which is sorted, how will you insert in sorted way
21. Given only a pointer to a node to be deleted in a singly linked list, how do you delete it?
22. Function to check if a singly linked list is palindrome
23. Intersection point of two Linked Lists.
24. Recursive function to print reverse of a Linked List
25. Remove duplicates from a sorted linked list
26. Remove duplicates from an unsorted linked list
27. Pairwise swap elements of a given linked list
28. Practice questions for Linked List and Recursion
29. Move last element to front of a given Linked List
30. Intersection of two Sorted Linked Lists
31. Delete alternate nodes of a Linked List
32. Alternating split of a given Singly Linked List
33. Identical Linked Lists
34. Merge Sort for Linked Lists
35. Reverse a Linked List in groups of given size
36. Reverse alternate K nodes in a Singly Linked List
37. Delete nodes which have a greater value on right side
38. Segregate even and odd nodes in a Linked List
39. Detect and Remove Loop in a Linked List
40. Add two numbers represented by linked lists | Set 1
41. Delete a given node in Linked List under given constraints
42. Union and Intersection of two Linked Lists
43. Find a triplet from three linked lists with sum equal to a given number
44. Rotate a Linked List
45. Flattening a Linked List
46. Add two numbers represented by linked lists | Set 2
47. Sort a linked list of 0s, 1s and 2s
48. Flatten a multilevel linked list
49. Delete N nodes after M nodes of a linked list
50. QuickSort on Singly Linked List
51. Merge a linked list into another linked list at alternate positions
52. Pairwise swap elements of a given linked list by changing links
53. Given a linked list of line segments, remove middle points
54. Construct a Maximum Sum Linked List out of two Sorted Linked Lists having some Common nodes
55. Can we reverse a linked list in less than $O(n)$?
56. Clone a linked list with next and random pointer | Set 1
57. Clone a linked list with next and random pointer | Set 2
58. Insertion Sort for Singly Linked List
59. Point to next higher value node in a linked list with an arbitrary pointer
60. Rearrange a given linked list in-place.
61. Sort a linked list that is sorted alternating ascending and descending orders?
62. Select a Random Node from a Singly Linked List
63. Why Quick Sort preferred for Arrays and Merge Sort for Linked Lists?
64. Merge two sorted linked lists such that merged list is in reverse order
65. Compare two strings represented as linked lists
66. Rearrange a linked list such that all even and odd positioned nodes are together
67. Rearrange a Linked List in Zig-Zag fashion
68. Add 1 to a number represented as linked list
69. Point arbit pointer to greatest value right side node in a linked list
70. Merge two sorted linked lists such that merged list is in reverse order
71. Convert a given Binary Tree to Doubly Linked List | Set
72. Check if a linked list of strings forms a palindrome
73. Sort linked list which is already sorted on absolute values
74. Delete last occurrence of an item from linked list
75. Delete a Linked List node at a given position
76. Linked List in java

Circular Linked List:

1. Circular Linked List Introduction and Applications,
2. Circular Linked List Traversal
3. Split a Circular Linked List into two halves
4. Sorted insert for circular linked list

Doubly Linked List:

1. Doubly Linked List Introduction and Insertion
2. Delete a node in a Doubly Linked List
3. Reverse a Doubly Linked List
4. The Great Tree-List Recursion Problem.

5. Copy a linked list with next and arbit pointer
6. QuickSort on Doubly Linked List
7. Swap Kth node from beginning with Kth node from end in a Linked List
8. Merge Sort for Doubly Linked List

[Quiz on Linked List](#)

[Practice Programming Questions on Linked List](#)

[Data Structures](#)

[Recent articles on Linked List](#)

[Ask a Question](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Company Wise Coding Practice Topic Wise Coding Practice

Stack

- Introduction to Stack
- Infix to Postfix Conversion using Stack
- Evaluation of Postfix Expression
- Reverse a String using Stack
- Implement two stacks in an array
- Check for balanced parentheses in an expression
- Next Greater Element
- Reverse a stack using recursion
- Sort a stack using recursion
- The Stock Span Problem
- Design and Implement Special Stack Data Structure
- Implement Stack using Queues
- Design a stack with operations on middle element
- How to efficiently implement k stacks in a single array?
- Iterative Tower of Hanoi
- Length of the longest valid substring
- Find maximum of minimum for every window size in a given array
- Check if a given array can represent Preorder Traversal of Binary Search Tree
- Minimum number of bracket reversals needed to make an expression balanced
- Iterative Depth First Traversal of Graph
- How to create mergable stack?
- Print ancestors of a given binary tree node without recursion
- Expression Evaluation
- Largest Rectangular Area in a Histogram | Set 2
- The Celebrity Problem
- Iterative Postorder Traversal | Set 2 (Using One Stack)
- Iterative Postorder Traversal | Set 1 (Using Two Stacks)
- Implement a stack using single queue
- Design a stack that supports getMin() in O(1) time and O(1) extra space

Coding Practice on Stack

[Quiz on Stack](#)

[Forum Questions on Stack](#)

[Ask a Question](#)

[Data Structures](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Company Wise Coding Practice Topic Wise Coding Practice

1. Queue Introduction and Array Implementation
2. Linked List Implementation of Queue
3. Applications of Queue Data Structure
4. Priority Queue Introduction
5. Deque (Introduction and Applications)
6. Implement Queue using Stacks
7. Check whether a given Binary Tree is Complete or not
8. Find the largest multiple of 3
9. Find the first circular tour that visits all petrol pumps
10. Maximum of all subarrays of size k
11. An Interesting Method to Generate Binary Numbers from 1 to n
12. How to efficiently implement k Queues in a single array?
13. Minimum time required to rot all oranges
14. Iterative Method to find Height of Binary Tree
15. Construct Complete Binary Tree from its Linked List Representation
16. Implement LRU Cache
17. Breadth First Traversal for a Graph
18. Implement a stack using single queue

[Quiz on Queue](#)

[Coding Practice On Queue](#)

[Forum Questions on Queue](#)

[Ask a Question](#)

[Data Structures](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Company Wise Coding Practice Topic Wise Coding Practice

Binary Tree:

1. Binary Tree Introduction
2. Binary Tree Properties
3. Types of Binary Tree
4. Handshaking Lemma and Interesting Tree Properties
5. Enumeration of Binary Tree
6. Applications of tree data structure
7. Tree Traversals
8. BFS vs DFS for Binary Tree
9. Level Order Tree Traversal
10. Print level order traversal line by line
11. Inorder Tree Traversal without Recursion
12. Inorder Tree Traversal without recursion and without stack!
13. Threaded Binary Tree
14. Size of a tree
15. Determine if Two Trees are Identical
16. Maximum Depth or Height of a Tree
17. Write a C program to Delete a Tree.
18. Write an Efficient C Function to Convert a Binary Tree into its Mirror Tree
19. If you are given two traversal sequences, can you construct the binary tree?
20. Given a binary tree, print out all of its root-to-leaf paths one per line.
21. The Great Tree-List Recursion Problem.
22. Count leaf nodes in a binary tree
23. Level order traversal in spiral form
24. Check for Children Sum Property in a Binary Tree.
25. Convert an arbitrary Binary Tree to a tree that holds Children Sum Property
26. Diameter of a Binary Tree
27. How to determine if a binary tree is height-balanced?
28. Root to leaf path sum equal to a given number
29. Construct Tree from given Inorder and Preorder traversals
30. Given a binary tree, print all root-to-leaf paths
31. Double Tree
32. Maximum width of a binary tree
33. Foldable Binary Trees
34. Print nodes at k distance from root
35. Get Level of a node in a Binary Tree
36. Print Ancestors of a given node in Binary Tree
37. Check if a given Binary Tree is SumTree
38. Check if a binary tree is subtree of another binary tree
39. Connect nodes at same level
40. Connect nodes at same level using constant extra space
41. Populate Inorder Successor for all nodes
42. Convert a given tree to its Sum Tree
43. Vertical Sum in a given Binary Tree
44. Find the maximum sum leaf to root path in a Binary Tree
45. Construct Special Binary Tree from given Inorder traversal
46. Construct a special tree from given preorder traversal
47. Check whether a given Binary Tree is Complete or not
48. Boundary Traversal of binary tree
49. Construct Full Binary Tree from given preorder and postorder traversals
50. Iterative Preorder Traversal
51. Morris traversal for Preorder
52. Linked complete binary tree & its creation
53. Ternary Search Tree
54. Largest Independent Set Problem
55. Iterative Postorder Traversal | Set 1 (Using Two Stacks)
56. Iterative Postorder Traversal | Set 2 (Using One Stack)
57. Reverse Level Order Traversal
58. Construct Complete Binary Tree from its Linked List Representation
59. Convert a given Binary Tree to Doubly Linked List | Set 1
60. Tree Isomorphism Problem
61. Find all possible interpretations of an array of digits
62. Iterative Method to find Height of Binary Tree
63. Custom Tree Problem
64. Convert a given Binary Tree to Doubly Linked List | Set 2
65. Print ancestors of a given binary tree node without recursion
66. Difference between sums of odd level and even level nodes of a Binary Tree
67. Print Postorder traversal from given Inorder and Preorder traversals
68. Find depth of the deepest odd level leaf node
69. Check if all leaves are at same level
70. Print Left View of a Binary Tree
71. Remove all nodes which don't lie in any path with sum $\geq k$
72. Extract Leaves of a Binary Tree in a Doubly Linked List
73. Deepest left leaf node in a binary tree
74. Find next right node of a given key
75. Sum of all the numbers that are formed from root to leaf paths
76. Convert a given Binary Tree to Doubly Linked List | Set 3
77. Lowest Common Ancestor in a Binary Tree | Set 1
78. Find distance between two given keys of a Binary Tree
79. Print all nodes that are at distance k from a leaf node
80. Check if a given Binary Tree is height balanced like a Red-Black Tree,
81. Print all nodes at distance k from a given node
82. Print a Binary Tree in Vertical Order | Set 1
83. Construct a tree from Inorder and Level order traversals
84. Find the maximum path sum between two leaves of a binary tree
85. Reverse alternate levels of a perfect binary tree
86. Check if two nodes are cousins in a Binary Tree
87. Check if a binary tree is subtree of another binary tree | Set 2
88. Serialize and Deserialize a Binary Tree
89. Print nodes between two given level numbers of a binary tree
90. closest leaf in a Binary Tree
91. Convert a Binary Tree to Threaded binary tree
92. Print Nodes in Top View of Binary Tree
93. Bottom View of a Binary Tree

94. Perfect Binary Tree Specific Level Order Traversal
95. Convert left-right representation of a binary tree to down-right
96. Minimum no. of iterations to pass information to all nodes in the tree
97. Clone a Binary Tree with Random Pointers
98. Given a binary tree, how do you remove all the half nodes?
99. Vertex Cover Problem | Set 2 (Dynamic Programming Solution for Tree)
100. Check whether a binary tree is a full binary tree or not
101. Find sum of all left leaves in a given Binary Tree
102. Remove nodes on root to leaf paths of length $< K$
103. Find Count of Single Valued Subtrees
104. Check if a given array can represent Preorder Traversal of Binary Search Tree
105. Mirror of n -ary Tree
106. Find multiplication of sums of data of leaves at same levels
107. Succinct Encoding of Binary Tree
108. Construct Binary Tree from given Parent Array representation
109. Symmetric Tree (Mirror Image of itself)
110. Find Minimum Depth of a Binary Tree
111. Maximum Path Sum in a Binary Tree
112. Expression Tree
113. Check whether a binary tree is a complete tree or not | Set 2 (Recursive Solution)
114. Change a Binary Tree so that every node stores sum of all nodes in left subtree
115. Iterative Search for a key 'x' in Binary Tree
116. Find maximum (or minimum) in Binary Tree

[Quiz on Binary Tree](#)

[Quiz on Binary Tree Traversals](#)

[Forum Questions on Tree](#)

[Data Structures](#)

[Ask a Question](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Company Wise Coding Practice Topic Wise Coding Practice

Binary Search Tree:

1. Search and Insert in BST
2. Deletion from BST
3. Data Structure for a single resource reservations
4. Advantages of BST over Hash Table
5. Minimum value in a Binary Search Tree
6. Inorder predecessor and successor for a given key in BST
7. Check if a binary tree is BST or not
8. Lowest Common Ancestor in a Binary Search Tree.
9. Sorted order printing of a given array that represents a BST
10. Inorder Successor in Binary Search Tree
11. Find k -th smallest element in BST (Order Statistics in BST)
12. K 'th smallest element in BST using $O(1)$ Extra Space
13. Print BST keys in the given range
14. Sorted Array to Balanced BST
15. Find the largest BST subtree in a given Binary Tree
16. Check for Identical BSTs without building the trees
17. Add all greater values to every node in a given BST
18. Remove BST keys outside the given range
19. Check if each internal node of a BST has exactly one child
20. Find if there is a triplet in a Balanced BST that adds to zero
21. Merge two BSTs with limited extra space
22. Two nodes of a BST are swapped, correct the BST
23. Construct BST from given preorder traversal | Set 1
24. Construct BST from given preorder traversal | Set 2
25. Floor and Cell from a BST
26. Convert a BST to a Binary Tree such that sum of all greater keys is added to every key
27. Sorted Linked List to Balanced BST
28. In-place conversion of Sorted DLL to Balanced BST
29. Find a pair with given sum in a Balanced BST
30. Total number of possible Binary Search Trees with n keys
31. Merge Two Balanced Binary Search Trees
32. Binary Tree to Binary Search Tree Conversion
33. Transform a BST to greater sum tree
34. K 'th Largest Element in BST when modification to BST is not allowed
35. How to handle duplicates in Binary Search Tree?
36. Print Common Nodes in Two Binary Search Trees
37. Construct all possible BSTs for keys 1 to N
38. Print Common Nodes in Two Binary Search Trees
39. Count BST subtrees that lie in given range
40. Count BST nodes that lie in a given range
41. How to implement decrease key or change key in Binary Search Tree
42. Second largest element in BST
43. Count inversions in an array | Set 2 (Using Self-Balancing BST)

[Quiz on Binary Search Trees](#)

[Recent Articles on Binary Search Tree](#)

[Quiz on Balanced Binary Search Trees](#)

[Ask a Question](#)

[Data Structures](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Company Wise Coding Practice Topic Wise Coding Practice

Heap:

1. Binary Heap
2. Time Complexity of building a heap
3. Applications of Heap Data Structure
4. Why is Binary Heap Preferred over BST for Priority Queue?
5. Binomial Heap
6. Fibonacci Heap
7. Heap Sort
8. K'th Largest Element in an array
9. Sort an almost sorted array/
10. Tournament Tree (Winner Tree) and Binary Heap
11. Check if a given Binary Tree is Heap
12. How to check if a given array represents a Binary Heap?
13. Print all elements in sorted order from row and column wise sorted matrix
14. Connect n ropes with minimum cost
15. Design an efficient data structure for given operations
16. Merge k sorted arrays | Set 1
17. Sort numbers stored on different machines

[Coding Practice on Heap](#)

[Quiz on Heap](#)

[Forum Questions on Heap](#)

[Ask a Question](#)

[Data Structures](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

[Company Wise Coding Practice](#) [Topic Wise Coding Practice](#)

Hashing

1. Hashing Introduction
2. Separate Chaining for Collision Handling
3. Open Addressing for Collision Handling
4. Print a Binary Tree in Vertical Order
5. Find whether an array is subset of another array
6. Union and Intersection of two Linked Lists
7. Find a pair with given sum
8. Check if a given array contains duplicate elements within k distance from each other
9. Find Itinerary from a given list of tickets
10. Find number of Employees Under every Employee
11. Check if an array can be divided into pairs whose sum is divisible by k
12. Find four elements a, b, c and d in an array such that $a+b = c+d$
13. Given an array of pairs, find all symmetric pairs in it
14. Find the largest subarray with 0 sum
15. Longest Consecutive Subsequence
16. Count distinct elements in every window of size k
17. Design a data structure that supports insert, delete, search and getRandom in constant time
18. Advantages of BST over Hash Table
19. Group multiple occurrence of array elements ordered by first occurrence
20. How to check if two given sets are disjoint?
21. Length of the largest subarray with contiguous elements | Set 2
22. Clone a Binary Tree with Random Pointers
23. Find if there is a subarray with 0 sum
24. Largest subarray with equal number of 0s and 1s
25. Palindrome Substring Queries
26. Print all subarrays with 0 sum
27. Find subarray with given sum | Set 2 (Handles Negative Numbers)
28. Find smallest range containing elements from k lists
29. Pair with given product | Set 1 (Find if any pair exists)
30. Find missing elements of a range
31. Cuckoo Hashing – Worst case $O(1)$ Lookup!
32. Implementing our Own Hash Table with Separate Chaining in Java
33. Count pairs with given sum
34. Convert an array to reduced form

[Recent Articles on Hashing](#)

[Coding Practice on Hashing](#)

[Quiz on Hashing](#)

[Ask a Question](#)

[Data Structures](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

[Company Wise Coding Practice](#) [Topic Wise Coding Practice](#)

Introduction, DFS and BFS:

1. Graph and its representations
2. Breadth First Traversal for a Graph
3. Depth First Traversal for a Graph
4. Applications of Depth First Search
5. Applications of Breadth First Traversal
6. Longest Path in a Directed Acyclic Graph
7. Find Mother Vertex in a Graph
8. Transitive Closure of a Graph using DFS
9. Find K cores of an undirected Graph
10. Iterative Depth First Search

11. Iterative Deepening Search(IDS) or Iterative Deepening Depth First Search(IDDFS)

Graph Cycle:

1. Detect Cycle in a Directed Graph
2. Detect Cycle in a an Undirected Graph
3. Detect cycle in an undirected graph
4. Detect cycle in a direct graph using colors
5. Assign directions to edges so that the directed graph remains acyclic

Topological Sorting:

1. Topological Sorting
2. All topological sorts of a Directed Acyclic Graph
3. Kahn's Algorithm for Topological Sorting

Minimum Spanning Tree:

1. Prim's Minimum Spanning Tree (MST)
2. Applications of Minimum Spanning Tree Problem
3. Prim's MST for Adjacency List Representation
4. Kruskal's Minimum Spanning Tree Algorithm
5. Boruvka's algorithm for Minimum Spanning Tree
6. Steiner Tree

BackTracking

1. Find if there is a path of more than k length from a source
2. Tug of War
3. The Knight-Tour Problem
4. Rat in a Maze
5. n-Queen's Problem
6. m Coloring Problem
7. Hamiltonian Cycle

Shortest Paths:

1. Dijkstra's shortest path algorithm
2. Dijkstra's Algorithm for Adjacency List Representation
3. Bellman-Ford Algorithm
4. Floyd Warshall Algorithm
5. Johnson's algorithm for All-pairs shortest paths
6. Shortest Path in Directed Acyclic Graph
7. Some interesting shortest path questions,
8. Shortest path with exactly k edges in a directed and weighted graph
9. Dial's Algorithm
10. Printing paths in Dijkstra's Algorithm
11. Shortest path of a weighted graph where weight is 1 or 2

Connectivity:

1. Find if there is a path between two vertices in a directed graph
2. Connectivity in a directed graph
3. Articulation Points (or Cut Vertices) in a Graph
4. Biconnected graph
5. Bridges in a graph
6. Eulerian path and circuit
7. Fleury's Algorithm for printing Eulerian Path or Circuit
8. Strongly Connected Components
9. Transitive closure of a graph
10. Find the number of islands
11. Count all possible walks from a source to a destination with exactly k edges
12. Euler Circuit in a Directed Graph
13. Biconnected Components
14. Check if a given graph is tree or not
15. Karger's algorithm for Minimum Cut
16. Find if there is a path of more than k length
17. Length of shortest chain to reach the target word
18. Print all paths from a given source to destination
19. Find minimum cost to reach destination using train
20. Tarjan's Algorithm to find strongly connected Components

Hard Problems:

1. Graph Coloring (Introduction and Applications)
2. Greedy Algorithm for Graph Coloring
3. Travelling Salesman Problem (Naive and Dynamic Programming)
4. Travelling Salesman Problem (Approximate using MST)
5. Hamiltonian Cycle
6. Vertex Cover Problem | Set 1 (Introduction and Approximate Algorithm)
7. K Centers Problem | Set 1 (Greedy Approximate Algorithm)

Maximum Flow:

1. Ford-Fulkerson Algorithm for Maximum Flow Problem
2. Find maximum number of edge disjoint paths between two vertices
3. Find minimum s-t cut in a flow network
4. Maximum Bipartite Matching
5. Channel Assignment Problem
6. Push Relabel- Set 1-Introduction
7. Push Relabel- Set 2- Implementation
8. Karger's Algorithm- Set 1- Introduction and Implementation
9. Karger's Algorithm- Set 2 – Analysis and Applications

STL Implementation of Algorithms

1. Kruskal's Minimum Spanning Tree using STL in C++

2. Prim's Algorithm using Priority Queue STL
3. Dijkstra's Shortest Path Algorithm using STL
4. Dijkstra's Shortest Path Algorithm using set in STL

Misc

1. Number of triangles in an undirected Graph
2. Number of triangles in directed and undirected Graph
3. Check whether a given graph is Bipartite or not
4. Snake and Ladder Problem
5. Minimize Cash Flow among a given set of friends who have borrowed money from each other
6. Boggle (Find all possible words in a board of characters)
7. Hopcroft Karp Algorithm for Maximum Matching-Introduction
8. Hopcroft Karp Algorithm for Maximum Matching-Implementation
9. Minimum Time to rot all oranges
10. Find same contents in a list of contacts
11. Optimal read list for a given number of days
12. Print all jumping numbers smaller than or equal to a given value

Quiz on Graph

Quiz on Graph Traversals

Quiz on Graph Shortest Paths

Quiz on Graph Minimum Spanning Tree

Coding Practice on Graph

Company Wise Coding Practice Topic Wise Coding Practice

Array:

1. Given an array A[] and a number x, check for pair in A[] with sum as x
2. Majority Element
3. Find the Number Occurring Odd Number of Times
4. Largest Sum Contiguous Subarray
5. Find the Missing Number
6. Search an element in a sorted and pivoted array
7. Merge an array of size n into another array of size m+n
8. Median of two sorted arrays
9. Write a program to reverse an array
10. Program for array rotation
11. Reversal algorithm for array rotation
12. Block swap algorithm for array rotation
13. Maximum sum such that no two elements are adjacent
14. Leaders in an array
15. Sort elements by frequency | Set 1
16. Count Inversions in an array
17. Two elements whose sum is closest to zero
18. Find the smallest and second smallest element in an array
19. Check for Majority Element in a sorted array
20. Maximum and minimum of an array using minimum number of comparisons
21. Segregate 0s and 1s in an array
22. k largest(or smallest) elements in an array | added Min Heap method
23. Maximum difference between two elements
24. Union and Intersection of two sorted arrays
25. Floor and Ceiling in a sorted array
26. A Product Array Puzzle
27. Segregate Even and Odd numbers
28. Find the two repeating elements in a given array
29. Sort an array of 0s, 1s and 2s
30. Find the Minimum length Unsorted Subarray, sorting which makes the complete array sorted
31. Find duplicates in O(n) time and O(1) extra space
32. Equilibrium index of an array
33. Linked List vs Array
34. Which sorting algorithm makes minimum number of memory writes?
35. Turn an image by 90 degree
36. Next Greater Element
37. Check if array elements are consecutive | Added Method 3
38. Find the smallest missing number
39. Count the number of occurrences in a sorted array
40. Interpolation search vs Binary search
41. Given an array arr[], find the maximum j – i such that arr[j] > arr[i]
42. Maximum of all subarrays of size k (Added a O(n) method)
43. Find whether an array is subset of another array | Added Method 3
44. Find the minimum distance between two numbers
45. Find the repeating and the missing | Added 3 new methods
46. Median in a stream of integers (running integers)
47. Find a Fixed Point in a given array
48. Maximum Length Bitonic Subarray
49. Find the maximum element in an array which is first increasing and then decreasing
50. Count smaller elements on right side
51. Minimum number of jumps to reach end
52. Implement two stacks in an array
53. Find subarray with given sum
54. Dynamic Programming | Set 14 (Maximum Sum Increasing Subsequence)
55. Longest Monotonically Increasing Subsequence Size (N log N)
56. Find a triplet that sum to a given value
57. Find the smallest positive number missing from an unsorted array
58. Find the two numbers with odd occurrences in an unsorted array
59. The Celebrity Problem
60. Dynamic Programming | Set 15 (Longest Bitonic Subsequence)

61. Find a sorted subsequence of size 3 in linear time
62. Largest subarray with equal number of 0s and 1s
63. Dynamic Programming | Set 18 (Partition problem)
64. Maximum Product Subarray
65. Find a pair with the given difference
66. Replace every element with the next greatest
67. Dynamic Programming | Set 20 (Maximum Length Chain of Pairs)
68. Find four elements that sum to a given value | Set 1 (n^3 solution)
69. Find four elements that sum to a given value | Set 2 ($O(n^2 \log n)$ Solution)
70. Sort a nearly sorted (or K sorted) array
71. Maximum circular subarray sum
72. Find the row with maximum number of 1s
73. Median of two sorted arrays of different sizes
74. Shuffle a given array
75. Count the number of possible triangles
76. Iterative Quick Sort
77. Find the number of islands
78. Construction of Longest Monotonically Increasing Subsequence ($N \log N$)
79. Find the first circular tour that visits all petrol pumps
80. Arrange given numbers to form the biggest number
81. Pancake sorting
82. A Pancake Sorting Problem
83. Tug of War
84. Divide and Conquer | Set 3 (Maximum Subarray Sum)
85. Counting Sort
86. Merge Overlapping Intervals
87. Find the maximum repeating number in $O(n)$ time and $O(1)$ extra space
88. Stock Buy Sell to Maximize Profit
89. Rearrange positive and negative numbers in $O(n)$ time and $O(1)$ extra space
90. Sort elements by frequency | Set 2
91. Find a peak element
92. Print all possible combinations of r elements in a given array of size n
93. Given an array of size n and a number k , find all elements that appear more than n/k times
94. Find the point where a monotonically increasing function becomes positive first time
95. Find the Increasing subsequence of length three with maximum product
96. Find the minimum element in a sorted and rotated array
97. Stable Marriage Problem
98. Merge k sorted arrays | Set 1
99. Radix Sort
100. Move all zeroes to end of array
101. Find number of pairs such that $x^y > y^x$
102. Count all distinct pairs with difference equal to k
103. Find if there is a subarray with 0 sum
104. Smallest subarray with sum greater than a given value
105. Sort an array according to the order defined by another array
106. Maximum Sum Path in Two Arrays
107. Sort an array in wave form
108. K 'th Smallest/Largest Element in Unsorted Array
109. K 'th Smallest/Largest Element in Unsorted Array in Expected Linear Time
110. K 'th Smallest/Largest Element in Unsorted Array in Worst Case Linear Time
111. Find Index of 0 to be replaced with 1 to get longest continuous sequence of 1s in a binary array
112. Find the closest pair from two sorted arrays
113. Given a sorted array and a number x , find the pair in array whose sum is closest to x
114. Count 1's in a sorted binary array
115. Print All Distinct Elements of a given integer array
116. Construct an array from its pair-sum array
117. Find common elements in three sorted arrays
118. Find the first repeating element in an array of integers
119. Find the smallest positive integer value that cannot be represented as sum of any subset of a given array
120. Rearrange an array such that 'arr[j]' becomes 'j' if 'arr[j]' is 'j'
121. Find position of an element in a sorted array of infinite numbers
122. Can QuickSort be implemented in $O(n \log n)$ worst case time complexity?
123. Check if a given array contains duplicate elements within k distance from each other
124. Find the element that appears once
125. Replace every array element by multiplication of previous and next
126. Check if any two intervals overlap among a given set of intervals
127. Delete an element from array (Using two traversals and one traversal)
128. Find the largest pair sum in an unsorted array
129. Online algorithm for checking palindrome in a stream
130. Pythagorean Triplet in an array
131. Maximum profit by buying and selling a share at most twice
132. Find Union and Intersection of two unsorted Arrays
133. Count frequencies of all elements in array in $O(1)$ extra space and $O(n)$ time
134. Generate all possible sorted arrays from alternate elements of two given sorted arrays
135. Minimum number of swaps required for arranging pairs adjacent to each other
136. Trapping Rain Water
137. Convert array into Zig-Zag fashion
138. Find maximum average subarray of k length
139. Find maximum value of $\text{Sum}(i^{\text{arr}[i]})$ with only rotations on given array allowed
140. Reorder an array according to given indexes
141. Find zeroes to be flipped so that number of consecutive 1's is maximized
142. Count triplets with sum smaller than a given value
143. Find the subarray with least average
144. Count Inversions of size three in a give array
145. Longest Span with same Sum in two Binary arrays
146. Merge two sorted arrays with $O(1)$ extra space
147. Form minimum number from given sequence
148. Subarray/Substring vs Subsequence and Programs to Generate them
149. Count Strictly Increasing Subarrays
150. Rearrange an array in maximum minimum form
151. Find minimum difference between any two elements
152. Find lost element from a duplicated array
153. Count pairs with given sum
154. Count minimum steps to get the given desired array
155. Find minimum number of merge operations to make an array palindrome

Company Wise Coding Practice Topic Wise Coding Practice

Matrix:

- Search in a row wise and column wise sorted matrix
- Print a given matrix in spiral form
- A Boolean Matrix Question
- Print unique rows in a given boolean matrix
- Maximum size square sub-matrix with all 1s
- Inplace M x N size matrix transpose | Updated
- Print Matrix Diagonally
- Dynamic Programming | Set 27 (Maximum sum rectangle in a 2D matrix)
- Strassen's Matrix Multiplication
- Create a matrix with alternating rectangles of O and X
- Find the row with maximum number of 1s
- Print all elements in sorted order from row and column wise sorted matrix
- Given an n x n square matrix, find sum of all sub-squares of size k x k
- Count number of islands where every island is row-wise and column-wise separated
- Given a matrix of 'O' and 'X', replace 'O' with 'X' if surrounded by 'X'
- Find the longest path in a matrix with given constraints
- Given a Boolean Matrix, find k such that all elements in k'th row are 0 and k'th column are 1.
- Find the largest rectangle of 1's with swapping of columns allowed
- Validity of a given Tic-Tac-Toe board configuration
- Minimum Initial Points to Reach Destination
- Find length of the longest consecutive path from a given starting character
- Collect maximum points in a grid using two traversals
- Rotate Matrix Elements
- Find sum of all elements in a matrix except the elements in row and/or column of given cell?
- Find a common element in all rows of a given row-wise sorted matrix
- Number of paths with exactly k coins
- Collect maximum coins before hitting a dead end
- Program for Rank of Matrix
- Submatrix Sum Queries
- Maximum size rectangle binary sub-matrix with all 1s
- Count Negative Numbers in a Column-Wise and Row-Wise Sorted Matrix
- Construct Ancestor Matrix from a Given Binary Tree
- Construct tree from ancestor matrix
- In-place convert matrix in specific order
- Common elements in all rows of a given matrix
- Print maximum sum square sub-matrix of given size
- Find a specific pair in Matrix
- Find orientation of a pattern in a matrix
- Shortest path in a Binary Maze
- Inplace rotate square matrix by 90 degrees
- Return previous element in an expanding matrix

Company Wise Coding Practice Topic Wise Coding Practice

Overview of Data Structures | Set 1 (Linear Data Structures)

A data structure is a particular way of organizing data in a computer so that it can be used effectively. The idea is to reduce the space and time complexities of different tasks. Below is an overview of some popular linear data structures.

1. **Array**
2. **Linked List**
3. **Stack**
4. **Queue**

Array

Array is a data structure used to store homogeneous elements at contiguous locations. Size of an array must be provided before storing data.

Let size of array be n.
 Accessing Time: $O(1)$ [This is possible because elements are stored at contiguous locations]
 Search Time: $O(n)$ for Sequential Search:
 $O(\log n)$ for Binary Search [If Array is sorted]
 Insertion Time: $O(n)$ [The worst case occurs when insertion happens at the Beginning of an array and requires shifting all of the elements]
 Deletion Time: $O(n)$ [The worst case occurs when deletion happens at the Beginning of an array and requires shifting all of the elements]

Example : For example, let us say, we want to store marks of all students in a class, we can use an array to store them. This helps in reducing the use of number of variables as we don't need to create a separate variable for marks of every subject. All marks can be accessed by simply traversing the array.

Linked List

A linked list is a linear data structure (like arrays) where each element is a separate object. Each element (that is node) of a list is comprising of two items – the data and a reference to the next node.

Types of Linked List :

1. **Singly Linked List :** In this type of linked list, every node stores address or reference of next node in list and the last node has next address or reference as NULL. For example 1->2->3->4->NULL
2. **Doubly Linked List :** In this type of Linked list, there are two references associated with each node. One of the reference points to the next node and one to the previous node. Advantage of this data structure is that we can traverse in both the directions and for deletion we don't need to have explicit access to previous node. Eg. NULL23->NULL
3. **Circular Linked List :** Circular linked list is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list. Advantage of this data structure is that any node can be made as starting node. This is useful in implementation of circular queue in linked list. Eg. 1->2->3->1 [The next pointer of last node is pointing to the first]

Accessing time of an element : $O(n)$
 Search time of an element : $O(n)$
 Insertion of an Element : $O(1)$ [If we are at the position where we have to insert an element]
 Deletion of an Element : $O(1)$ [If we know address of node previous the node to be]

deleted]

Example : Consider the previous example where we made an array of marks of student. Now if a new subject is added in the course, its marks also to be added in the array of marks. But the size of the array was fixed and it is already full so it can not add any new element. If we make an array of a size lot more than the number of subjects it is possible that most of the array will remain empty. We reduce the space wastage Linked List is formed which adds a node only when a new element is introduced. Insertions and deletions also become easier with linked list.

One big drawback of linked list is, random access is not allowed. With arrays, we can access ith element in $O(1)$ time. In linked list, it takes $O(i)$ time.

Stack

A stack or LIFO (last in, first out) is an abstract data type that serves as a collection of elements, with two principal operations: push, which adds an element to the collection, and pop, which removes the last element that was added. In stack both the operations of push and pop takes place at the same end that is top of the stack. It can be implemented by using both array and linked list.

Insertion : $O(1)$
Deletion : $O(1)$
Access Time : $O(n)$ [Worst Case]
Insertion and Deletion are allowed on one end.

Example : Stacks are used for maintaining function calls (the last called function must finish execution first), we can always remove recursion with the help of stacks. Stacks are also used in cases where we have to reverse a word, check for balanced parenthesis and in editors where the word you typed the last is the first to be removed when you use undo operation. Similarly, to implement back functionality in web browsers.

Queue

A queue or FIFO (first in, first out) is an abstract data type that serves as a collection of elements, with two principal operations: enqueue, the process of adding an element to the collection. (The element is added from the rear side) and dequeue, the process of removing the first element that was added. (The element is removed from the front side). It can be implemented by using both array and linked list.

Insertion : $O(1)$
Deletion : $O(1)$
Access Time : $O(n)$ [Worst Case]

Example : Queue as the name says is the data structure built according to the queues of bus stop or train where the person who is standing in the front of the queue (standing for the longest time) is the first one to get the ticket. So any situation where resources are shared among multiple users and served on first come first server basis. Examples include CPU scheduling, Disk Scheduling. Another application of queue is when data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.

Circular Queue The advantage of this data structure is that it reduces wastage of space in case of array implementation, As the insertion of the $(n+1)$ th element is done at the 0th index if it is empty.

[Overview of Data Structures](#) | [Set 2 \(Binary Tree, BST, Heap and Hash\)](#)

This article is contributed by **Abhiraj Smit**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Misc

Overview of Data Structures | Set 2 (Binary Tree, BST, Heap and Hash)

We have discussed [Overview of Array](#), [Linked List](#), [Queue](#) and [Stack](#). In this article following Data Structures are discussed.

5. [Binary Tree](#)
6. [Binary Search Tree](#)
7. [Binary Heap](#)
9. [Hashing](#)

Binary Tree

Unlike Arrays, Linked Lists, Stack and queues, which are linear data structures, trees are hierarchical data structures.

A binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child. It is implemented mainly using Links.

Binary Tree Representation: A tree is represented by a pointer to the topmost node in tree. If the tree is empty, then value of root is NULL. A Binary Tree node contains following parts.

1. Data
2. Pointer to left child
3. Pointer to right child

A Binary Tree can be traversed in two ways:

Depth First Traversal: Inorder (Left-Root-Right), Preorder (Root-Left-Right) and Postorder (Left-Right-Root)

Breadth First Traversal: Level Order Traversal

Binary Tree Properties:

The maximum number of nodes at level 'l' = 2^{l-1} .

Maximum number of nodes = $2^h - 1$.
Here h is height of a tree. Height is considered as is maximum number of nodes on root to leaf path

Minimum possible height = $\text{ceil}(\text{Log}_2(n+1))$

In Binary tree, number of leaf nodes is always one more than nodes with two children.

Time Complexity of Tree Traversal: $O(n)$

Examples : One reason to use binary tree or tree in general is for the things that form a hierarchy. They are useful in File structures where each file is located in a particular directory and there is a specific hierarchy associated with files and directories. Another example where Trees are useful is storing hierarchical objects like JavaScript Document Object Model considers HTML page as a tree with nesting of tags as parent child relations.

Binary Search Tree

In Binary Search Tree is a Binary Tree with following additional properties:

1. The left subtree of a node contains only nodes with keys less than the node's key.
2. The right subtree of a node contains only nodes with keys greater than the node's key.
3. The left and right subtree each must also be a binary search tree.

Time Complexities:

Search : $O(h)$
Insertion : $O(h)$
Deletion : $O(h)$
Extra Space : $O(n)$ for pointers

h: Height of BST
n: Number of nodes in BST

If Binary Search Tree is Height Balanced,
then $h = O(\text{Log } n)$

Self-Balancing BSTs such as AVL Tree, Red-Black Tree and Splay Tree make sure that height of BST remains $O(\text{Log } n)$

BST provide moderate access/search (quicker than Linked List and slower than arrays).

BST provide moderate insertion/deletion (quicker than Arrays and slower than Linked Lists).

Examples : Its main use is in search application where data is constantly entering/leaving and data needs to be printed in sorted order. For example in implementation in E-commerce websites where a new product is added or product goes out of stock and all products are listed in sorted order.

Binary Heap

A Binary Heap is a Binary Tree with following properties.

- 1) It's a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array.
- 2) A Binary Heap is either Min Heap or Max Heap. In a Min Binary Heap, the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree. Max Binary Heap is similar to Min Heap. It is mainly implemented using array.

```
Get Minimum in Min Heap: O(1) [Or Get Max in Max Heap]
Extract Minimum Min Heap: O(Log n) [Or Extract Max in Max Heap]
Decrease Key in Min Heap: O(Log n) [Or Extract Max in Max Heap]
Insert: O(Log n)
Delete: O(Log n)
```

Example : Used in implementing efficient priority-queues, which in turn are used for scheduling processes in operating systems. Priority Queues are also used in Dijkstra's and Prim's graph algorithms.

Order statistics: The Heap data structure can be used to efficiently find the k'th smallest (or largest) element in an array.

Heap is a special data structure and it cannot be used for searching of a particular element.

Hashing

Hash Function: A function that converts a given big input key to a small practical integer value. The mapped integer value is used as an index in hash table. A good hash function should have following properties

- 1) Efficiently computable.
- 2) Should uniformly distribute the keys (Each table position equally likely for each key)

Hash Table: An array that stores pointers to records corresponding to a given phone number. An entry in hash table is NIL if no existing phone number has hash function value equal to the index for the entry.

Collision Handling: Since a hash function gets us a small number for a key which is a big integer or string, there is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some collision handling technique. Following are the ways to handle collisions:

Chaining: The idea is to make each cell of hash table point to a linked list of records that have same hash function value. Chaining is simple, but requires additional memory outside the table.

Open Addressing: In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we one by one examine table slots until the desired element is found or it is clear that the element is not in the table.

```
Space : O(n)
Search : O(1) [Average] O(n) [Worst case]
Insertion : O(1) [Average] O(n) [Worst Case]
Deletion : O(1) [Average] O(n) [Worst Case]
```

Hashing seems better than BST for all the operations. But in hashing, elements are unordered and in BST elements are stored in an ordered manner. Also BST is easy to implement but hash functions can sometimes be very complex to generate. In BST, we can also efficiently find floor and ceil of values.

Example : Hashing can be used to remove duplicates from a set of elements. Can also be used to find frequency of all items. For example, in web browsers, we can check visited URLs using hashing. In firewalls, we can use hashing to detect spam. We need to hash IP addresses. Hashing can be used in any situation where we want search() insert() and delete() in O(1) time.

This article is contributed by **Abhiraj Smit**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Misc
Self-Balancing-BST

Overview of Data Structures | Set 3 (Graph, Trie, Segment Tree and Suffix Tree)

We have discussed below data structures in previous two sets.

Set 1 : Overview of Array, Linked List, Queue and Stack.

Set 2 : Overview of Binary Tree, BST, Heap and Hash.

9. Graph

10. Trie

11. Segment Tree

12. Suffix Tree

Graph

Graph is a data structure that consists of following two components:

1. A finite set of vertices also called as nodes.
2. A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not same as (v, u) in case of directed graph (di-graph). The pair of form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

V -> Number of Vertices.

E -> Number of Edges.

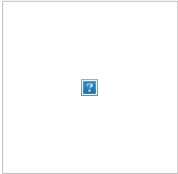
Graph can be classified on the basis of many things, below are the two most common classifications :

1. **Direction :**
Undirected Graph : The graph in which all the edges are bidirectional. Directed Graph : The graph in which all the edges are unidirectional.
2. **Weight :**
Weighted Graph : The Graph in which weight is associated with the edges. Unweighted Graph : The Graph in which there is no weight associated to the edges.

Graph can be represented in many ways, below are the two most common representations :

Let us take below example graph two to see two representations of graph.



1. 
Adjacency Matrix Representation of the above graph

Adjacency List Representation of Graph



2.

Adjacency List Representation of the above Graph

Time Complexities in case of Adjacency Matrix :

Traversal : (By BFS or DFS) $O(V^2)$

Space : $O(V^2)$

Time Complexities in case of Adjacency List :

Traversal : (By BFS or DFS) $O(E \log V)$

Space : $O(V+E)$

Examples : The most common example of the graph is to find shortest path in any network. Used in google maps or bing. Another common use application of graph are social networking websites where the friend suggestion depends on number of intermediate suggestions and other things.

Trie

Trie is an efficient data structure for searching words in dictionaries, search complexity with Trie is linear in terms of word (or key) length to be searched. If we store keys in binary search tree, a well balanced BST will need time proportional to $M \cdot \log N$, where M is maximum string length and N is number of keys in tree. Using trie, we can search the key in $O(M)$ time. So it is much faster than BST.

Hashing also provides word search in $O(n)$ time on average. But the advantages of Trie are there are no collisions (like hashing) so worst case time complexity is $O(n)$. Also, the most important thing is Prefix Search. With Trie, we can find all words beginning with a prefix (This is not possible with Hashing). The only problem with Tries is they require a lot of extra space. Tries are also known as radix tree or prefix tree.

The Trie structure can be defined as follows :

```
struct trie_node
{
    int value; /* Used to mark leaf nodes */
    trie_node_t* children[ALPHABET_SIZE];
};
```

```

      root
      /  \  \
     t  a  b
      |  |  |
      h  n  y
      |  |  |
      e  s  e
      /  |  |
     i  r  w
      |  |  |
     r  e  e
      |
      r
```

The leaf nodes are in blue.

Insert time : $O(M)$ where M is the length of the string.

Search time : $O(M)$ where M is the length of the string.

Space : $O(ALPHABET_SIZE \cdot M \cdot N)$ where N is number of

keys in trie, ALPHABET_SIZE is 26 if we are

only considering upper case Latin characters.

Deletion time : $O(M)$

Example : The most common use of Tries is to implement dictionaries due to prefix search capability. Tries are also well suited for implementing approximate matching algorithms, including those used in spell checking. It is also used for searching Contact from Mobile Contact list OR Phone Directory.

Segment Tree

This data structure is usually implemented when there are a lot of queries on a set of values. These queries involve minimum, maximum, sum, .. etc on an input range of given set. Queries also involve updation of values in given set. Segment Trees are implemented using array.



Construction of segment tree : $O(N)$

Query : $O(\log N)$

Update : $O(\log N)$

Space : $O(N)$ [Exact space = $2 \cdot N - 1$]

Example : It is used when we need to find Maximum/Minimum/Sum/Product of numbers in a range.

Suffix Tree

Suffix Tree is mainly used to search a pattern in a text. The idea is to preprocess the text so that search operation can be done in time linear in terms of pattern length. The pattern searching algorithms like KMP, Z, etc take time proportional to text length. This is really a great improvement because length of pattern is generally much smaller than text.

Imagine we have stored complete work of William Shakespeare and preprocessed it. You can search any string in the complete work in time just proportional to length of the pattern. But using Suffix Tree may not be a good idea when text changes frequently like text editor, etc.

Suffix Tree is compressed trie of all suffixes, so following are very abstract steps to build a suffix tree from given text.

1) Generate all suffixes of given text.

2) Consider all suffixes as individual words and build a compressed trie.



Example : Used to find all occurrences of the pattern in string. It is also used to find the longest repeated substring (when test doesn't change often), the longest common substring and the longest palindrome in a string.

This article is contributed by **Abhiraj Smit**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Segment-Tree
Suffix-Tree
TRIE

Linked List | Set 1 (Introduction)

Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at contiguous location; the elements are linked using pointers.

linkedlist



Why Linked List?

Arrays can be used to store linear data of similar types, but arrays have following limitations.

- 1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.
- 2) Inserting a new element in an array of elements is expensive, because room has to be created for the new elements and to create room existing elements have to be shifted.

For example, in a system if we maintain a sorted list of IDs in an array `id[]`.

`id[] = [1000, 1010, 1050, 2000, 2040]`.

And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).

Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in `id[]`, everything after 1010 has to be moved.

Advantages over arrays

- 1) Dynamic size
- 2) Ease of insertion/deletion

Drawbacks:

- 1) Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists.
- 2) Extra memory space for a pointer is required with each element of the list.

Representation in C:

A linked list is represented by a pointer to the first node of the linked list. The first node is called head. If the linked list is empty, then value of head is NULL.

Each node in a list consists of at least two parts:

- 1) data
- 2) pointer to the next node

In C, we can represent a node using structures. Below is an example of a linked list node with an integer data.

In Java, `LinkedList` can be represented as a class and a `Node` as a separate class. The `LinkedList` class contains a reference of `Node` class type.

C

```
// A linked list node
struct node
{
    int data;
    struct node *next;
};
```

Java

```
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;

        // Constructor to create a new node
        // Next is by default initialized as null
        Node(int d) {data = d;}
    }
}
```

Python

```
# Node class
class Node:

    # Function to initialize the node object
    def __init__(self, data):
        self.data = data # Assign data
        self.next = None # Initialize next as null

# Linked List class
class LinkedList:

    # Function to initialize the Linked List object
    def __init__(self):
        self.head = None
```

First Simple Linked List in C Let us create a simple linked list with 3 nodes.

C

```
// A simple C program to introduce a linked list
#include<stdio.h>
#include<stdlib.h>

struct node
{
    int data;
    struct node *next;
};

// Program to create a simple linked list with 3 nodes
int main()
{
    struct node* head = NULL;
    struct node* second = NULL;
    struct node* third = NULL;

    // allocate 3 nodes in the heap
    head = (struct node*)malloc(sizeof(struct node));
    second = (struct node*)malloc(sizeof(struct node));
    third = (struct node*)malloc(sizeof(struct node));

    /* Three blocks have been allocated dynamically.
    We have pointers to these three blocks as first, second and third
    head      second      third
    |          |          |
    |          |          |
    +-----+ +-----+ +-----+
    | # | # | | # | # | | # | # |
    +-----+ +-----+ +-----+

    # represents any random value.
    Data is random because we haven't assigned anything yet */

    head->data = 1; //assign data in first node
    head->next = second; // Link first node with the second node

    /* data has been assigned to data part of first block (block
    pointed by head). And next pointer of first block points to
    second. So they both are linked.

    head      second      third
    |          |          |
    |          |          |
    +-----+ +-----+ +-----+
    | 1 | 0----->| # | # | | # | # |
    +-----+ +-----+ +-----+
    */

    second->data = 2; //assign data to second node
    second->next = third; // Link second node with the third node

    /* data has been assigned to data part of second block (block pointed by
    second). And next pointer of the second block points to third block.
    So all three blocks are linked.

    head      second      third
    |          |          |
    |          |          |
    +-----+ +-----+ +-----+
    | 1 | 0----->| 2 | 0----->| # | # |
    +-----+ +-----+ +-----+ */

    third->data = 3; //assign data to third node
    third->next = NULL;

    /* data has been assigned to data part of third block (block pointed
    by third). And next pointer of the third block is made NULL to indicate
    that the linked list is terminated here.

    We have the linked list ready.

    head
    |
    |
    +-----+ +-----+ +-----+
    | 1 | 0----->| 2 | 0----->| 3 | NULL |
    +-----+ +-----+ +-----+

    Note that only head is sufficient to represent the whole list. We can
    traverse the complete list by following next pointers. */

    return 0;
}
```

Java

```
// A simple Java program to introduce a linked list
class LinkedList
{
    Node head; // head of list

    /* Linked list Node. This inner class is made static so that
    main() can access it */
    static class Node {
        int data;
        Node next;
        Node(int d) { data = d; next=null; } // Constructor
    }

    /* method to create a simple linked list with 3 nodes*/
    public static void main(String[] args)
    {
        /* Start with the empty list. */
        LinkedList llist = new LinkedList();

        llist.head = new Node(1);
        Node second = new Node(2);
        Node third = new Node(3);
    }
}
```

```

/* Three nodes have been allocated dynamically.
We have references to these three blocks as first,
second and third

l1list.head    second    third
|             |          |
|             |          |
+-----+-----+-----+
| 1 | null | | 2 | null | | 3 | null |
+-----+-----+-----+ */

l1list.head.next = second; // Link first node with the second node

/* Now next of first Node refers to second. So they
both are linked.

l1list.head    second    third
|             |          |
|             |          |
+-----+-----+-----+
| 1 | 0----->| 2 | null | | 3 | null |
+-----+-----+-----+ */

second.next = third; // Link second node with the third node

/* Now next of second Node refers to third. So all three
nodes are linked.

l1list.head    second    third
|             |          |
|             |          |
+-----+-----+-----+
| 1 | 0----->| 2 | 0----->| 3 | null |
+-----+-----+-----+ */
}
}

```

Python

```

# A simple Python program to introduce a linked list

# Node class
class Node:

    # Function to initialise the node object
    def __init__(self, data):
        self.data = data # Assign data
        self.next = None # Initialize next as null

# Linked List class contains a Node object
class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

# Code execution starts here
if __name__ == '__main__':

    # Start with the empty list
    l1list = LinkedList()

    l1list.head = Node(1)
    second = Node(2)
    third = Node(3)

    """
    Three nodes have been created.
    We have references to these three blocks as first,
    second and third

    l1list.head    second    third
    |             |          |
    |             |          |
    +-----+-----+-----+
    | 1 | None | | 2 | None | | 3 | None |
    +-----+-----+-----+
    """

    l1list.head.next = second; # Link first node with second

    """
    Now next of first Node refers to second. So they
    both are linked.

    l1list.head    second    third
    |             |          |
    |             |          |
    +-----+-----+-----+
    | 1 | 0----->| 2 | null | | 3 | null |
    +-----+-----+-----+
    """

    second.next = third; # Link second node with the third node

    """
    Now next of second Node refers to third. So all three
    nodes are linked.

    l1list.head    second    third
    |             |          |
    |             |          |
    +-----+-----+-----+
    | 1 | 0----->| 2 | 0----->| 3 | null |
    +-----+-----+-----+
    """
    """
    """

```

Linked List Traversal

In the previous program, we have created a simple linked list with three nodes. Let us traverse the created list and print the data of each node. For traversal, let us write a general purpose function `printList()` that prints any

given list.

We strongly recommend that you click [here](#) and practice it, before moving on to the solution.

C

```
// A simple C program for traversal of a linked list
#include<stdio.h>
#include<stdlib.h>

struct node
{
    int data;
    struct node *next;
};

// This function prints contents of linked list starting from
// the given node
void printList(struct node *n)
{
    while (n != NULL)
    {
        printf(" %d ", n->data);
        n = n->next;
    }
}

int main()
{
    struct node* head = NULL;
    struct node* second = NULL;
    struct node* third = NULL;

    // allocate 3 nodes in the heap
    head = (struct node*)malloc(sizeof(struct node));
    second = (struct node*)malloc(sizeof(struct node));
    third = (struct node*)malloc(sizeof(struct node));

    head->data = 1; //assign data in first node
    head->next = second; // Link first node with second

    second->data = 2; //assign data to second node
    second->next = third;

    third->data = 3; //assign data to third node
    third->next = NULL;

    printList(head);

    return 0;
}
```

Java

```
// A simple Java program for traversal of a linked list
class LinkedList
{
    Node head; // head of list

    /* Linked list Node. This inner class is made static so that
    main() can access it */
    static class Node {
        int data;
        Node next;
        Node(int d) { data = d; next=null; } // Constructor
    }

    /* This function prints contents of linked list starting from head */
    public void printList()
    {
        Node n = head;
        while (n != null)
        {
            System.out.print(n.data+" ");
            n = n.next;
        }
    }

    /* method to create a simple linked list with 3 nodes*/
    public static void main(String[] args)
    {
        /* Start with the empty list. */
        LinkedList llist = new LinkedList();

        llist.head = new Node(1);
        Node second = new Node(2);
        Node third = new Node(3);

        llist.head.next = second; // Link first node with the second node
        second.next = third; // Link first node with the second node

        llist.printList();
    }
}
```

Python

```
# A simple Python program for traversal of a linked list

# Node class
class Node:

    # Function to initialise the node object
    def __init__(self, data):
        self.data = data # Assign data
        self.next = None # Initialize next as null
```



```
# Linked List class contains a Node object
class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    # This function prints contents of linked list
    # starting from head
    def printList(self):
        temp = self.head
        while (temp):
            print temp.data,
            temp = temp.next

# Code execution starts here
if __name__ == '__main__':

    # Start with the empty list
    llist = LinkedList()

    llist.head = Node(1)
    second = Node(2)
    third = Node(3)

    llist.head.next = second; # Link first node with second
    second.next = third; # Link second node with the third node

    llist.printList()
```

Output:

1 2 3

You may like to try [Practice MCQ Questions on Linked List](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: [Linked List](#)

Linked List vs Array

Difficulty Level: Rookie

Both Arrays and [Linked List](#) can be used to store linear data of similar types, but they both have some advantages and disadvantages over each other.

Following are the points in favour of Linked Lists.

(1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage, and in practical uses, upper limit is rarely reached.

(2) Inserting a new element in an array of elements is expensive, because room has to be created for the new elements and to create room existing elements have to be shifted.

For example, suppose we maintain a sorted list of IDs in an array id[].

id[] = [1000, 1010, 1050, 2000, 2040,].

And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).

Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.

So Linked list provides following two advantages over arrays

- 1) Dynamic size
- 2) Ease of insertion/deletion

Linked lists have following drawbacks:

- 1) Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists.
- 2) Extra memory space for a pointer is required with each element of the list.
- 3) Arrays have better cache locality that can make a pretty big difference in performance.

References:

<http://cslibrary.stanford.edu/103/LinkedListBasics.pdf>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Arrays
Linked Lists

Linked List | Set 2 (Inserting a node)

We have introduced Linked Lists in the [previous post](#). We also created a simple linked list with 3 nodes and discussed linked list traversal.

All programs discussed in this post consider following representations of linked list .

C

```
// A linked list node
struct node
{
    int data;
    struct node *next;
};
```

Java

```
// Linked List Class
class LinkedList
{
    Node head; // head of list

    /* Node Class */
    class Node
    {
        int data;
        Node next;

        // Constructor to create a new node
        Node(int d) {data = d; next = null;}
    }
}
```

Python

```
# Node class
class Node:

    # Function to initialize the node object
    def __init__(self, data):
        self.data = data # Assign data
        self.next = None # Initialize next as null

# Linked List class
class LinkedList:

    # Function to initialize the Linked List object
    def __init__(self):
        self.head = None
```

In this post, methods to insert a new node in linked list are discussed. A node can be added in three ways

1) At the front of the linked list

2) After a given node.

3) At the end of the linked list.

Add a node at the front: (A 4 steps process)

The new node is always added before the head of the given Linked List. And newly added node becomes the new head of the Linked List. For example if the given Linked List is 10->15->20->25 and we add an item 5 at the front, then the Linked List becomes 5->10->15->20->25. Let us call the function that adds at the front of the list is push(). The push() must receive a pointer to the head pointer, because push must change the head pointer to point to the new node (See [this](#))

[linkedlist_insert_at_start](#)



Following are the 4 steps to add node at the front.

C

```
/* Given a reference (pointer to pointer) to the head of a list
and an int, inserts a new node on the front of the list. */
void push(struct node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct node* new_node = (struct node*) malloc(sizeof(struct node));

    /* 2. put in the data */
    new_node->data = new_data;

    /* 3. Make next of new node as head */
    new_node->next = (*head_ref);

    /* 4. move the head to point to the new node */
    (*head_ref) = new_node;
}
```

Java

```
/* This function is in LinkedList class. Inserts a
new Node at front of the list. This method is
defined inside LinkedList class shown above */
public void push(int new_data)
{
    /* 1 & 2: Allocate the Node &
    Put in the data*/
    Node new_node = new Node(new_data);

    /* 3. Make next of new Node as head */
    new_node.next = head;

    /* 4. Move the head to point to new Node */
    head = new_node;
}
```

Python

```
# This function is in LinkedList class
# Function to insert a new node at the beginning
def push(self, new_data):

    # 1 & 2: Allocate the Node &
    # Put in the data
    new_node = Node(new_data)

    # 3. Make next of new Node as head
    new_node.next = self.head

    # 4. Move the head to point to new Node
    self.head = new_node
```

Time complexity of push() is $O(1)$ as it does constant amount of work.

Add a node after a given node: (5 steps process)

We are given pointer to a node, and the new node is inserted after the given node.

linkedlist_insert_middle



C

```
/* Given a node prev_node, insert a new node after the given
prev_node */
void insertAfter(struct node* prev_node, int new_data)
{
    /*1. check if the given prev_node is NULL */
    if (prev_node == NULL)
    {
        printf("the given previous node cannot be NULL");
        return;
    }

    /* 2. allocate new node */
    struct node* new_node = (struct node*) malloc(sizeof(struct node));

    /* 3. put in the data */
    new_node->data = new_data;

    /* 4. Make next of new node as next of prev_node */
    new_node->next = prev_node->next;

    /* 5. move the next of prev_node as new_node */
    prev_node->next = new_node;
}
```

Java

```

/* This function is in LinkedList class.
Inserts a new node after the given prev_node. This method is
defined inside LinkedList class shown above */
public void insertAfter(Node prev_node, int new_data)
{
    /* 1. Check if the given Node is null */
    if (prev_node == null)
    {
        System.out.println("The given previous node cannot be null");
        return;
    }

    /* 2. Allocate the Node &
    3. Put in the data*/
    Node new_node = new Node(new_data);

    /* 4. Make next of new Node as next of prev_node */
    new_node.next = prev_node.next;

    /* 5. make next of prev_node as new_node */
    prev_node.next = new_node;
}

```

Python

```

# This function is in LinkedList class.
# Inserts a new node after the given prev_node. This method is
# defined inside LinkedList class shown above */
def insertAfter(self, prev_node, new_data):

    # 1. check if the given prev_node exists
    if prev_node is None:
        print "The given previous node must in LinkedList."
        return

    # 2. Create new node &
    # 3. Put in the data
    new_node = Node(new_data)

    # 4. Make next of new Node as next of prev_node
    new_node.next = prev_node.next

    # 5. make next of prev_node as new_node
    prev_node.next = new_node

```

Time complexity of insertAfter() is $O(1)$ as it does constant amount of work.

Add a node at the end: (6 steps process)

The new node is always added after the last node of the given Linked List. For example if the given Linked List is 5->10->15->20->25 and we add an item 30 at the end, then the Linked List becomes 5->10->15->20->25->30. Since a Linked List is typically represented by the head of it, we have to traverse the list till end and then change the next of last node to new node.

linkedList_insert_last



Following are the 6 steps to add node at the end.

C

```

/* Given a reference (pointer to pointer) to the head
of a list and an int, appends a new node at the end */
void append(struct node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct node* new_node = (struct node*) malloc(sizeof(struct node));

    struct node *last = *head_ref; /* used in step 5*/

    /* 2. put in the data */
    new_node->data = new_data;

    /* 3. This new node is going to be the last node, so make next
    of it as NULL*/
    new_node->next = NULL;

    /* 4. If the Linked List is empty, then make the new node as head */
    if (*head_ref == NULL)
    {
        *head_ref = new_node;
        return;
    }

    /* 5. Else traverse till the last node */
    while (last->next != NULL)
        last = last->next;

    /* 6. Change the next of last node */
    last->next = new_node;
    return;
}

```

Java

```

/* Appends a new node at the end. This method is
defined inside LinkedList class shown above */
public void append(int new_data)
{

```

```

/* 1. Allocate the Node &
2. Put in the data
3. Set next as null */
Node new_node = new Node(new_data);

/* 4. If the Linked List is empty, then make the
new node as head */
if (head == null)
{
    head = new Node(new_data);
    return;
}

/* 4. This new node is going to be the last node, so
make next of it as null */
new_node.next = null;

/* 5. Else traverse till the last node */
Node last = head;
while (last.next != null)
    last = last.next;

/* 6. Change the next of last node */
last.next = new_node;
return;
}

```

Python

```

# This function is defined in Linked List class
# Appends a new node at the end. This method is
# defined inside LinkedList class shown above */
def append(self, new_data):

    # 1. Create a new node
    # 2. Put in the data
    # 3. Set next as None
    new_node = Node(new_data)

    # 4. If the Linked List is empty, then make the
    # new node as head
    if self.head is None:
        self.head = new_node
        return

    # 5. Else traverse till the last node
    last = self.head
    while (last.next):
        last = last.next

    # 6. Change the next of last node
    last.next = new_node

```

Time complexity of append is $O(n)$ where n is the number of nodes in linked list. Since there is a loop from head to end, the function does $O(n)$ work. This method can also be optimized to work in $O(1)$ by keeping an extra pointer to tail of linked list/
Following is a complete program that uses all of the above methods to create a linked list.

C

```

// A complete working C program to demonstrate all insertion methods
// on Linked List
#include <stdio.h>
#include <stdlib.h>

// A linked list node
struct node
{
    int data;
    struct node *next;
};

/* Given a reference (pointer to pointer) to the head of a list and
an int, inserts a new node on the front of the list. */
void push(struct node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct node* new_node = (struct node*) malloc(sizeof(struct node));

    /* 2. put in the data */
    new_node->data = new_data;

    /* 3. Make next of new node as head */
    new_node->next = (*head_ref);

    /* 4. move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Given a node prev_node, insert a new node after the given
prev_node */
void insertAfter(struct node* prev_node, int new_data)
{
    /* 1. check if the given prev_node is NULL */
    if (prev_node == NULL)
    {
        printf("the given previous node cannot be NULL");
        return;
    }

    /* 2. allocate new node */
    struct node* new_node = (struct node*) malloc(sizeof(struct node));

    /* 3. put in the data */
    new_node->data = new_data;

    /* 4. Make next of new node as next of prev_node */
    new_node->next = prev_node->next;

    /* 5. move the next of prev_node as new_node */
    prev_node->next = new_node;
}

```

```

}

/* Given a reference (pointer to pointer) to the head
of a list and an int, appends a new node at the end */
void append(struct node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct node* new_node = (struct node*) malloc(sizeof(struct node));

    struct node *last = *head_ref; /* used in step 5*/

    /* 2. put in the data */
    new_node->data = new_data;

    /* 3. This new node is going to be the last node, so make next of
    it as NULL*/
    new_node->next = NULL;

    /* 4. If the Linked List is empty, then make the new node as head */
    if (*head_ref == NULL)
    {
        *head_ref = new_node;
        return;
    }

    /* 5. Else traverse till the last node */
    while (last->next != NULL)
        last = last->next;

    /* 6. Change the next of last node */
    last->next = new_node;
    return;
}

// This function prints contents of linked list starting from head
void printList(struct node *node)
{
    while (node != NULL)
    {
        printf(" %d ", node->data);
        node = node->next;
    }
}

/* Driver program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

    // Insert 6. So linked list becomes 6->NULL
    append(&head, 6);

    // Insert 7 at the beginning. So linked list becomes 7->6->NULL
    push(&head, 7);

    // Insert 1 at the beginning. So linked list becomes 1->7->6->NULL
    push(&head, 1);

    // Insert 4 at the end. So linked list becomes 1->7->6->4->NULL
    append(&head, 4);

    // Insert 8, after 7. So linked list becomes 1->7->8->6->4->NULL
    insertAfter(head->next, 8);

    printf("\n Created Linked list is: ");
    printList(head);

    return 0;
}

```

Java

```

// A complete working Java program to demonstrate all insertion methods
// on linked list
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;
        Node(int d) {data = d; next = null;}
    }

    /* Inserts a new Node at front of the list. */
    public void push(int new_data)
    {
        /* 1 & 2: Allocate the Node &
        Put in the data*/
        Node new_node = new Node(new_data);

        /* 3. Make next of new Node as head */
        new_node.next = head;

        /* 4. Move the head to point to new Node */
        head = new_node;
    }

    /* Inserts a new node after the given prev_node. */
    public void insertAfter(Node prev_node, int new_data)
    {
        /* 1. Check if the given Node is null */
        if (prev_node == null)
        {
            System.out.println("The given previous node cannot be null");
            return;
        }

        /* 2 & 3: Allocate the Node &
        Put in the data*/
    }
}

```

```

Node new_node = new Node(new_data);

/* 4. Make next of new Node as next of prev_node */
new_node.next = prev_node.next;

/* 5. make next of prev_node as new_node */
prev_node.next = new_node;
}

/* Appends a new node at the end. This method is
defined inside LinkedList class shown above */
public void append(int new_data)
{
    /* 1. Allocate the Node &
    2. Put in the data
    3. Set next as null */
    Node new_node = new Node(new_data);

    /* 4. If the Linked List is empty, then make the
    new node as head */
    if (head == null)
    {
        head = new Node(new_data);
        return;
    }

    /* 4. This new node is going to be the last node, so
    make next of it as null */
    new_node.next = null;

    /* 5. Else traverse till the last node */
    Node last = head;
    while (last.next != null)
        last = last.next;

    /* 6. Change the next of last node */
    last.next = new_node;
    return;
}

/* This function prints contents of linked list starting from
the given node */
public void printList()
{
    Node tnode = head;
    while (tnode != null)
    {
        System.out.print(tnode.data+" ");
        tnode = tnode.next;
    }
}

/* Driver program to test above functions. Ideally this function
should be in a separate user class. It is kept here to keep
code compact */
public static void main(String[] args)
{
    /* Start with the empty list */
    LinkedList llist = new LinkedList();

    // Insert 6. So linked list becomes 6->NULLlist
    llist.append(6);

    // Insert 7 at the beginning. So linked list becomes
    // 7->6->NULLlist
    llist.push(7);

    // Insert 1 at the beginning. So linked list becomes
    // 1->7->6->NULLlist
    llist.push(1);

    // Insert 4 at the end. So linked list becomes
    // 1->7->6->4->NULLlist
    llist.append(4);

    // Insert 8, after 7. So linked list becomes
    // 1->7->8->6->4->NULLlist
    llist.insertAfter(llist.head.next, 8);

    System.out.println("\nCreated Linked list is: ");
    llist.printList();
}
}
// This code is contributed by Rajat Mishra

```

Python

A complete working Python program to demonstrate all
insertion methods of linked list

Node class
class Node:

```

    # Function to initialise the node object
    def __init__(self, data):
        self.data = data # Assign data
        self.next = None # Initialize next as null

```

Linked List class contains a Node object
class LinkedList:

```

    # Function to initialize head
    def __init__(self):
        self.head = None

```

```

    # Function to insert a new node at the beginning
    def push(self, new_data):

```

```

        # 1 & 2: Allocate the Node &
        # Put in the data
        new_node = Node(new_data)

```

```

# 3. Make next of new Node as head
new_node.next = self.head

# 4. Move the head to point to new Node
self.head = new_node

# This function is in LinkedList class. Inserts a
# new node after the given prev_node. This method is
# defined inside LinkedList class shown above */
def insertAfter(self, prev_node, new_data):

    # 1. check if the given prev_node exists
    if prev_node is None:
        print "The given previous node must inLinkedList."
        return

    # 2. create new node &
    # Put in the data
    new_node = Node(new_data)

    # 4. Make next of new Node as next of prev_node
    new_node.next = prev_node.next

    # 5. make next of prev_node as new_node
    prev_node.next = new_node

# This function is defined in Linked List class
# Appends a new node at the end. This method is
# defined inside LinkedList class shown above */
def append(self, new_data):

    # 1. Create a new node
    # 2. Put in the data
    # 3. Set next as None
    new_node = Node(new_data)

    # 4. If the Linked List is empty, then make the
    # new node as head
    if self.head is None:
        self.head = new_node
        return

    # 5. Else traverse till the last node
    last = self.head
    while (last.next):
        last = last.next

    # 6. Change the next of last node
    last.next = new_node

# Utility function to print the linked list
def printList(self):
    temp = self.head
    while (temp):
        print temp.data,
        temp = temp.next

# Code execution starts here
if __name__ == '__main__':

    # Start with the empty list
    llist = LinkedList()

    # Insert 6. So linked list becomes 6->None
    llist.append(6)

    # Insert 7 at the beginning. So linked list becomes 7->6->None
    llist.push(7);

    # Insert 1 at the beginning. So linked list becomes 1->7->6->None
    llist.push(1);

    # Insert 4 at the end. So linked list becomes 1->7->6->4->None
    llist.append(4)

    # Insert 8, after 7. So linked list becomes 1 -> 7-> 8-> 6-> 4-> None
    llist.insertAfter(llist.head.next, 8)

    print 'Created linked list is:',
    llist.printList()

# This code is contributed by Manikantan Narasimhan

```

Output:

```
Created Linked list is: 1 7 8 6 4
```


You may like to try [Practice MCQ Questions on Linked List](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: [Linked List](#)

Linked List | Set 3 (Deleting a node)

We have discussed [Linked List Introduction](#) and [Linked List Insertion](#) in previous posts on singly linked list.

Let us formulate the problem statement to understand the deletion process. *Given a 'key', delete the first occurrence of this key in linked list.*

To delete a node from linked list, we need to do following steps.

- 1) Find previous node of the node to be deleted.
- 2) Changed next of previous node.
- 3) Free memory for the node to be deleted.

linkedList_deletion



Since every node of linked list is dynamically allocated using malloc() in C, we need to call [free\(\)](#) for freeing memory allocated for the node to be deleted.

C/C++

```
// A complete working C program to demonstrate deletion in singly
// linked list
#include <stdio.h>
#include <stdlib.h>

// A linked list node
struct node
{
    int data;
    struct node *next;
};

/* Given a reference (pointer to pointer) to the head of a list
and an int, inserts a new node on the front of the list. */
void push(struct node** head_ref, int new_data)
{
    struct node* new_node = (struct node*) malloc(sizeof(struct node));
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

/* Given a reference (pointer to pointer) to the head of a list
and a key, deletes the first occurrence of key in linked list */
void deleteNode(struct node** head_ref, int key)
{
    // Store head node
    struct node* temp = *head_ref, *prev;

    // If head node itself holds the key to be deleted
    if (temp != NULL && temp->data == key)
    {
        *head_ref = temp->next; // Changed head
        free(temp);           // free old head
        return;
    }

    // Search for the key to be deleted, keep track of the
    // previous node as we need to change 'prev->next'
    while (temp != NULL && temp->data != key)
    {
        prev = temp;
        temp = temp->next;
    }

    // If key was not present in linked list
    if (temp == NULL) return;

    // Unlink the node from linked list
    prev->next = temp->next;

    free(temp); // Free memory
}

// This function prints contents of linked list starting from
// the given node
void printList(struct node *node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Driver program to test above functions */
int main()
{
    /* Start with the empty list */
}
```

```

struct node* head = NULL;

push(&head, 7);
push(&head, 1);
push(&head, 3);
push(&head, 2);

puts("Created Linked List: ");
printList(head);
deleteNode(&head, 1);
puts("\nLinked List after Deletion of 1: ");
printList(head);
return 0;
}

```

Java

```

// A complete working Java program to demonstrate deletion in singly
// linked list
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;
        Node(int d)
        {
            data = d;
            next = null;
        }
    }

    /* Given a key, deletes the first occurrence of key in linked list */
    void deleteNode(int key)
    {
        // Store head node
        Node temp = head, prev = null;

        // If head node itself holds the key to be deleted
        if (temp != null && temp.data == key)
        {
            head = temp.next; // Changed head
            return;
        }

        // Search for the key to be deleted, keep track of the
        // previous node as we need to change temp.next
        while (temp != null && temp.data != key)
        {
            prev = temp;
            temp = temp.next;
        }

        // If key was not present in linked list
        if (temp == null) return;

        // Unlink the node from linked list
        prev.next = temp.next;
    }

    /* Inserts a new Node at front of the list. */
    public void push(int new_data)
    {
        Node new_node = new Node(new_data);
        new_node.next = head;
        head = new_node;
    }

    /* This function prints contents of linked list starting from
    the given node */
    public void printList()
    {
        Node tnode = head;
        while (tnode != null)
        {
            System.out.print(tnode.data+" ");
            tnode = tnode.next;
        }
    }

    /* Driver program to test above functions. Ideally this function
    should be in a separate user class. It is kept here to keep
    code compact */
    public static void main(String[] args)
    {
        LinkedList llist = new LinkedList();

        llist.push(7);
        llist.push(1);
        llist.push(3);
        llist.push(2);

        System.out.println("\nCreated Linked list is:");
        llist.printList();

        llist.deleteNode(1); // Delete node at position 4

        System.out.println("\nLinked List after Deletion at position 4:");
        llist.printList();
    }
}

```

Python

```

# Python program to delete a node from linked list

# Node class

```

```

class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    # Function to insert a new node at the beginning
    def push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        self.head = new_node

    # Given a reference to the head of a list and a key,
    # delete the first occurrence of key in linked list
    def deleteNode(self, key):

        # Store head node
        temp = self.head

        # If head node itself holds the key to be deleted
        if (temp is not None):
            if (temp.data == key):
                self.head = temp.next
                temp = None
                return

        # Search for the key to be deleted, keep track of the
        # previous node as we need to change 'prev.next'
        while (temp is not None):
            if temp.data == key:
                break
            prev = temp
            temp = temp.next

        # If key was not present in linked list
        if (temp == None):
            return

        # Unlink the node from linked list
        prev.next = temp.next

        temp = None

    # Utility function to print the linked LinkedList
    def printList(self):
        temp = self.head
        while (temp):
            print "%d" %(temp.data),
            temp = temp.next

# Driver program
l1 = LinkedList()
l1.push(7)
l1.push(1)
l1.push(3)
l1.push(2)

print "Created Linked List: "
l1.printList()
l1.deleteNode(1)
print "\nLinked List after Deletion of 1:"
l1.printList()

# This code is contributed by Nikhil Kumar Singh (nickzuck_007)

```

Output:

```

Created Linked List:
2 3 1 7
Linked List after Deletion of 1:
2 3 7

```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.

Delete a Linked List node at a given position

Given a singly linked list and a position, delete a linked list node at the given position.

Example:

Input: position = 1, Linked List = 8->2->3->1->7
Output: Linked List = 8->3->1->7

Input: position = 0, Linked List = 8->2->3->1->7
Output: Linked List = 2->3->1->7

We strongly recommend you to minimize your browser and try this yourself first

If node to be deleted is root, simply delete it. To delete a middle node, we must have pointer to the node previous to the node to be deleted. So if position is not zero, we run a loop position-1 times and get pointer to the previous node.

Below is C implementation of above idea.

C/C++

```
// A complete working C program to delete a node in a linked list
// at a given position
#include <stdio.h>
#include <stdlib.h>

// A linked list node
struct node
{
    int data;
    struct node *next;
};

/* Given a reference (pointer to pointer) to the head of a list
and an int, inserts a new node on the front of the list. */
void push(struct node** head_ref, int new_data)
{
    struct node* new_node = (struct node*) malloc(sizeof(struct node));
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

/* Given a reference (pointer to pointer) to the head of a list
and a position, deletes the node at the given position */
void deleteNode(struct node** head_ref, int position)
{
    // If linked list is empty
    if (*head_ref == NULL)
        return;

    // Store head node
    struct node* temp = *head_ref;

    // If head needs to be removed
    if (position == 0)
    {
        *head_ref = temp->next; // Change head
        free(temp);           // free old head
        return;
    }

    // Find previous node of the node to be deleted
    for (int i=0; temp!=NULL && i<position-1; i++)
        temp = temp->next;

    // If position is more than number of nodes
    if (temp == NULL || temp->next == NULL)
        return;

    // Node temp->next is the node to be deleted
    // Store pointer to the next of node to be deleted
    struct node *next = temp->next->next;

    // Unlink the node from linked list
    free(temp->next); // Free memory

    temp->next = next; // Unlink the deleted node from list
}

// This function prints contents of linked list starting from
// the given node
void printList(struct node *node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Driver program to test above functions */
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

    push(&head, 7);
    push(&head, 1);
    push(&head, 3);
    push(&head, 2);
    push(&head, 8);

    puts("Created Linked List: ");
    printList(head);
    deleteNode(&head, 4);
    puts("\nLinked List after Deletion at position 4: ");
    printList(head);
    return 0;
}
```

Java

```
// A complete working Java program to delete a node in a linked list
// at a given position
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;
        Node(int d)
        {
            data = d;
            next = null;
        }
    }

    /* Inserts a new Node at front of the list. */
    public void push(int new_data)
    {
        /* 1 & 2: Allocate the Node &
           Put in the data*/
        Node new_node = new Node(new_data);

        /* 3. Make next of new Node as head */
        new_node.next = head;

        /* 4. Move the head to point to new Node */
        head = new_node;
    }

    /* Given a reference (pointer to pointer) to the head of a list
    and a position, deletes the node at the given position */
    void deleteNode(int position)
    {
        // If linked list is empty
        if (head == null)
            return;

        // Store head node
        Node temp = head;

        // If head needs to be removed
        if (position == 0)
        {
            head = temp.next; // Change head
            return;
        }

        // Find previous node of the node to be deleted
        for (int i=0; temp!=null && i<position-1; i++)
            temp = temp.next;

        // If position is more than number of nodes
        if (temp == null || temp.next == null)
            return;

        // Node temp->next is the node to be deleted
        // Store pointer to the next of node to be deleted
        Node next = temp.next.next;

        temp.next = next; // Unlink the deleted node from list
    }

    /* This function prints contents of linked list starting from
    the given node */
    public void printList()
    {
        Node tnode = head;
        while (tnode != null)
        {
            System.out.print(tnode.data+" ");
            tnode = tnode.next;
        }
    }

    /* Driver program to test above functions. Ideally this function
    should be in a separate user class. It is kept here to keep
    code compact */
    public static void main(String[] args)
    {
        /* Start with the empty list */
        LinkedList llist = new LinkedList();

        llist.push(7);
        llist.push(1);
        llist.push(3);
        llist.push(2);
        llist.push(8);

        System.out.println("\nCreated Linked list is: ");
        llist.printList();

        llist.deleteNode(4); // Delete node at position 4

        System.out.println("\nLinked List after Deletion at position 4: ");
        llist.printList();
    }
}
```

Python

```
# Python program to delete a node in a linked list
# at a given position

# Node class
class Node:
```

```

# Constructor to initialize the node object
def __init__(self, data):
    self.data = data
    self.next = None

class LinkedList:

    # Constructor to initialize head
    def __init__(self):
        self.head = None

    # Function to insert a new node at the beginning
    def push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        self.head = new_node

    # Given a reference to the head of a list
    # and a position, delete the node at a given position
    def deleteNode(self, position):

        # If linked list is empty
        if self.head == None:
            return

        # Store head node
        temp = self.head

        # If head needs to be removed
        if position == 0:
            self.head = temp.next
            temp = None
            return

        # Find previous node of the node to be deleted
        for i in range(position - 1):
            temp = temp.next
            if temp is None:
                break

        # If position is more than number of nodes
        if temp is None:
            return
        if temp.next is None:
            return

        # Node temp.next is the node to be deleted
        # store pointer to the next of node to be deleted
        next = temp.next.next

        # Unlink the node from linked list
        temp.next = None

        temp.next = next

    # Utility function to print the linked LinkedList
    def printList(self):
        temp = self.head
        while(temp):
            print "%d " %(temp.data),
            temp = temp.next

# Driver program to test above function
l1 = LinkedList()
l1.push(7)
l1.push(1)
l1.push(3)
l1.push(2)
l1.push(8)

print "Created Linked List: "
l1.printList()
l1.deleteNode(4)
print "\nLinked List after Deletion at position 4: "
l1.printList()

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

```

Created Linked List:
8 2 3 1 7
Linked List after Deletion at position 4:
8 2 3 1

```

A Programmer's approach of looking at Array vs. Linked List

In general, array is considered a data structure for which size is fixed at the compile time and array memory is allocated either from Data section (e.g. global array) or Stack section (e.g. local array).

Similarly, linked list is considered a data structure for which size is not fixed and memory is allocated from Heap section (e.g. using malloc() etc.) as and when needed. In this sense, array is taken as a static data structure (residing in Data or Stack section) while linked list is taken as a dynamic data structure (residing in Heap section). Memory representation of array and linked list can be visualized as follows:

An array of 4 elements (integer type) which have been initialized with 1, 2, 3 and 4. Suppose, these elements are allocated at memory addresses 0x100, 0x104, 0x08 and 0x10B respectively.

```
[(1)] [(2)] [(3)] [(4)]
0x100 0x104 0x108 0x10B
```

A linked list with 4 nodes where each node has integer as data and these data are initialized with 1, 2, 3 and 4. Suppose, these nodes are allocated via malloc() and memory allocated for them is 0x200, 0x308, 0x404 and 0x20B respectively.

```
[(1), 0x308] [(2), 0x404] [(3), 0x20B] [(4), NULL]
0x200      0x308      0x404      0x20B
```

Anyone with even little understanding of array and linked-list might not be interested in the above explanation. I mean, it is well known that the array elements are allocated memory in sequence i.e. contiguous memory while nodes of a linked list are non-contiguous in memory. Though it sounds trivial yet this is the most important difference between array and linked list. It should be noted that due to this contiguous versus non-contiguous memory, array and linked list are different. In fact, this difference is what makes array vs. linked list! In the following sections, we will try to explore on this very idea further.

Since elements of array are contiguous in memory, we can access any element randomly using index e.g. `intArr[3]` will access directly fourth element of the array. (For newbies, array indexing starts from 0 and that's why fourth element is indexed with 3). Also, due to contiguous memory for successive elements in array, no extra information is needed to be stored in individual elements i.e. no overhead of metadata in arrays. Contrary to this, linked list nodes are non-contiguous in memory. It means that we need some mechanism to traverse or access linked list nodes. To achieve this, each node stores the location of next node and this forms the basis of the link from one node to next node. Therefore, it's called Linked list. Though storing the location of next node is overhead in linked list but it's required. Typically, we see linked list node declaration as follows:

```
struct llnode
{
    int dataInt;

    /* nextNode is the pointer to next node in linked list */
    struct llnode * nextNode;
};
```

So array elements are contiguous in memory and therefore not requiring any metadata. And linked list nodes are non-contiguous in memory thereby requiring metadata in the form of location of next node. Apart from this difference, we can see that array could have several unused elements because memory has already been allocated. But linked list will have only the required no. of data items. All the above information about array and linked list has been mentioned in several textbooks though in different ways.

What if we need to allocate array memory from Heap section (i.e. at run time) and linked list memory from Data/Stack section. First of all, is it possible? Before that, one might ask why would someone need to do this? Now, I hope that the remaining article would make you rethink about the idea of array vs. linked-list [\[1\]](#)

Now consider the case when we need to store certain data in array (because array has the property of random access due to contiguous memory) but we don't know the total size a priori. One possibility is to allocate memory of this array from Heap at run time. For example, as follows:

/*At run-time, suppose we know the required size for integer array (e.g. input size from user). Say, the array size is stored in variable `arrSize`. Allocate this array from Heap as follows*/

```
int * dynArr = (int *)malloc(sizeof(int)*arrSize);
```

Though the memory of this array is allocated from Heap, the elements can still be accessed via index mechanism e.g. `dynArr[i]`. Basically, based on the programming problem, we have combined one benefit of array (i.e. random access of elements) and one benefit of linked list (i.e. delaying the memory allocation till run time and allocating memory from Heap). Another advantage of having this type of dynamic array is that, this method of allocating array from Heap at run time could reduce code-size (of course, it depends on certain other factors e.g. program format etc.)

Now consider the case when we need to store data in a linked list (because no. of nodes in linked list would be equal to actual data items stored i.e. no extra space like array) but we aren't allowed to get this memory from Heap again and again for each node. This might look hypothetical situation to some folks but it's not very uncommon requirement in embedded systems. Basically, in several embedded programs, allocating memory via malloc() etc. isn't allowed due to multiple reasons. One obvious reason is performance i.e. allocating memory via malloc() is costly in terms of time complexity because your embedded program is required to be deterministic most of the times. Another reason could be module specific memory management i.e. it's possible that each module in embedded system manages its own memory. In short, if we need to perform our own memory management, instead of relying on system provided APIs of malloc() and free(), we might choose the linked list which is simulated using array. I hope that you got some idea why we might need to simulate linked list using array. Now, let us first see how this can be done. Suppose, type of a node in linked list (i.e. underlying array) is declared as follows:

```
struct sllNode
{
    int dataInt;

    /*Here, note that nextIndex stores the location of next node in
    linked list*/
    int nextIndex;
};

struct sllNode arrayLL[5];
```

If we initialize this linked list (which is actually an array), it would look as follows in memory:

```
[(0),-1] [(0),-1] [(0),-1] [(0),-1] [(0),-1]
0x500   0x508   0x510   0x518   0x520
```

The important thing to notice is that all the nodes of the linked list are contiguous in memory (each one occupying 8 bytes) and `nextIndex` of each node is set to -1. This (i.e. -1) is done to denote that the each node of the linked list is empty as of now. This linked list is denoted by head index 0.

Now, if this linked list is updated with four elements of data part 4, 3, 2 and 1 successively, it would look as follows in memory. This linked list can be viewed as 0x500 -> 0x508 -> 0x510 -> 0x518.

```
[(1),1] [(2),2] [(3),3] [(4),-2] [(0),-1]
0x500   0x508   0x510   0x518   0x520
```

The important thing to notice is `nextIndex` of last node (i.e. fourth node) is set to -2. This (i.e. -2) is done to denote the end of linked list. Also, head node of the linked list is index 0. This concept of simulating linked list using array would look more interesting if we delete say second node from the above linked list. In that case, the linked list will look as follows in memory:

```
[(1),2] [(0),-1] [(3),3] [(4),-2] [(0),-1]
0x500   0x508   0x510   0x518   0x520
```

The resultant linked list is 0x500 -> 0x510 -> 0x518. Here, it should be noted that even though we have deleted second node from our linked list, the memory for this node is still there because underlying array is still there. But the `nextIndex` of first node now points to third node (for which index is 2).

Hopefully, the above examples would have given some idea that for the simulated linked list, we need to write our own API similar to malloc() and free() which would basically be used to insert and delete a node. Now this is what's called own memory management. Let us see how this can be done in algorithmic manner.

There are multiple ways to do so. If we take the simplistic approach of creating linked list using array, we can use the following logic. For inserting a node, traverse the underlying array and find a node whose `nextIndex` is -1. It means that this node is empty. Use this node as a new node. Update the data part in this new node and set the `nextIndex` of this node to current head node (i.e. head index) of the linked list. Finally, make the index of this

new node as head index of the linked list. To visualize it, let us take an example. Suppose the linked list is as follows where head Index is 0 i.e. linked list is 0x500 -> 0x508 -> 0x518 -> 0x520

| | | | | |
|---------|---------|----------|---------|----------|
| [(1),1] | [(2),3] | [(0),-1] | [(4),4] | [(5),-2] |
| 0x500 | 0x508 | 0x510 | 0x518 | 0x520 |

After inserting a new node with data 8, the linked list would look as follows with head index as 2.

| | | | | |
|---------|---------|---------|---------|----------|
| [(1),1] | [(2),3] | [(8),0] | [(4),4] | [(5),-2] |
| 0x500 | 0x508 | 0x510 | 0x518 | 0x520 |

So the linked list nodes would be at addresses 0x510 -> 0x500 -> 0x508 -> 0x518 -> 0x520

For deleting a node, we need to set the nextIndex of the node as -1 so that the node is marked as empty node. But, before doing so, we need to make sure that the nextIndex of the previous node is updated correctly to index of next node of this node to be deleted. We can see that we have done own memory management for creating a linked list out of the array memory. But, this is one way of inserting and deleting nodes in this linked list. It can be easily noticed that finding an empty node is not so efficient in terms of time complexity. Basically, we're searching the complete array linearly to find an empty node.

Let us see if we can optimize it further. Basically we can maintain a linked list of empty nodes as well in the same array. In that case, the linked list would be denoted by two indexes – one index would be for linked list which has the actual data values i.e. nodes which have been inserted so far and other index would for linked list of empty nodes. By doing so, whenever, we need to insert a new node in existing linked list, we can quickly find an empty node. Let us take an example:

| | | | | | |
|---------|---------|---------|----------|---------|----------|
| [(4),2] | [(0),3] | [(5),5] | [(0),-1] | [(0),1] | [(9),-1] |
| 0x500 | 0x508 | 0x510 | 0x518 | 0x520 | 0x528 |

The above linked list which is represented using two indexes (0 and 5) has two linked lists: one for actual values and another for empty nodes. The linked list with actual values has nodes at address 0x500 -> 0x510 -> 0x528 while the linked list with empty nodes has nodes at addresses 0x520 -> 0x508 -> 0x518. It can be seen that finding an empty node (i.e. writing own API similar to malloc()) should be relatively faster now because we can quickly find a free node. In real world embedded programs, a fixed chunk of memory (normally called memory pool) is allocated using malloc() only once by a module. And then the management of this memory pool (which is basically an array) is done by that module itself using techniques mentioned earlier. Sometimes, there are multiple memory pools each one having different size of node. Of course, there are several other aspects of own memory management but we'll leave it here itself. But it's worth mentioning that there are several methods by which the insertion (which requires our own memory allocation) and deletion (which requires our own memory freeing) can be improved further.

If we look carefully, it can be noticed that the Heap section of memory is basically a big array of bytes which is being managed by the underlying operating system (OS). And OS is providing this memory management service to programmers via malloc(), free() etc. Aha !!

The important take-aways from this article can be summed as follows:

- A) Array means contiguous memory. It can exist in any memory section be it Data or Stack or Heap.
- B) Linked List means non-contiguous linked memory. It can exist in any memory section be it Heap or Data or Stack.
- C) As a programmer, looking at a data structure from memory perspective could provide us better insight in choosing a particular data structure or even designing a new data structure. For example, we might create an array of linked lists etc.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: [Linked List](#)

Find Length of a Linked List (Iterative and Recursive)

Write a C function to count number of nodes in a given singly linked list.

linkedlist_find_length



For example, the function should return 5 for linked list 1->3->1->2->1.

Iterative Solution

- 1) Initialize count as 0
- 2) Initialize a node pointer, current = head.
- 3) Do following while current is not NULL
 - a) current = current -> next
 - b) count++;
- 4) Return count

Following are C/C++, Java and Python implementations of above algorithm to find count of nodes.

C/C++

```
// Iterative C program to find length or count of nodes in a linked list
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Given a reference (pointer to pointer) to the head
of a list and an int, push a new node on the front
of the list. */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);
```



```

/* move the head to point to the new node */
(*head_ref) = new_node;
}

/* Counts no. of nodes in linked list */
int getCount(struct node* head)
{
    int count = 0; // Initialize count
    struct node* current = head; // Initialize current
    while (current != NULL)
    {
        count++;
        current = current->next;
    }
    return count;
}

/* Driver program to test count function */
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

    /* Use push() to construct below list
    1->2->1->3->1 */
    push(&head, 1);
    push(&head, 3);
    push(&head, 1);
    push(&head, 2);
    push(&head, 1);

    /* Check the count function */
    printf("count of nodes is %d", getCount(head));
    return 0;
}

```

Java

```

// Java program to count number of nodes in a linked list

/* Linked list Node*/
class Node
{
    int data;
    Node next;
    Node(int d) { data = d; next = null;}
}

// Linked List class
class LinkedList
{
    Node head; // head of list

    /* Inserts a new Node at front of the list. */
    public void push(int new_data)
    {
        /* 1 & 2: Allocate the Node &
        Put in the data*/
        Node new_node = new Node(new_data);

        /* 3. Make next of new Node as head */
        new_node.next = head;

        /* 4. Move the head to point to new Node */
        head = new_node;
    }

    /* Returns count of nodes in linked list */
    public int getCount()
    {
        Node temp = head;
        int count = 0;
        while (temp != null)
        {
            count++;
            temp = temp.next;
        }
        return count;
    }

    /* Driver program to test above functions. Ideally
    this function should be in a separate user class.
    It is kept here to keep code compact */
    public static void main(String[] args)
    {
        /* Start with the empty list */
        LinkedList llist = new LinkedList();
        llist.push(1);
        llist.push(3);
        llist.push(1);
        llist.push(2);
        llist.push(1);

        System.out.println("Count of nodes is " +
            llist.getCount());
    }
}

```

Python

```

# A complete working Python program to find length of a
# Linked List iteratively

# Node class
class Node:
    # Function to initialise the node object
    def __init__(self, data):
        self.data = data # Assign data
        self.next = None # Initialize next as null

```

```
# Linked List class contains a Node object
class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    # This function is in LinkedList class. It inserts
    # a new node at the beginning of Linked List.
    def push(self, new_data):

        # 1 & 2: Allocate the Node &
        # Put in the data
        new_node = Node(new_data)

        # 3. Make next of new Node as head
        new_node.next = self.head

        # 4. Move the head to point to new Node
        self.head = new_node

    # This function counts number of nodes in Linked List
    # iteratively, given 'node' as starting node.
    def getCount(self):
        temp = self.head # Initialise temp
        count = 0 # Initialise count

        # Loop while end of linked list is not reached
        while (temp):
            count += 1
            temp = temp.next
        return count

# Code execution starts here
if __name__ == '__main__':
    llist = LinkedList()
    llist.push(1)
    llist.push(3)
    llist.push(1)
    llist.push(2)
    llist.push(1)
    print ("Count of nodes is :",llist.getCount())
```

Output:

```
count of nodes is 5
```

Recursive Solution

```
int getCount(head)
1) If head is NULL, return 0.
2) Else return 1 + getCount(head->next)
```

Following are C/C++, Java and Python implementations of above algorithm to find count of nodes.

C/C++

```
// Recursive C program to find length or count of nodes in a linked list
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Given a reference (pointer to pointer) to the head
of a list and an int, push a new node on the front
of the list. */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Counts the no. of occurrences of a node
(search_for) in a linked list (head)*/
int getCount(struct node* head)
{
    // Base case
    if (head == NULL)
        return 0;

    // count is 1 + count of remaining list
    return 1 + getCount(head->next);
}

/* Driver program to test count function */
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

    /* Use push() to construct below list
    1->2->1->3->1 */
    push(&head, 1);
```

```

push(&head, 3);
push(&head, 1);
push(&head, 2);
push(&head, 1);

/* Check the count function */
printf("count of nodes is %d", getCount(head));
return 0;
}

```

Java

```

// Recursive Java program to count number of nodes in
// a linked list

/* Linked list Node*/
class Node
{
    int data;
    Node next;
    Node(int d) { data = d; next = null;}
}

// Linked List class
class LinkedList
{
    Node head; // head of list

    /* Inserts a new Node at front of the list. */
    public void push(int new_data)
    {
        /* 1 & 2: Allocate the Node &
        Put in the data*/
        Node new_node = new Node(new_data);

        /* 3. Make next of new Node as head */
        new_node.next = head;

        /* 4. Move the head to point to new Node */
        head = new_node;
    }

    /* Returns count of nodes in linked list */
    public int getCountRec(Node node)
    {
        // Base case
        if (node == null)
            return 0;

        // Count is this node plus rest of the list
        return 1 + getCountRec(node.next);
    }

    /* Wrapper over getCountRec() */
    public int getCount()
    {
        return getCountRec(head);
    }

    /* Driver program to test above functions. Ideally
    this function should be in a separate user class.
    It is kept here to keep code compact */
    public static void main(String[] args)
    {
        /* Start with the empty list */
        LinkedList llist = new LinkedList();
        llist.push(1);
        llist.push(3);
        llist.push(1);
        llist.push(2);
        llist.push(1);

        System.out.println("Count of nodes is " +
            llist.getCount());
    }
}

```

Python

```

# A complete working Python program to find length of a
# Linked List recursively

# Node class
class Node:
    # Function to initialise the node object
    def __init__(self, data):
        self.data = data # Assign data
        self.next = None # Initialize next as null

# Linked List class contains a Node object
class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    # This function is in LinkedList class. It inserts
    # a new node at the beginning of Linked List.
    def push(self, new_data):

        # 1 & 2: Allocate the Node &
        # Put in the data
        new_node = Node(new_data)

        # 3. Make next of new Node as head
        new_node.next = self.head

        # 4. Move the head to point to new Node
        self.head = new_node

```

```

# This function counts number of nodes in Linked List
# recursively, given 'node' as starting node.
def getCountRec(self, node):
    if (not node): # Base case
        return 0
    else:
        return 1 + self.getCountRec(node.next)

# A wrapper over getCountRec()
def getCount(self):
    return self.getCountRec(self.head)

# Code execution starts here
if __name__ == '__main__':
    llist = LinkedList()
    llist.push(1)
    llist.push(3)
    llist.push(1)
    llist.push(2)
    llist.push(1)
    print 'Count of nodes is :', llist.getCount()

```

Output:

```
count of nodes is 5
```

This article is contributed by **Ravi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: [Linked List](#)

How to write C functions that modify head pointer of a Linked List?

Consider simple representation (without any dummy node) of Linked List. Functions that operate on such Linked lists can be divided in two categories:

1) Functions that do not modify the head pointer: Examples of such functions include, printing a linked list, updating data members of nodes like adding given a value to all nodes, or some other operation which access/update data of nodes

It is generally easy to decide prototype of functions of this category. We can always pass head pointer as an argument and traverse/update the list. For example, the following function that adds x to data members of all nodes.

```

void addXtoList(struct node *node, int x)
{
    while(node != NULL)
    {
        node->data = node->data + x;
        node = node->next;
    }
}

```

2) Functions that modify the head pointer: Examples include, inserting a node at the beginning (head pointer is always modified in this function), inserting a node at the end (head pointer is modified only when the first node is being inserted), deleting a given node (head pointer is modified when the deleted node is first node). There may be different ways to update the head pointer in these functions. Let us discuss these ways using following simple problem:

"Given a linked list, write a function deleteFirst() that deletes the first node of a given linked list. For example, if the list is 1->2->3->4, then it should be modified to 2->3->4"

Algorithm to solve the problem is a simple 3 step process: (a) Store the head pointer (b) change the head pointer to point to next node (c) delete the previous head node.

Following are different ways to update head pointer in deleteFirst() so that the list is updated everywhere.

2.1) Make head pointer global: We can make the head pointer global so that it can be accessed and updated in our function. Following is C code that uses global head pointer.

```

// global head pointer
struct node *head = NULL;

// function to delete first node: uses approach 2.1
// See http://ideone.com/CtIQB for complete program and output
void deleteFirst()
{
    if(head != NULL)
    {
        // store the old value of head pointer
        struct node *temp = head;

        // Change head pointer to point to next node
        head = head->next;

        // delete memory allocated for the previous head node
        free(temp);
    }
}

```

See [this](#) for complete running program that uses above function.

This is not a recommended way as it has many problems like following:

- head is globally accessible, so it can be modified anywhere in your project and may lead to unpredictable results.
- If there are multiple linked lists, then multiple global head pointers with different names are needed.

See [this](#) to know all reasons why should we avoid global variables in our projects.

2.2) Return head pointer: We can write deleteFirst() in such a way that it returns the modified head pointer. Whoever is using this function, have to use the returned value to update the head node.

```
// function to delete first node: uses approach 2.2
// See http://ideone.com/P5oLe for complete program and output
struct node *deleteFirst(struct node *head)
{
    if(head != NULL)
    {
        // store the old value of head pointer
        struct node *temp = head;

        // Change head pointer to point to next node
        head = head->next;

        // delete memory allocated for the previous head node
        free(temp);
    }

    return head;
}
```

See [this](#) for complete program and output.

This approach is much better than the previous 1. There is only one issue with this, if user misses to assign the returned value to head, then things become messy. C/C++ compilers allows to call a function without assigning the returned value.

```
head = deleteFirst(head); // proper use of deleteFirst()
deleteFirst(head); // improper use of deleteFirst(), allowed by compiler
```

2.3) Use Double Pointer: This approach follows the simple C rule: *if you want to modify local variable of one function inside another function, pass pointer to that variable.* So we can pass pointer to the head pointer to modify the head pointer in our deleteFirst() function.

```
// function to delete first node: uses approach 2.3
// See http://ideone.com/9GwTb for complete program and output
void deleteFirst(struct node **head_ref)
{
    if(*head_ref != NULL)
    {
        // store the old value of pointer to head pointer
        struct node *temp = *head_ref;

        // Change head pointer to point to next node
        *head_ref = (*head_ref)->next;

        // delete memory allocated for the previous head node
        free(temp);
    }
}
```

See [this](#) for complete program and output.

This approach seems to be the best among all three as there are less chances of having problems.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

[Linked Lists](#)

Swap nodes in a linked list without swapping data

Given a linked list and two keys in it, swap nodes for two given keys. Nodes should be swapped by changing links. Swapping data of nodes may be expensive in many situations when data contains many fields.

It may be assumed that all keys in linked list are distinct.

Examples:

```
Input: 10->15->12->13->20->14, x = 12, y = 20
Output: 10->15->20->13->12->14

Input: 10->15->12->13->20->14, x = 10, y = 20
Output: 20->15->12->13->10->14

Input: 10->15->12->13->20->14, x = 12, y = 13
Output: 10->15->13->12->20->14
```

This may look a simple problem, but is interesting question as it has following cases to be handled.

- x and y may or may not be adjacent.
- Either x or y may be a head node.
- Either x or y may be last node.
- x and/or y may not be present in linked list.

How to write a clean working code that handles all of the above possibilities.

We strongly recommend to minimize your browser and try this yourself first.

The idea is to first search x and y in given linked list. If any of them is not present, then return. While searching for x and y, keep track of current and previous pointers. First change next of previous pointers, then change next of current pointers. Following are C and Java implementations of this approach.

C

```
/* This program swaps the nodes of linked list rather
than swapping the field from the nodes.*/

#include<stdio.h>
#include<stdlib.h>

/* A linked list node */
struct node
{
```

```

int data;
struct node *next;
};

/* Function to swap nodes x and y in linked list by
changing links */
void swapNodes(struct node **head_ref, int x, int y)
{
    // Nothing to do if x and y are same
    if (x == y) return;

    // Search for x (keep track of prevX and CurrX
    struct node *prevX = NULL, *currX = *head_ref;
    while (currX && currX->data != x)
    {
        prevX = currX;
        currX = currX->next;
    }

    // Search for y (keep track of prevY and CurrY
    struct node *prevY = NULL, *currY = *head_ref;
    while (currY && currY->data != y)
    {
        prevY = currY;
        currY = currY->next;
    }

    // If either x or y is not present, nothing to do
    if (currX == NULL || currY == NULL)
        return;

    // If x is not head of linked list
    if (prevX != NULL)
        prevX->next = currY;
    else // Else make y as new head
        *head_ref = currY;

    // If y is not head of linked list
    if (prevY != NULL)
        prevY->next = currX;
    else // Else make x as new head
        *head_ref = currX;

    // Swap next pointers
    struct node *temp = currY->next;
    currY->next = currX->next;
    currX->next = temp;
}

/* Function to add a node at the beginning of List */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
    while(node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Driver program to test above function */
int main()
{
    struct node *start = NULL;

    /* The constructed linked list is:
    1->2->3->4->5->6->7 */
    push(&start, 7);
    push(&start, 6);
    push(&start, 5);
    push(&start, 4);
    push(&start, 3);
    push(&start, 2);
    push(&start, 1);

    printf("\n Linked list before calling swapNodes() ");
    printList(start);

    swapNodes(&start, 4, 3);

    printf("\n Linked list after calling swapNodes() ");
    printList(start);

    return 0;
}

```

Java

```

// Java program to swap two given nodes of a linked list

class Node
{
    int data;
    Node next;
    Node(int d)
    {
        data = d;
    }
}

```

```

        next = null;
    }
}

class LinkedList
{
    Node head; // head of list

    /* Function to swap Nodes x and y in linked list by
    changing links */
    public void swapNodes(int x, int y)
    {
        // Nothing to do if x and y are same
        if (x == y) return;

        // Search for x (keep track of prevX and currX)
        Node prevX = null, currX = head;
        while (currX != null && currX.data != x)
        {
            prevX = currX;
            currX = currX.next;
        }

        // Search for y (keep track of prevY and currY)
        Node prevY = null, currY = head;
        while (currY != null && currY.data != y)
        {
            prevY = currY;
            currY = currY.next;
        }

        // If either x or y is not present, nothing to do
        if (currX == null || currY == null)
            return;

        // If x is not head of linked list
        if (prevX != null)
            prevX.next = currY;
        else //make y the new head
            head = currY;

        // If y is not head of linked list
        if (prevY != null)
            prevY.next = currX;
        else // make x the new head
            head = currX;

        // Swap next pointers
        Node temp = currX.next;
        currX.next = currY.next;
        currY.next = temp;
    }

    /* Function to add Node at beginning of list. */
    public void push(int new_data)
    {
        /* 1. alloc the Node and put the data */
        Node new_Node = new Node(new_data);

        /* 2. Make next of new Node as head */
        new_Node.next = head;

        /* 3. Move the head to point to new Node */
        head = new_Node;
    }

    /* This function prints contents of linked list starting
    from the given Node */
    public void printList()
    {
        Node tNode = head;
        while (tNode != null)
        {
            System.out.print(tNode.data+" ");
            tNode = tNode.next;
        }
    }

    /* Druver program to test above function */
    public static void main(String[] args)
    {
        LinkedList llist = new LinkedList();

        /* The constructed linked list is:
        1->2->3->4->5->6->7 */
        llist.push(7);
        llist.push(6);
        llist.push(5);
        llist.push(4);
        llist.push(3);
        llist.push(2);
        llist.push(1);

        System.out.print("\n Linked list before calling swapNodes() ");
        llist.printList();

        llist.swapNodes(4, 3);

        System.out.print("\n Linked list after calling swapNodes() ");
        llist.printList();
    }
}
// This code is contributed by Rajat Mishra

```

Python

```

# Python program to swap two given nodes of a linked list
class LinkedList(object):
    def __init__(self):
        self.head = None

# head of list

```

```

class Node(object):
    def __init__(self, d):
        self.data = d
        self.next = None

# Function to swap Nodes x and y in linked list by
# changing links
def swapNodes(self, x, y):

    # Nothing to do if x and y are same
    if x == y:
        return

    # Search for x (keep track of prevX and currX)
    prevX = None
    currX = self.head
    while currX != None and currX.data != x:
        prevX = currX
        currX = currX.next

    # Search for y (keep track of prevY and currY)
    prevY = None
    currY = self.head
    while currY != None and currY.data != y:
        prevY = currY
        currY = currY.next

    # If either x or y is not present, nothing to do
    if currX == None or currY == None:
        return

    # If x is not head of linked list
    if prevX != None:
        prevX.next = currY
    else: # make y the new head
        self.head = currY

    # If y is not head of linked list
    if prevY != None:
        prevY.next = currX
    else: # make x the new head
        self.head = currX

    # Swap next pointers
    temp = currX.next
    currX.next = currY.next
    currY.next = temp

# Function to add Node at beginning of list.
def push(self, new_data):

    # 1. alloc the Node and put the data
    new_Node = self.Node(new_data)

    # 2. Make next of new Node as head
    new_Node.next = self.head

    # 3. Move the head to point to new Node
    self.head = new_Node

# This function prints contents of linked list starting
# from the given Node
def printList(self):
    tNode = self.head
    while tNode != None:
        print tNode.data,
        tNode = tNode.next

# Driver program to test above function
l1 = Linkedlist()

# The constructed linked list is:
# 1->2->3->4->5->6->7
l1.push(7)
l1.push(6)
l1.push(5)
l1.push(4)
l1.push(3)
l1.push(2)
l1.push(1)
print "Linked list before calling swapNodes() "
l1.printList()
l1.swapNodes(4, 3)
print "\nLinked list after calling swapNodes() "
l1.printList()

# This code is contributed by BHAVYA JAIN

```

Output:

```

Linked list before calling swapNodes() 1 2 3 4 5 6 7
Linked list after calling swapNodes() 1 2 4 3 5 6 7

```

Optimizations: The above code can be optimized to search x and y in single traversal. Two loops are used to keep program simple.

This article is contributed by **Gautam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Linked Lists

Write a function to reverse a linked list

Given pointer to the head node of a linked list, the task is to reverse the linked list.

Examples:

```

Input : Head of following linked list
1->2->3->4->NULL
Output : Linked list should be changed to,
4->3->2->1->NULL

```


Input : Head of following linked list
1->2->3->4->5->NULL
Output : Linked list should be changed to,
5->4->3->2->1->NULL

Input : NULL
Output : NULL

Input : 1->NULL
Output : 1->NULL

Iterative Method

Iterate through the linked list. In loop, change next to prev, prev to current and current to next.

Implementation of Iterative Method

C

```
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Function to reverse the linked list */
static void reverse(struct node** head_ref)
{
    struct node* prev = NULL;
    struct node* current = *head_ref;
    struct node* next;
    while (current != NULL)
    {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head_ref = prev;
}

/* Function to push a node */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Function to print linked list */
void printList(struct node *head)
{
    struct node *temp = head;
    while(temp != NULL)
    {
        printf("%d ", temp->data);
        temp = temp->next;
    }
}

/* Driver program to test above function */
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

    push(&head, 20);
    push(&head, 4);
    push(&head, 15);
    push(&head, 85);

    printf("Given linked list\n");
    printList(head);
    reverse(&head);
    printf("\nReversed Linked list\n");
    printList(head);
    getch();
}
```

Java

```
// Java program for reversing the linked list

class LinkedList {

    static Node head;

    static class Node {

        int data;
        Node next;

        Node(int d) {
            data = d;
            next = null;
        }
    }

    /* Function to reverse the linked list */
```

```

Node reverse(Node node) {
    Node prev = null;
    Node current = node;
    Node next = null;
    while (current != null) {
        next = current.next;
        current.next = prev;
        prev = current;
        current = next;
    }
    node = prev;
    return node;
}

// prints content of double linked list
void printList(Node node) {
    while (node != null) {
        System.out.print(node.data + " ");
        node = node.next;
    }
}

public static void main(String[] args) {
    LinkedList list = new LinkedList();
    list.head = new Node(85);
    list.head.next = new Node(15);
    list.head.next.next = new Node(4);
    list.head.next.next.next = new Node(20);

    System.out.println("Given Linked list");
    list.printList(head);
    head = list.reverse(head);
    System.out.println("");
    System.out.println("Reversed linked list ");
    list.printList(head);
}
}

// This code has been contributed by Mayank Jaiswal

```

Python

```

# Python program to reverse a linked list
# Time Complexity : O(n)
# Space Complexity : O(1)

# Node class
class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    # Function to reverse the linked list
    def reverse(self):
        prev = None
        current = self.head
        while(current is not None):
            next = current.next
            current.next = prev
            prev = current
            current = next
        self.head = prev

    # Function to insert a new node at the beginning
    def push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        self.head = new_node

    # Utility function to print the linked LinkedList
    def printList(self):
        temp = self.head
        while(temp):
            print temp.data,
            temp = temp.next

# Driver program to test above functions
l1 = LinkedList()
l1.push(20)
l1.push(4)
l1.push(15)
l1.push(85)

print "Given Linked List"
l1.printList()
l1.reverse()
print "\nReversed Linked List"
l1.printList()

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

```

Given linked list
85 15 4 20
Reversed Linked list
20 4 15 85

```

Time Complexity: O(n)

Space Complexity: O(1)

Recursive Method:

- 1) Divide the list in two parts - first node and rest of the linked list.
- 2) Call reverse for the rest of the linked list.
- 3) Link rest to first.
- 4) Fix head pointer

Linked List Rverse



```
void recursiveReverse(struct node** head_ref)
{
    struct node* first;
    struct node* rest;

    /* empty list */
    if (*head_ref == NULL)
        return;

    /* suppose first = {1, 2, 3}, rest = {2, 3} */
    first = *head_ref;
    rest = first->next;

    /* List has only one node */
    if (rest == NULL)
        return;

    /* reverse the rest list and put the first element at the end */
    recursiveReverse(&rest);
    first->next->next = first;

    /* tricky step -- see the diagram */
    first->next = NULL;

    /* fix the head pointer */
    *head_ref = rest;
}
```

Time Complexity: $O(n)$

Space Complexity: $O(1)$

A Simpler and Tail Recursive Method

Below is C++ implementation of this method.

C++

```
// A simple and tail recursive C++ program to reverse
// a linked list
#include <bits/stdc++.h>
using namespace std;

struct node
{
    int data;
    struct node *next;
};

void reverseUtil(node *curr, node *prev, node **head);

// This function mainly calls reverseUtil()
// with prev as NULL
void reverse(node **head)
{
    if (!head)
        return;
    reverseUtil(*head, NULL, head);
}

// A simple and tail recursive function to reverse
// a linked list. prev is passed as NULL initially.
void reverseUtil(node *curr, node *prev, node **head)
{
    /* If last node mark it head */
    if (!curr->next)
    {
        *head = curr;

        /* Update next to prev node */
        curr->next = prev;
        return;
    }

    /* Save curr->next node for recursive call */
    node *next = curr->next;

    /* and update next */
    curr->next = prev;

    reverseUtil(next, curr, head);
}
```

```

// A utility function to create a new node
node *newNode(int key)
{
    node *temp = new node;
    temp->data = key;
    temp->next = NULL;
    return temp;
}

// A utility function to print a linked list
void printlist(node *head)
{
    while(head != NULL)
    {
        cout << head->data << " ";
        head = head->next;
    }
    cout << endl;
}

// Driver program to test above functions
int main()
{
    node *head1 = newNode(1);
    head1->next = newNode(2);
    head1->next->next = newNode(3);
    head1->next->next->next = newNode(4);
    head1->next->next->next->next = newNode(5);
    head1->next->next->next->next->next = newNode(6);
    head1->next->next->next->next->next->next = newNode(7);
    head1->next->next->next->next->next->next->next = newNode(8);
    cout << "Given linked list\n";
    printlist(head1);
    reverse(&head1);
    cout << "\nReversed linked list\n";
    printlist(head1);
    return 0;
}

```

Java

```

// Java program for reversing the Linked list

class LinkedList {

    static Node head;

    static class Node {

        int data;
        Node next;

        Node(int d) {
            data = d;
            next = null;
        }
    }

    // A simple and tail recursive function to reverse
    // a linked list. prev is passed as NULL initially.
    Node reverseUtil(Node curr, Node prev) {

        /* If last node mark it head */
        if (curr.next == null) {
            head = curr;

            /* Update next to prev node */
            curr.next = prev;
            return null;
        }

        /* Save curr->next node for recursive call */
        Node next1 = curr.next;

        /* and update next */
        curr.next = prev;

        reverseUtil(next1, curr);
        return head;
    }

    // prints content of double linked list
    void printList(Node node) {
        while (node != null) {
            System.out.print(node.data + " ");
            node = node.next;
        }
    }

    public static void main(String[] args) {
        LinkedList list = new LinkedList();
        list.head = new Node(1);
        list.head.next = new Node(2);
        list.head.next.next = new Node(3);
        list.head.next.next.next = new Node(4);
        list.head.next.next.next.next = new Node(5);
        list.head.next.next.next.next.next = new Node(6);
        list.head.next.next.next.next.next.next = new Node(7);
        list.head.next.next.next.next.next.next.next = new Node(8);

        System.out.println("Original Linked list ");
        list.printList(head);
        Node res = list.reverseUtil(head, null);
        System.out.println("");
        System.out.println("");
        System.out.println("Reversed linked list ");
        list.printList(res);
    }
}

// This code has been contributed by Mayank Jaiswal

```

Python

```
# Simple and tail recursive Python program to
# reverse a linked list

# Node class
class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    def reverseUtil(self, curr, prev):

        # If last node mark it head
        if curr.next is None :
            self.head = curr

        # Update next to prev node
        curr.next = prev
        return

        # Save curr.next node for recursive call
        next = curr.next

        # And update next
        curr.next = prev

        self.reverseUtil(next, curr)

    # This function mainly calls reverseUtil()
    # with previous as None
    def reverse(self):
        if self.head is None:
            return
        self.reverseUtil(self.head, None)

    # Function to insert a new node at the beginning
    def push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        self.head = new_node

    # Utility function to print the linked LinkedList
    def printList(self):
        temp = self.head
        while(temp):
            print temp.data,
            temp = temp.next

# Driver program
l1list = LinkedList()
l1list.push(8)
l1list.push(7)
l1list.push(6)
l1list.push(5)
l1list.push(4)
l1list.push(3)
l1list.push(2)
l1list.push(1)

print "Given linked list"
l1list.printList()

l1list.reverse()

print "\nReverse linked list"
l1list.printList()

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
Given linked list
1 2 3 4 5 6 7 8

Reversed linked list
8 7 6 5 4 3 2 1
```

Thanks to Gaurav Ahirwar for suggesting this solution.

Iteratively Reverse a linked list using only 2 pointers (An Interesting Method)

References:

<http://cslibrary.stanford.edu/105/LinkedListProblems.pdf>

GATE CS Corner Company Wise Coding Practice

Linked Lists
Reverse
Yatra.com-Question

Merge two sorted linked lists

Write a SortedMerge() function that takes two lists, each of which is sorted in increasing order, and merges the two together into one list which is in increasing order. SortedMerge() should return the new list. The new list

should be made by splicing
together the nodes of the first two lists.

For example if the first linked list a is 5->10->15 and the other linked list b is 2->3->20, then SortedMerge() should return a pointer to the head node of the merged list 2->3->5->10->15->20.

There are many cases to deal with: either 'a' or 'b' may be empty, during processing either 'a' or 'b' may run out first, and finally there's the problem of starting the result list empty, and building it up while going through 'a' and 'b'.

Method 1 (Using Dummy Nodes)

The strategy here uses a temporary dummy node as the start of the result list. The pointer Tail always points to the last node in the result list, so appending new nodes is easy.

The dummy node gives tail something to point to initially when the result list is empty. This dummy node is efficient, since it is only temporary, and it is allocated in the stack. The loop proceeds, removing one node from either 'a' or 'b', and adding it to tail. When

we are done, the result is in dummy.next.

```
/* C/C++ program to merge two sorted linked lists */
#include<stdio.h>
#include<stdlib.h>
#include<assert.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* pull off the front node of the source and put it in dest */
void MoveNode(struct node** destRef, struct node** sourceRef);

/* Takes two lists sorted in increasing order, and splices
their nodes together to make one big sorted list which
is returned. */
struct node* SortedMerge(struct node* a, struct node* b)
{
    /* a dummy first node to hang the result on */
    struct node dummy;

    /* tail points to the last result node */
    struct node* tail = &dummy;

    /* so tail->next is the place to add new nodes
to the result. */
    dummy.next = NULL;
    while (1)
    {
        if (a == NULL)
        {
            /* if either list runs out, use the
other list */
            tail->next = b;
            break;
        }
        else if (b == NULL)
        {
            tail->next = a;
            break;
        }
        if (a->data <= b->data)
            MoveNode(&(tail->next), &a);
        else
            MoveNode(&(tail->next), &b);

        tail = tail->next;
    }
    return(dummy.next);
}

/* UTILITY FUNCTIONS */
/* MoveNode() function takes the node from the front of the
source, and move it to the front of the dest.
It is an error to call this with the source list empty.

Before calling MoveNode():
source == { 1, 2, 3 }
dest == { 1, 2, 3 }

After calling MoveNode():
source == { 2, 3 }
dest == { 1, 1, 2, 3 } */
void MoveNode(struct node** destRef, struct node** sourceRef)
{
    /* the front source node */
    struct node* newNode = *sourceRef;
    assert(newNode != NULL);

    /* Advance the source pointer */
    *sourceRef = newNode->next;

    /* Link the old dest off the new node */
    newNode->next = *destRef;

    /* Move dest to point to the new node */
    *destRef = newNode;
}

/* Function to insert a node at the beginning of the
linked list */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Function to print nodes in a given linked list */
```

```

void printList(struct node *node)
{
    while (node!=NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Drier program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct node* res = NULL;
    struct node* a = NULL;
    struct node* b = NULL;

    /* Let us create two sorted linked lists to test
    the functions
    Created lists, a: 5->10->15, b: 2->3->20 */
    push(&a, 15);
    push(&a, 10);
    push(&a, 5);

    push(&b, 20);
    push(&b, 3);
    push(&b, 2);

    /* Remove duplicates from linked list */
    res = SortedMerge(a, b);

    printf("Merged Linked List is: \n");
    printList(res);

    return 0;
}

```

Output :

```

Merged Linked List is:
2 3 5 10 15 20

```

Method 2 (Using Local References)

This solution is structurally very similar to the above, but it avoids using a dummy node. Instead, it maintains a struct node** pointer, lastPtrRef, that always points to the last pointer of the result list. This solves the same case that the dummy node did — dealing with the result list when it is empty. If you are trying to build up a list at its tail, either the dummy node or the struct node** “reference” strategy can be used (see Section 1 for details).

```

struct node* SortedMerge(struct node* a, struct node* b)
{
    struct node* result = NULL;

    /* point to the last result pointer */
    struct node** lastPtrRef = &result;

    while(1)
    {
        if (a == NULL)
        {
            *lastPtrRef = b;
            break;
        }
        else if (b==NULL)
        {
            *lastPtrRef = a;
            break;
        }
        if(a->data <= b->data)
        {
            MoveNode(lastPtrRef, &a);
        }
        else
        {
            MoveNode(lastPtrRef, &b);
        }

        /* tricky: advance to point to the next ".next" field */
        lastPtrRef = &((*lastPtrRef)->next);
    }
    return(result);
}

```

Method 3 (Using Recursion)

Merge is one of those nice recursive problems where the recursive solution code is much cleaner than the iterative code. You probably wouldn't want to use the recursive version for production code however, because it will use stack space which is proportional to the length of the lists.

```

struct node* SortedMerge(struct node* a, struct node* b)
{
    struct node* result = NULL;

    /* Base cases */
    if (a == NULL)
        return(b);
    else if (b==NULL)
        return(a);

    /* Pick either a or b, and recur */
    if (a->data <= b->data)
    {
        result = a;
        result->next = SortedMerge(a->next, b);
    }
    else
    {
        result = b;
        result->next = SortedMerge(a, b->next);
    }
    return(result);
}

```

Please write comments if you find the above code/algorithm incorrect, or find better ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Linked Lists
Merge Sort

Merge Sort for Linked Lists

Merge sort is often preferred for sorting a linked list. The slow random-access performance of a linked list makes some other algorithms (such as quicksort) perform poorly, and others (such as heapsort) completely impossible.

Let head be the first node of the linked list to be sorted and headRef be the pointer to head. Note that we need a reference to head in MergeSort() as the below implementation changes next links to sort the linked lists (not data at the nodes), so head node has to be changed if the data at original head is not the smallest value in linked list.

```
MergeSort(headRef)
1) If head is NULL or there is only one element in the Linked List
   then return.
2) Else divide the linked list into two halves.
   FrontBackSplit(head, &a, &b); /* a and b are two halves */
3) Sort the two halves a and b.
   MergeSort(a);
   MergeSort(b);
4) Merge the sorted a and b (using SortedMerge() discussed here)
   and update the head pointer using headRef.
   *headRef = SortedMerge(a, b);
```

```
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* function prototypes */
struct node* SortedMerge(struct node* a, struct node* b);
void FrontBackSplit(struct node* source,
    struct node** frontRef, struct node** backRef);

/* sorts the linked list by changing next pointers (not data) */
void MergeSort(struct node** headRef)
{
    struct node* head = *headRef;
    struct node* a;
    struct node* b;

    /* Base case -- length 0 or 1 */
    if ((head == NULL) || (head->next == NULL))
    {
        return;
    }

    /* Split head into 'a' and 'b' sublists */
    FrontBackSplit(head, &a, &b);

    /* Recursively sort the sublists */
    MergeSort(&a);
    MergeSort(&b);

    /* answer = merge the two sorted lists together */
    *headRef = SortedMerge(a, b);
}

/* See http://geeksforgeeks.org/?p=3622 for details of this
function */
struct node* SortedMerge(struct node* a, struct node* b)
{
    struct node* result = NULL;

    /* Base cases */
    if (a == NULL)
        return(b);
    else if (b==NULL)
        return(a);

    /* Pick either a or b, and recur */
    if (a->data <= b->data)
    {
        result = a;
        result->next = SortedMerge(a->next, b);
    }
    else
    {
        result = b;
        result->next = SortedMerge(a, b->next);
    }
    return(result);
}

/* UTILITY FUNCTIONS */
/* Split the nodes of the given list into front and back halves,
and return the two lists using the reference parameters.
If the length is odd, the extra node should go in the front list.
Uses the fast/slow pointer strategy. */
void FrontBackSplit(struct node* source,
    struct node** frontRef, struct node** backRef)
{
    struct node* fast;
    struct node* slow;
    if (source==NULL || source->next==NULL)
    {
        /* length < 2 cases */
        *frontRef = source;
        *backRef = NULL;
    }
    else
    {
        slow = source;
        fast = source->next;
```



```

/* Advance 'fast' two nodes, and advance 'slow' one node */
while (fast != NULL)
{
    fast = fast->next;
    if (fast != NULL)
    {
        slow = slow->next;
        fast = fast->next;
    }
}

/* 'slow' is before the midpoint in the list, so split it in two
at that point. */
*frontRef = source;
*backRef = slow->next;
slow->next = NULL;
}
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
    while(node!=NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Function to insert a node at the beging of the linked list */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Drier program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct node* res = NULL;
    struct node* a = NULL;

    /* Let us create a unsorted linked lists to test the functions
    Created lists shall be a: 2->3->20->5->10->15 */
    push(&a, 15);
    push(&a, 10);
    push(&a, 5);
    push(&a, 20);
    push(&a, 3);
    push(&a, 2);

    /* Sort the above created Linked List */
    MergeSort(&a);

    printf("\n Sorted Linked List is: \n");
    printList(a);

    getchar();
    return 0;
}

```

Time Complexity: $O(n \log n)$

Sources:

http://en.wikipedia.org/wiki/Merge_sort

<http://cslibrary.stanford.edu/105/LinkedListProblems.pdf>

Please write comments if you find the above code/algorithm incorrect, or find better ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

[Linked Lists](#)
[Sorting](#)
[Merge Sort](#)

Reverse a Linked List in groups of given size

Given a linked list, write a function to reverse every k nodes (where k is an input to the function).

Example:

Inputs: 1->2->3->4->5->6->7->8->NULL and k = 3
 Output: 3->2->1->6->5->4->8->7->NULL.

Inputs: 1->2->3->4->5->6->7->8->NULL and k = 5
 Output: 5->4->3->2->1->8->7->6->NULL.

Algorithm: *reverse(head, k)*

- 1) Reverse the first sub-list of size k. While reversing keep track of the next node and previous node. Let the pointer to the next node be *next* and pointer to the previous node be *prev*. See [this post](#) for reversing a linked list.
- 2) *head->next = reverse(next, k)* /* Recursively call for rest of the list and link the two sub-lists */
- 3) return *prev* /* *prev* becomes the new head of the list (see the diagrams of iterative method of [this post](#)) */

C/C++

```

// C program to reverse a linked list in groups of given size
#include<stdio.h>
#include<stdlib.h>

/* Link list node */

```

```

struct node
{
    int data;
    struct node* next;
};

/* Reverses the linked list in groups of size k and returns the
pointer to the new head node. */
struct node *reverse (struct node *head, int k)
{
    struct node* current = head;
    struct node* next = NULL;
    struct node* prev = NULL;
    int count = 0;

    /*reverse first k nodes of the linked list*/
    while (current != NULL && count < k)
    {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
        count++;
    }

    /* next is now a pointer to (k+1)th node
    Recursively call for the list starting from current.
    And make rest of the list as next of first node */
    if (next != NULL)
        head->next = reverse(next, k);

    /* prev is new head of the input list */
    return prev;
}

/* UTILITY FUNCTIONS */
/* Function to push a node */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Function to print linked list */
void printList(struct node *node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Driver program to test above function */
int main(void)
{
    /* Start with the empty list */
    struct node* head = NULL;

    /* Created Linked list is 1->2->3->4->5->6->7->8->9 */
    push(&head, 9);
    push(&head, 8);
    push(&head, 7);
    push(&head, 6);
    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 1);

    printf("\nGiven linked list \n");
    printList(head);
    head = reverse(head, 3);

    printf("\nReversed Linked list \n");
    printList(head);

    return(0);
}

```

Java

```

// Java program to reverse a linked list in groups of
// given size
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;
        Node(int d) {data = d; next = null;}
    }

    Node reverse(Node head, int k)
    {
        Node current = head;
        Node next = null;
        Node prev = null;

        int count = 0;
    }
}

```

```

/* Reverse first k nodes of linked list */
while (count < k && current != null)
{
    next = current.next;
    current.next = prev;
    prev = current;
    current = next;
    count++;
}

/* next is now a pointer to (k+1)th node
   Recursively call for the list starting from current.
   And make rest of the list as next of first node */
if (next != null)
    head.next = reverse(next, k);

// prev is now head of input list
return prev;
}

/* Utility functions */

/* Inserts a new Node at front of the list. */
public void push(int new_data)
{
    /* 1 & 2: Allocate the Node &
       Put in the data*/
    Node new_node = new Node(new_data);

    /* 3. Make next of new Node as head */
    new_node.next = head;

    /* 4. Move the head to point to new Node */
    head = new_node;
}

/* Function to print linked list */
void printList()
{
    Node temp = head;
    while (temp != null)
    {
        System.out.print(temp.data+" ");
        temp = temp.next;
    }
    System.out.println();
}

/* Driver program to test above functions */
public static void main(String args[])
{
    LinkedList llist = new LinkedList();

    /* Constructed Linked List is 1->2->3->4->5->6->
       7->8->9->null */
    llist.push(9);
    llist.push(8);
    llist.push(7);
    llist.push(6);
    llist.push(5);
    llist.push(4);
    llist.push(3);
    llist.push(2);
    llist.push(1);

    System.out.println("Given Linked List");
    llist.printList();

    llist.head = llist.reverse(llist.head, 3);

    System.out.println("Reversed list");
    llist.printList();
}
}
/* This code is contributed by Rajat Mishra */

```

Python

Python program to reverse a linked list in group of given size

Node class
class Node:

```

# Constructor to initialize the node object
def __init__(self, data):
    self.data = data
    self.next = None

```

class LinkedList:

```

# Function to initialize head
def __init__(self):
    self.head = None

```

```

def reverse(self, head, k):
    current = head
    next = None
    prev = None
    count = 0

```

```

# Reverse first k nodes of the linked list
while(current is not None and count < k):
    next = current.next
    current.next = prev
    prev = current
    current = next
    count += 1

```

```

# next is now a pointer to (k+1)th node
# recursively call for the list starting

```

```

# from current . And make rest of the list as
# next of first node
if next is not None:
    head.next = self.reverse(next, k)

# prev is new head of the input list
return prev

# Function to insert a new node at the beginning
def push(self, new_data):
    new_node = Node(new_data)
    new_node.next = self.head
    self.head = new_node

# Utility function to print the linked LinkedList
def printList(self):
    temp = self.head
    while(temp):
        print temp.data,
        temp = temp.next

# Driver program
l1 = LinkedList()
l1.push(9)
l1.push(8)
l1.push(7)
l1.push(6)
l1.push(5)
l1.push(4)
l1.push(3)
l1.push(2)
l1.push(1)

print "Given linked list"
l1.printList()
l1.head = l1.reverse(l1.head, 3)

print "\nReversed Linked list"
l1.printList()

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

```

Given Linked List
1 2 3 4 5 6 7 8 9
Reversed list
3 2 1 6 5 4 9 8 7

```

Time Complexity: $O(n)$ where n is the number of nodes in the given list.

Please write comments if you find the above code/algorithm incorrect, or find other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

[Linked Lists](#)
[Adobe-Question](#)
[Amazon-Question](#)
[Reverse](#)
[Snapdeal-Question](#)
[Yatra.com-Question](#)

Detect and Remove Loop in a Linked List

Write a function *detectAndRemoveLoop()* that checks whether a given Linked List contains loop and if loop is present then removes the loop and returns true. And if the list doesn't contain loop then returns false. Below diagram shows a linked list with a loop. *detectAndRemoveLoop()* must change the below list to 1->2->3->4->5->NULL.



We also recommend to read following post as a prerequisite of the solution discussed here.

[Write a C function to detect loop in a linked list](#)

Before trying to remove the loop, we must detect it. Techniques discussed in the above post can be used to detect loop. To remove loop, all we need to do is to get pointer to the last node of the loop. For example, node with value 5 in the above diagram. Once we have pointer to the last node, we can make the next of this node as NULL and loop is gone.

We can easily use Hashing or Visited node techniques (discussed in the above mentioned post) to get the pointer to the last node. Idea is simple: the very first node whose next is already visited (or hashed) is the last node.

We can also use Floyd Cycle Detection algorithm to detect and remove the loop. In the Floyd's algo, the slow and fast pointers meet at a loop node. We can use this loop node to remove cycle. There are following two different ways of removing loop when Floyd's algorithm is used for Loop detection.

Method 1 (Check one by one)

We know that Floyd's Cycle detection algorithm terminates when fast and slow pointers meet at a common point. We also know that this common point is one of the loop nodes (2 or 3 or 4 or 5 in the above diagram). We store the address of this in a pointer variable say ptr2. Then we start from the head of the Linked List and check for nodes one by one if they are reachable from ptr2. When we find a node that is reachable, we know that this node is the starting node of the loop in Linked List and we can get pointer to the previous of this node.

C

```

#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Function to remove loop. Used by detectAndRemoveLoop() */
void removeLoop(struct node *, struct node *);

/* This function detects and removes loop in the list
If loop was there in the list then it returns 1,

```

```

otherwise returns 0 */
int detectAndRemoveLoop(struct node *list)
{
    struct node *slow_p = list, *fast_p = list;

    while (slow_p && fast_p && fast_p->next)
    {
        slow_p = slow_p->next;
        fast_p = fast_p->next->next;

        /* If slow_p and fast_p meet at some point then there
        is a loop */
        if (slow_p == fast_p)
        {
            removeLoop(slow_p, list);

            /* Return 1 to indicate that loop is found */
            return 1;
        }
    }

    /* Return 0 to indicate that there is no loop */
    return 0;
}

/* Function to remove loop.
loop_node --> Pointer to one of the loop nodes
head --> Pointer to the start node of the linked list */
void removeLoop(struct node *loop_node, struct node *head)
{
    struct node *ptr1;
    struct node *ptr2;

    /* Set a pointer to the beginning of the Linked List and
    move it one by one to find the first node which is
    part of the Linked List */
    ptr1 = head;
    while (1)
    {
        /* Now start a pointer from loop_node and check if it ever
        reaches ptr2 */
        ptr2 = loop_node;
        while (ptr2->next != loop_node && ptr2->next != ptr1)
            ptr2 = ptr2->next;

        /* If ptr2 reached ptr1 then there is a loop. So break the
        loop */
        if (ptr2->next == ptr1)
            break;

        /* If ptr2 didn't reach ptr1 then try the next node after ptr1 */
        ptr1 = ptr1->next;
    }

    /* After the end of loop ptr2 is the last node of the loop. So
    make next of ptr2 as NULL */
    ptr2->next = NULL;
}

/* Function to print linked list */
void printList(struct node *node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

struct node *newNode(int key)
{
    struct node *temp = new struct node;
    temp->data = key;
    temp->next = NULL;
    return temp;
}

/* Driver program to test above function */
int main()
{
    struct node *head = newNode(50);
    head->next = newNode(20);
    head->next->next = newNode(15);
    head->next->next->next = newNode(4);
    head->next->next->next->next = newNode(10);

    /* Create a loop for testing */
    head->next->next->next->next->next = head->next->next;

    detectAndRemoveLoop(head);

    printf("Linked List after removing loop \n");
    printList(head);
    return 0;
}

```

Java

```

// Java program to detect and remove loop in linked list

class LinkedList {

    static Node head;

    static class Node {

        int data;
        Node next;

        Node(int d) {
            data = d;
            next = null;
        }
    }
}

```

```

}

// Function that detects loop in the list
int detectAndRemoveLoop(Node node) {
    Node slow = node, fast = node;
    while (slow != null && fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;

        // If slow and fast meet at same point then loop is present
        if (slow == fast) {
            removeLoop(slow, node);
            return 1;
        }
    }
    return 0;
}

// Function to remove loop
void removeLoop(Node loop, Node curr) {
    Node ptr1 = null, ptr2 = null;

    /* Set a pointer to the beging of the Linked List and
    move it one by one to find the first node which is
    part of the Linked List */
    ptr1 = curr;
    while (ptr1 != null) {

        /* Now start a pointer from loop_node and check if it ever
        reaches ptr2 */
        ptr2 = loop;
        while (ptr2.next != loop && ptr2.next != ptr1) {
            ptr2 = ptr2.next;
        }

        /* If ptr2 reached ptr1 then there is a loop. So break the
        loop */
        if (ptr2.next == ptr1) {
            break;
        }

        /* If ptr2 didn't reach ptr1 then try the next node after ptr1 */
        ptr1 = ptr1.next;
    }

    /* After the end of loop ptr2 is the last node of the loop. So
    make next of ptr2 as NULL */
    ptr2.next = null;
}

// Function to print the linked list
void printList(Node node) {
    while (node != null) {
        System.out.print(node.data + " ");
        node = node.next;
    }
}

// Driver program to test above functions
public static void main(String[] args) {
    LinkedList list = new LinkedList();
    list.head = new Node(50);
    list.head.next = new Node(20);
    list.head.next.next = new Node(15);
    list.head.next.next.next = new Node(4);
    list.head.next.next.next.next = new Node(10);

    // Creating a loop for testing
    head.next.next.next.next.next = head.next.next;
    list.detectAndRemoveLoop(head);
    System.out.println("Linked List after removing loop : ");
    list.printList(head);
}
}

// This code has been contributed by Mayank Jaiswal

```

Python

Python program to detect and remove loop in linked list

Node class
class Node:

```

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

```

class LinkedList:

```

    # Function to initialize head
    def __init__(self):
        self.head = None

```

```

    def detectAndRemoveLoop(self):
        slow_p = fast_p = self.head
        while(slow_p and fast_p and fast_p.next):
            slow_p = slow_p.next
            fast_p = fast_p.next.next

```

```

        # If slow_p and fast_p meet at some poin
        # then there is a loop
        if slow_p == fast_p:
            self.removeLoop(slow_p)

```

```

        # Return 1 to indicate that loop if found
        return 1

```

```

    # Return 0 to indicate that there is no loop
    return 0

```

```

# Function to remove loop
# loop node-> Pointer to one of the loop nodes
# head --> Pointer to the start node of the
# linked list
def removeLoop(self, loop_node):

    # Set a pointer to the beginning of the linked
    # list and move it one by one to find the first
    # node which is part of the linked list
    ptr1 = self.head
    while(1):
        # Now start a pointer from loop_node and check
        # if it ever reaches ptr2
        ptr2 = loop_node
        while(ptr2.next!= loop_node and ptr2.next !=ptr1):
            ptr2 = ptr2.next

        # If ptr2 reached ptr1 then there is a loop.
        # So break the loop
        if ptr2.next == ptr1 :
            break

        ptr1 = ptr1.next

    # After the end of loop ptr2 is the last node of
    # the loop. So make next of ptr2 as NULL
    ptr2.next = None
    # Function to insert a new node at the beginning
def push(self, new_data):
    new_node = Node(new_data)
    new_node.next = self.head
    self.head = new_node

# Utility function to print the linked LinkedList
def printList(self):
    temp = self.head
    while(temp):
        print temp.data,
        temp = temp.next

# Driver program
l1 = LinkedList()
l1.push(10)
l1.push(4)
l1.push(15)
l1.push(20)
l1.push(50)

# Create a loop for testing
l1.head.next.next.next.next = l1.head.next.next

l1.detectAndRemoveLoop()

print "Linked List after removing loop"
l1.printList()

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

```

Linked List after removing loop
50 20 15 4 10

```

Method 2 (Better Solution)

This method is also dependent on Floyd's Cycle detection algorithm.

- 1) Detect Loop using Floyd's Cycle detection algo and get the pointer to a loop node.
- 2) Count the number of nodes in loop. Let the count be k.
- 3) Fix one pointer to the head and another to kth node from head.
- 4) Move both pointers at the same pace, they will meet at loop starting node.
- 5) Get pointer to the last node of loop and make next of it as NULL.

Thanks to WgpShashank for suggesting this method.

C

```

#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Function to remove loop. */
void removeLoop(struct node *, struct node *);

/* This function detects and removes loop in the list
If loop was there in the list then it returns 1,
otherwise returns 0 */
int detectAndRemoveLoop(struct node *list)
{
    struct node *slow_p = list, *fast_p = list;

    while (slow_p && fast_p && fast_p->next)
    {
        slow_p = slow_p->next;
        fast_p = fast_p->next->next;

        /* If slow_p and fast_p meet at some point then there
        is a loop */
        if (slow_p == fast_p)
        {
            removeLoop(slow_p, list);

            /* Return 1 to indicate that loop is found */
            return 1;
        }
    }
}

```

```

/* Return 0 to indicate that there is no loop */
return 0;
}

/* Function to remove loop.
loop_node -> Pointer to one of the loop nodes
head -> Pointer to the start node of the linked list */
void removeLoop(struct node *loop_node, struct node *head)
{
    struct node *ptr1 = loop_node;
    struct node *ptr2 = loop_node;

    // Count the number of nodes in loop
    unsigned int k = 1, i;
    while (ptr1->next != ptr2)
    {
        ptr1 = ptr1->next;
        k++;
    }

    // Fix one pointer to head
    ptr1 = head;

    // And the other pointer to k nodes after head
    ptr2 = head;
    for (i = 0; i < k; i++)
        ptr2 = ptr2->next;

    /* Move both pointers at the same pace,
    they will meet at loop starting node */
    while (ptr2 != ptr1)
    {
        ptr1 = ptr1->next;
        ptr2 = ptr2->next;
    }

    // Get pointer to the last node
    ptr2 = ptr2->next;
    while (ptr2->next != ptr1)
        ptr2 = ptr2->next;

    /* Set the next node of the loop ending node
    to fix the loop */
    ptr2->next = NULL;
}

/* Function to print linked list */
void printList(struct node *node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

struct node *newNode(int key)
{
    struct node *temp = new struct node;
    temp->data = key;
    temp->next = NULL;
    return temp;
}

/* Driver program to test above function */
int main()
{
    struct node *head = newNode(50);
    head->next = newNode(20);
    head->next->next = newNode(15);
    head->next->next->next = newNode(4);
    head->next->next->next->next = newNode(10);

    /* Create a loop for testing */
    head->next->next->next->next->next = head->next->next;

    detectAndRemoveLoop(head);

    printf("Linked List after removing loop\n");
    printList(head);
    return 0;
}

```

Java

```

// Java program to detect and remove loop in linked list

class LinkedList {

    static Node head;

    static class Node {

        int data;
        Node next;

        Node(int d) {
            data = d;
            next = null;
        }
    }

    // Function that detects loop in the list
    int detectAndRemoveLoop(Node node) {
        Node slow = node, fast = node;
        while (slow != null && fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;

            // If slow and fast meet at same point then loop is present
            if (slow == fast) {
                removeLoop(slow, node);
            }
        }
    }
}

```



```

        return 1;
    }
}
return 0;
}

// Function to remove loop
void removeLoop(Node loop, Node head) {
    Node ptr1 = loop;
    Node ptr2 = loop;

    // Count the number of nodes in loop
    int k = 1, i;
    while (ptr1.next != ptr2) {
        ptr1 = ptr1.next;
        k++;
    }

    // Fix one pointer to head
    ptr1 = head;

    // And the other pointer to k nodes after head
    ptr2 = head;
    for (i = 0; i < k; i++) {
        ptr2 = ptr2.next;
    }

    /* Move both pointers at the same pace,
    they will meet at loop starting node */
    while (ptr2 != ptr1) {
        ptr1 = ptr1.next;
        ptr2 = ptr2.next;
    }

    // Get pointer to the last node
    ptr2 = ptr2.next;
    while (ptr2.next != ptr1) {
        ptr2 = ptr2.next;
    }

    /* Set the next node of the loop ending node
    to fix the loop */
    ptr2.next = null;
}

// Function to print the linked list
void printList(Node node) {
    while (node != null) {
        System.out.print(node.data + " ");
        node = node.next;
    }
}

// Driver program to test above functions
public static void main(String[] args) {
    LinkedList list = new LinkedList();
    list.head = new Node(50);
    list.head.next = new Node(20);
    list.head.next.next = new Node(15);
    list.head.next.next.next = new Node(4);
    list.head.next.next.next.next = new Node(10);

    // Creating a loop for testing
    head.next.next.next.next.next = head.next.next;
    list.detectAndRemoveLoop(head);
    System.out.println("Linked List after removing loop : ");
    list.printList(head);
}
}

// This code has been contributed by Mayank Jaiswal

```

Python

```

# Python program to detect and remove loop in linked list

# Node class
class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    def detectAndRemoveLoop(self):
        slow_p = fast_p = self.head

        while(slow_p and fast_p and fast_p.next):
            slow_p = slow_p.next
            fast_p = fast_p.next.next

            # If slow_p and fast_p meet at some point then
            # there is a loop
            if slow_p == fast_p:
                self.removeLoop(slow_p)

            # Return 1 to indicate that loop is found
            return 1

        # Return 0 to indicate that there is no loop
        return 0

    # Function to remove loop
    # loop_node --> pointer to one of the loop nodes
    # head --> Pointer to the start node of the linked list

```

```

def removeLoop(self, loop_node):
    ptr1 = loop_node
    ptr2 = loop_node

    # Count the number of nodes in loop
    k = 1
    while(ptr1.next != ptr2):
        ptr1 = ptr1.next
        k += 1

    # Fix one pointer to head
    ptr1 = self.head

    # And the other pointer to k nodes after head
    ptr2 = self.head
    for i in range(k):
        ptr2 = ptr2.next

    # Move both pointers at the same place
    # they will meet at loop starting node
    while(ptr2 != ptr1):
        ptr1 = ptr1.next
        ptr2 = ptr2.next

    # Get pointer to the last node
    ptr2 = ptr2.next
    while(ptr2.next != ptr1):
        ptr2 = ptr2.next

    # Set the next node of the loop ending node
    # to fix the loop
    ptr2.next = None

    # Function to insert a new node at the beginning
    def push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        self.head = new_node

    # Utility function to print the linked LinkedList
    def printList(self):
        temp = self.head
        while(temp):
            print temp.data,
            temp = temp.next

# Driver program
l1 = LinkedList()
l1.push(10)
l1.push(4)
l1.push(15)
l1.push(20)
l1.push(50)

# Create a loop for testing
l1.head.next.next.next.next.next = l1.head.next.next

l1.detectAndRemoveLoop()

print "Linked List after removing loop"
l1.printList()

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

```

Linked List after removing loop
50 20 15 4 10

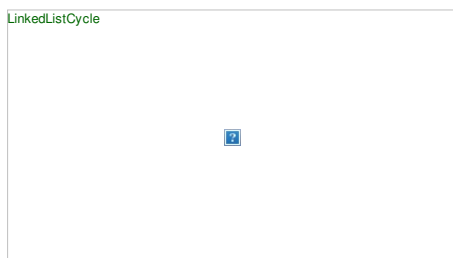
```

Method 3 (Optimized Method 2: Without Counting Nodes in Loop)

We do not need to count number of nodes in Loop. After detecting the loop, if we start slow pointer from head and move both slow and fast pointers at same speed until fast don't meet, they would meet at the beginning of linked list.

How does this work?

Let slow and fast meet at some point after Floyd's Cycle finding algorithm. Below diagram shows the situation when cycle is found.



We can conclude below from above diagram

Distance traveled by fast pointer = 2 * (Distance traveled by slow pointer)

$$(m + n * x + k) = 2 * (m + n * y + k)$$

Note that before meeting the point shown above, fast was moving at twice speed.

x --> Number of complete cyclic rounds made by fast pointer before they meet first time

y --> Number of complete cyclic rounds made by slow pointer before they meet first time

From above equation, we can conclude below

$$m + k = (x-2y)*n$$

Which means **m+k is a multiple of n**.

So if we start moving both pointers again at **same speed** such that one pointer (say slow) begins from head node of linked list and other pointer (say fast) begins from meeting point. When slow pointer reaches beginning of linked list (has made m steps). Fast pointer would have made also moved m steps as they are now moving same pace. Since m+k is a multiple of n and fast starts from k, they would meet at the beginning. Can they meet before also? No because slow pointer enters the cycle first time after m steps.

C++

```
// C++ program to detect and remove loop
#include<bits/stdc++.h>
using namespace std;

struct Node
{
    int key;
    struct Node *next;
};

Node *newNode(int key)
{
    Node *temp = new Node;
    temp->key = key;
    temp->next = NULL;
    return temp;
}

// A utility function to print a linked list
void printList(Node *head)
{
    while (head != NULL)
    {
        cout << head->key << " ";
        head = head->next;
    }
    cout << endl;
}

void detectAndRemoveLoop(Node *head)
{
    Node *slow = head;
    Node *fast = head->next;

    // Search for loop using slow and fast pointers
    while (fast && fast->next)
    {
        if (slow == fast)
            break;
        slow = slow->next;
        fast = fast->next->next;
    }

    /* If loop exists */
    if (slow == fast)
    {
        slow = head;
        while (slow != fast->next)
        {
            slow = slow->next;
            fast = fast->next;
        }

        /* since fast->next is the looping point */
        fast->next = NULL; /* remove loop */
    }
}

/* Driver program to test above function */
int main()
{
    Node *head = newNode(50);
    head->next = newNode(20);
    head->next->next = newNode(15);
    head->next->next->next = newNode(4);
    head->next->next->next->next = newNode(10);

    /* Create a loop for testing */
    head->next->next->next->next->next = head->next->next;

    detectAndRemoveLoop(head);

    printf("Linked List after removing loop \n");
    printList(head);

    return 0;
}
```

Java

```
// Java program to detect and remove loop in linked list

class LinkedList {

    static Node head;

    static class Node {

        int data;
        Node next;

        Node(int d) {
            data = d;
            next = null;
        }
    }

    // Function that detects loop in the list
    void detectAndRemoveLoop(Node node) {
        Node slow = node;
```

```

Node fast = node.next;

// Search for loop using slow and fast pointers
while (fast != null && fast.next != null) {
    if (slow == fast) {
        break;
    }
    slow = slow.next;
    fast = fast.next.next;
}

/* If loop exists */
if (slow == fast) {
    slow = node;
    while (slow != fast.next) {
        slow = slow.next;
        fast = fast.next;
    }

    /* since fast->next is the looping point */
    fast.next = null; /* remove loop */
}
}

// Function to print the linked list
void printList(Node node) {
    while (node != null) {
        System.out.print(node.data + " ");
        node = node.next;
    }
}

// Driver program to test above functions
public static void main(String[] args) {
    LinkedList list = new LinkedList();
    list.head = new Node(50);
    list.head.next = new Node(20);
    list.head.next.next = new Node(15);
    list.head.next.next.next = new Node(4);
    list.head.next.next.next.next = new Node(10);

    // Creating a loop for testing
    head.next.next.next.next.next = head.next.next;
    list.detectAndRemoveLoop(head);
    System.out.println("Linked List after removing loop : ");
    list.printList(head);
}
}

// This code has been contributed by Mayank Jaiswal

```

Python

```

# Python program to detect and remove loop

# Node class
class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    # Function to insert a new node at the beginning
    def push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        self.head = new_node

    def detectAndRemoveLoop(self):
        slow = self.head
        fast = self.head.next

        # Search for loop using slow and fast pointers
        while(fast is not None):
            if fast.next is None:
                break
            if slow == fast :
                break
            slow = slow.next
            fast = fast.next.next

        # If loop exists
        if slow == fast :
            slow = self.head
            while(slow != fast.next):
                slow = slow.next
                fast = fast.next

        # Since fast.next is the looping point
        fast.next = None # Remove loop

    # Utility function to print the linked LinkedList
    def printList(self):
        temp = self.head
        while(temp):
            print temp.data,
            temp = temp.next

# Driver program
lList = LinkedList()

```

```

l1list.head = Node(50)
l1list.head.next = Node(20)
l1list.head.next.next = Node(15)
l1list.head.next.next.next = Node(4)
l1list.head.next.next.next.next = Node(10)

#Create a loop for testing
l1list.head.next.next.next.next.next = l1list.head.next.next

l1list.detectAndRemoveLoop()

print "Linked List after removing loop"
l1list.printList()

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

```

Linked List after removing loop
50 20 15 4 10

```

Thanks to [Gaurav Ahirwar](#) for suggesting above solution.

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

[Linked Lists](#)

Add two numbers represented by linked lists | Set 1

Given two numbers represented by two lists, write a function that returns sum list. The sum list is list representation of addition of two input numbers.

Example 1

```

Input:
First List: 5->6->3 // represents number 365
Second List: 8->4->2 // represents number 248
Output
Resultant list: 3->1->6 // represents number 613

```

Example 2

```

Input:
First List: 7->5->9->4->6 // represents number 64957
Second List: 8->4 // represents number 48
Output
Resultant list: 5->0->0->5->6 // represents number 65005

```

Solution

Traverse both lists. One by one pick nodes of both lists and add the values. If sum is more than 10 then make carry as 1 and reduce sum. If one list has more elements than the other then consider remaining values of this list as 0. Following is the implementation of this approach.

C

```

#include<stdio.h>
#include<stdlib.h>

/* Linked list node */
struct node
{
    int data;
    struct node* next;
};

/* Function to create a new node with given data */
struct node* newNode(int data)
{
    struct node* new_node = (struct node*) malloc(sizeof(struct node));
    new_node->data = data;
    new_node->next = NULL;
    return new_node;
}

/* Function to insert a node at the beginning of the Doubly Linked List */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node = newNode(new_data);

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Adds contents of two linked lists and return the head node of resultant list */
struct node* addTwoLists (struct node* first, struct node* second)
{
    struct node* res = NULL; // res is head node of the resultant list
    struct node* temp, *prev = NULL;
    int carry = 0, sum;

    while (first != NULL || second != NULL) //while both lists exist
    {
        // Calculate value of next digit in resultant list.
        // The next digit is sum of following things
        // (i) Carry
        // (ii) Next digit of first list (if there is a next digit)
        // (ii) Next digit of second list (if there is a next digit)
        sum = carry + (first? first->data: 0) + (second? second->data: 0);

        // update carry for next calculation
        carry = (sum >= 10)? 1 : 0;

        // update sum if it is greater than 10
        sum = sum % 10;
    }

```

```

// Create a new node with sum as data
temp = newNode(sum);

// if this is the first node then set it as head of the resultant list
if(res == NULL)
    res = temp;
else // If this is not the first node then connect it to the rest.
    prev->next = temp;

// Set prev for next insertion
prev = temp;

// Move first and second pointers to next nodes
if (first) first = first->next;
if (second) second = second->next;
}

if (carry > 0)
    temp->next = newNode(carry);

// return head of the resultant list
return res;
}

// A utility function to print a linked list
void printList(struct node *node)
{
    while(node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
    printf("\n");
}

/* Driver program to test above function */
int main(void)
{
    struct node* res = NULL;
    struct node* first = NULL;
    struct node* second = NULL;

    // create first list 7->5->9->4->6
    push(&first, 6);
    push(&first, 4);
    push(&first, 9);
    push(&first, 5);
    push(&first, 7);
    printf("First List is ");
    printList(first);

    // create second list 8->4
    push(&second, 4);
    push(&second, 8);
    printf("Second List is ");
    printList(second);

    // Add the two lists and see result
    res = addTwoLists(first, second);
    printf("Resultant list is ");
    printList(res);

    return 0;
}

```

Java

```

// Java program to delete a given node in linked list under given constraints

class LinkedList {

    static Node head1, head2;

    static class Node {

        int data;
        Node next;

        Node(int d) {
            data = d;
            next = null;
        }
    }

    /* Adds contents of two linked lists and return the head node of resultant list */
    Node addTwoLists(Node first, Node second) {
        Node res = null; // res is head node of the resultant list
        Node prev = null;
        Node temp = null;
        int carry = 0, sum;

        while (first != null || second != null) //while both lists exist
        {
            // Calculate value of next digit in resultant list.
            // The next digit is sum of following things
            // (i) Carry
            // (ii) Next digit of first list (if there is a next digit)
            // (ii) Next digit of second list (if there is a next digit)
            sum = carry + (first != null ? first.data : 0)
                + (second != null ? second.data : 0);

            // update carry for next calculation
            carry = (sum >= 10) ? 1 : 0;

            // update sum if it is greater than 10
            sum = sum % 10;

            // Create a new node with sum as data
            temp = new Node(sum);

            // if this is the first node then set it as head of
            // the resultant list

```

```

        if (res == null) {
            res = temp;
        } else // If this is not the first node then connect it to the rest.
        {
            prev.next = temp;
        }

        // Set prev for next insertion
        prev = temp;

        // Move first and second pointers to next nodes
        if (first != null) {
            first = first.next;
        }
        if (second != null) {
            second = second.next;
        }
    }

    if (carry > 0) {
        temp.next = new Node(carry);
    }

    // return head of the resultant list
    return res;
}
/* Utility function to print a linked list */

void printList(Node head) {
    while (head != null) {
        System.out.print(head.data + " ");
        head = head.next;
    }
    System.out.println("");
}

public static void main(String[] args) {
    LinkedList list = new LinkedList();

    // creating first list
    list.head1 = new Node(7);
    list.head1.next = new Node(5);
    list.head1.next.next = new Node(9);
    list.head1.next.next.next = new Node(4);
    list.head1.next.next.next.next = new Node(6);
    System.out.print("First List is ");
    list.printList(head1);

    // creating second list
    list.head2 = new Node(8);
    list.head2.next = new Node(4);
    System.out.print("Second List is ");
    list.printList(head2);

    // add the two lists and see the result
    Node rs = list.addTwoLists(head1, head2);
    System.out.print("Resultant List is ");
    list.printList(rs);
}
}

// this code has been contributed by Mayank Jaiswal

```

Python

```

# Python program to add two numbers represented by linked list

# Node class
class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    # Function to insert a new node at the beginning
    def push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        self.head = new_node

    # Add contents of two linked lists and return the head
    # node of resultant list
    def addTwoLists(self, first, second):
        prev = None
        temp = None
        carry = 0

        # While both list exists
        while(first is not None or second is not None):

            # Calculate the value of next digit in
            # resultant list
            # The next digit is sum of following things
            # (i) Carry
            # (ii) Next digit of first list (if there is a
            # next digit)
            # (iii) Next digit of second list (if there
            # is a next digit)
            fdata = 0 if first is None else first.data
            sdata = 0 if second is None else second.data
            Sum = carry + fdata + sdata

            # update carry for next calculation
            carry = 1 if Sum >= 10 else 0

```

```

# update sum if it is greater than 10
Sum = Sum if Sum < 10 else Sum % 10

# Create a new node with sum as data
temp = Node(Sum)

# if this is the first node then set it as head
# of resultant list
if self.head is None:
    self.head = temp
else :
    prev.next = temp

# Set prev for next insertion
prev = temp

# Move first and second pointers to next nodes
if first is not None:
    first = first.next
if second is not None:
    second = second.next

if carry > 0:
    temp.next = Node(carry)

# Utility function to print the linked LinkedList
def printList(self):
    temp = self.head
    while(temp):
        print temp.data,
        temp = temp.next

# Driver program to test above function
first = LinkedList()
second = LinkedList()

# Create first list
first.push(6)
first.push(4)
first.push(9)
first.push(5)
first.push(7)
print "First List is ",
first.printList()

# Create second list
second.push(4)
second.push(8)
print "\nSecond List is ",
second.printList()

# Add the two lists and see result
res = LinkedList()
res.addTwoLists(first.head, second.head)
print "\nResultant list is ",
res.printList()

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

```

First List is 7 5 9 4 6
Second List is 8 4
Resultant list is 5 0 0 5 6

```

Time Complexity: $O(m + n)$ where m and n are number of nodes in first and second lists respectively.

Related Article : [Add two numbers represented by linked lists | Set 2](#)

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

[Linked Lists](#)

Rotate a Linked List

Given a singly linked list, rotate the linked list counter-clockwise by k nodes. Where k is a given positive integer. For example, if the given linked list is $10 \rightarrow 20 \rightarrow 30 \rightarrow 40 \rightarrow 50 \rightarrow 60$ and k is 4, the list should be modified to $50 \rightarrow 60 \rightarrow 10 \rightarrow 20 \rightarrow 30 \rightarrow 40$. Assume that k is smaller than the count of nodes in linked list.

We strongly recommend that you click here and practice it, before moving on to the solution.

To rotate the linked list, we need to change next of k th node to NULL, next of last node to previous head node, and finally change head to $(k+1)$ th node. So we need to get hold of three nodes: k th node, $(k+1)$ th node and last node.

Traverse the list from beginning and stop at k th node. Store pointer to k th node. We can get $(k+1)$ th node using k thNode->next. Keep traversing till end and store pointer to last node also. Finally, change pointers as stated above.

C/C++

```

// C/C++ program to rotate a linked list counter clock wise

#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

// This function rotates a linked list counter-clockwise and
// updates the head. The function assumes that k is smaller
// than size of linked list. It doesn't modify the list if

```



```

// k is greater than or equal to size
void rotate(struct node **head_ref, int k)
{
    if (k == 0)
        return;

    // Let us understand the below code for example k = 4 and
    // list = 10->20->30->40->50->60.
    struct node* current = *head_ref;

    // current will either point to kth or NULL after this loop.
    // current will point to node 40 in the above example
    int count = 1;
    while (count < k && current != NULL)
    {
        current = current->next;
        count++;
    }

    // If current is NULL, k is greater than or equal to count
    // of nodes in linked list. Don't change the list in this case
    if (current == NULL)
        return;

    // current points to kth node. Store it in a variable.
    // kthNode points to node 40 in the above example
    struct node *kthNode = current;

    // current will point to last node after this loop
    // current will point to node 60 in the above example
    while (current->next != NULL)
        current = current->next;

    // Change next of last node to previous head
    // Next of 60 is now changed to node 10
    current->next = *head_ref;

    // Change head to (k+1)th node
    // head is now changed to node 50
    *head_ref = kthNode->next;

    // change next of kth node to NULL
    // next of 40 is now NULL
    kthNode->next = NULL;
}

/* UTILITY FUNCTIONS */
/* Function to push a node */
void push (struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Function to print linked list */
void printList(struct node *node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Drier program to test above function*/
int main(void)
{
    /* Start with the empty list */
    struct node* head = NULL;

    // create a list 10->20->30->40->50->60
    for (int i = 60; i > 0; i -= 10)
        push(&head, i);

    printf("Given linked list \n");
    printList(head);
    rotate(&head, 4);

    printf("\nRotated Linked list \n");
    printList(head);

    return (0);
}

```

Java

```

// Java program to rotate a linked list

class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;
        Node(int d)
        {
            data = d;
            next = null;
        }
    }
}

```

```

// This function rotates a linked list counter-clockwise
// and updates the head. The function assumes that k is
// smaller than size of linked list. It doesn't modify
// the list if k is greater than or equal to size
void rotate(int k)
{
    if (k == 0) return;

    // Let us understand the below code for example k = 4
    // and list = 10->20->30->40->50->60.
    Node current = head;

    // current will either point to kth or NULL after this
    // loop. current will point to node 40 in the above example
    int count = 1;
    while (count < k && current != null)
    {
        current = current.next;
        count++;
    }

    // If current is NULL, k is greater than or equal to count
    // of nodes in linked list. Don't change the list in this case
    if (current == null)
        return;

    // current points to kth node. Store it in a variable.
    // kthNode points to node 40 in the above example
    Node kthNode = current;

    // current will point to last node after this loop
    // current will point to node 60 in the above example
    while (current.next != null)
        current = current.next;

    // Change next of last node to previous head
    // Next of 60 is now changed to node 10

    current.next = head;

    // Change head to (k+1)th node
    // head is now changed to node 50
    head = kthNode.next;

    // change next of kth node to null
    kthNode.next = null;
}

/* Given a reference (pointer to pointer) to the head
of a list and an int, push a new node on the front
of the list. */
void push(int new_data)
{
    /* 1 & 2: Allocate the Node &
       Put in the data */
    Node new_node = new Node(new_data);

    /* 3. Make next of new Node as head */
    new_node.next = head;

    /* 4. Move the head to point to new Node */
    head = new_node;
}

void printList()
{
    Node temp = head;
    while(temp != null)
    {
        System.out.print(temp.data+" ");
        temp = temp.next;
    }
    System.out.println();
}

/* Driver program to test above functions */
public static void main(String args[])
{
    LinkedList llist = new LinkedList();

    // create a list 10->20->30->40->50->60
    for (int i = 60; i >= 10; i -= 10)
        llist.push(i);

    System.out.println("Given list");
    llist.printList();

    llist.rotate(4);

    System.out.println("Rotated Linked List");
    llist.printList();
}
/* This code is contributed by Rajat Mishra */

```

Python

```

# Python program to rotate a linked list

# Node class
class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    # Function to initialize head
    def __init__(self):

```

```

self.head = None

# Function to insert a new node at the beginning
def push(self, new_data):
    # allocate node and put the data
    new_node = Node(new_data)

    # Make next of new node as head
    new_node.next = self.head

    # move the head to point to the new Node
    self.head = new_node

# Utility function to print the linked LinkedList
def printList(self):
    temp = self.head
    while(temp):
        print temp.data,
        temp = temp.next

# This function rotates a linked list counter-clockwise and
# updates the head. The function assumes that k is smaller
# than size of linked list. It doesn't modify the list if
# k is greater than or equal to size
def rotate(self, k):
    if k == 0:
        return

    # Let us understand the below code for example k = 4
    # and list = 10->20->30->40->50->60
    current = self.head

    # current will either point to kth or NULL after
    # this loop
    # current will point to node 40 in the above example
    count = 1
    while(count < k and current is not None):
        current = current.next
        count += 1

    # If current is None, k is greater than or equal
    # to count of nodes in linked list. Don't change
    # the list in this case
    if current is None:
        return

    # current points to kth node. Store it in a variable
    # kth node points to node 40 in the above example
    kthNode = current

    # current will point to last node after this loop
    # current will point to node 60 in above example
    while(current.next is not None):
        current = current.next

    # Change next of last node to previous head
    # Next of 60 is now changed to node 10
    current.next = self.head

    # Change head to (k+1)th node
    # head is not changed to node 50
    self.head = kthNode.next

    # change next of kth node to NULL
    # next of 40 is not NULL
    kthNode.next = None

# Driver program to test above function
l1 = LinkedList()

# Create a list 10->20->30->40->50->60
for i in range(60, 0, -10):
    l1.push(i)

print "Given linked list"
l1.printList()
l1.rotate(4)

print "\nRotated Linked list"
l1.printList()

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

```

Given linked list
10 20 30 40 50 60
Rotated Linked list
50 60 10 20 30 40

```

Time Complexity: O(n) where n is the number of nodes in Linked List. The code traverses the linked list only once.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Linked Lists
rotation

Generic Linked List in C

Unlike C++ and Java, C doesn't support generics. How to create a linked list in C that can be used for any data type? In C, we can use [void pointer](#) and function pointer to implement the same functionality. The great thing about void pointer is it can be used to point to any data type. Also, size of all types of pointers is always same, so we can always allocate a linked list node. Function pointer is needed process actual content stored at address pointed by void pointer.

Following is a sample C code to demonstrate working of generic linked list.

```

// C program for generic linked list
#include<stdio.h>

```

```
#include<stdlib.h>

/* A linked list node */
struct node
{
    // Any data type can be stored in this node
    void *data;

    struct node *next;
};

/* Function to add a node at the beginning of Linked List.
This function expects a pointer to the data to be added
and size of the data type */
void push(struct node** head_ref, void *new_data, size_t data_size)
{
    // Allocate memory for node
    struct node* new_node = (struct node*)malloc(sizeof(struct node));

    new_node->data = malloc(data_size);
    new_node->next = (*head_ref);

    // Copy contents of new_data to newly allocated memory.
    // Assumption: char takes 1 byte.
    int i;
    for (i=0; i<data_size; i++)
        *(char *) (new_node->data + i) = *(char *) (new_data + i);

    // Change head pointer as new node is added at the beginning
    (*head_ref) = new_node;
}

/* Function to print nodes in a given linked list. fptr is used
to access the function to be used for printing current node data.
Note that different data types need different specifier in printf() */
void printList(struct node *node, void (*fptr)(void *))
{
    while (node != NULL)
    {
        (*fptr)(node->data);
        node = node->next;
    }
}

// Function to print an integer
void printInt(void *n)
{
    printf(" %d", *(int *)n);
}

// Function to print a float
void printFloat(void *f)
{
    printf(" %f", *(float *)f);
}

/* Driver program to test above function */
int main()
{
    struct node *start = NULL;

    // Create and print an int linked list
    unsigned int_size = sizeof(int);
    int arr[] = {10, 20, 30, 40, 50}, i;
    for (i=4; i>=0; i--)
        push(&start, &arr[i], int_size);
    printf("Created integer linked list is \n");
    printList(start, printInt);

    // Create and print a float linked list
    unsigned float_size = sizeof(float);
    start = NULL;
    float arr2[] = {10.1, 20.2, 30.3, 40.4, 50.5};
    for (i=4; i>=0; i--)
        push(&start, &arr2[i], float_size);
    printf("\n\nCreated float linked list is \n");
    printList(start, printFloat);

    return 0;
}
```

Output:

```
Created integer linked list is
10 20 30 40 50

Created float linked list is
10.100000 20.200001 30.299999 40.400002 50.500000
```

This article is contributed by **Himanshu Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
C/C++ Puzzles
Linked Lists
cpp-pointer

Circular Linked List | Set 1 (Introduction and Applications)

We have discussed singly and doubly linked lists in the following posts.

[Introduction to Linked List & Insertion](#)
[Doubly Linked List Introduction and Insertion](#)

Circular linked list is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.



Advantages of Circular Linked Lists:

- 1) Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.
- 2) Useful for implementation of queue. Unlike [this](#) implementation, we don't need to maintain two pointers for front and rear if we use circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of last.
- 3) Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list. (Source <http://web.eecs.utk.edu/~bvz/cs140/notes/Dllists/>)
- 4) Circular Doubly Linked Lists are used for implementation of advanced data structures like [Fibonacci Heap](#).

Circular Singly Linked List | Insertion

Please write comments if you find any bug in above code/algorithm, or find other ways to solve the same problem

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: [Linked List](#)

Circular Singly Linked List | Insertion

We have discussed Singly and Circular Linked List in the following post:

[Singly Linked List](#)

[Circular Linked List](#)

Why Circular? In a singly linked list, for accessing any node of linked list, we start traversing from the first node. If we are at any node in the middle of the list, then it is not possible to access nodes that precede the given node. This problem can be solved by slightly altering the structure of singly linked list. In a singly linked list, next part (pointer to next node) is NULL, if we utilize this link to point to the first node then we can reach preceding nodes. Refer [this](#) for more advantages of circular linked lists.

The structure thus formed is circular singly linked list look like this:



In this post, implementation and insertion of a node in a Circular Linked List using singly linked list are explained.

Implementation

To implement a circular singly linked list, we take an external pointer that points to the last node of the list. If we have a pointer *last* pointing to the last node, then *last* -> *next* will point to the first node.



The pointer *last* points to node Z and *last* -> *next* points to node P.

Why have we taken a pointer that points to the last node instead of first node ?

For insertion of node in the beginning we need to traverse the whole list. Also, for insertion at the end, the whole list has to be traversed. If instead of *start* pointer we take a pointer to the last node then in both the cases there won't be any need to traverse the whole list. So insertion at the beginning or at the end takes constant time irrespective of the length of the list.

Insertion

A node can be added in three ways:

- Insertion in an empty list
- Insertion at the beginning of the list
- Insertion at the end of the list
- Insertion in between the nodes

Insertion in an empty List

Initially when the list is empty, *last* pointer will be NULL.



After inserting a node T,



After insertion, T is the last node so pointer *last* points to node T. And Node T is first and last node, so T is pointing to itself.

Function to insert node in an empty List,

```
struct Node *addToEmpty(struct Node *last, int data)
{
    // This function is only for empty list
    if (last != NULL)
        return last;

    // Creating a node dynamically.
    struct Node *last =
        (struct Node *)malloc(sizeof(struct Node));

    // Assigning the data.
    last->data = data;

    // Note : list was empty. We link single node
    // to itself.
    last->next = last;

    return last;
}
```

Insertion at the beginning of the list

To Insert a node at the beginning of the list, follow these step:

1. Create a node, say T.
2. Make T -> next = last -> next.
3. last -> next = T.



After insertion,



Function to insert node in the beginning of the List,

```
struct Node *addBegin(struct Node *last, int data)
{
    if (last == NULL)
        return addToEmpty(last, data);

    // Creating a node dynamically.
    struct Node *temp
        = (struct Node *)malloc(sizeof(struct Node));
```

```
// Assigning the data.
temp -> data = data;

// Adjusting the links.
temp -> next = last -> next;
last -> next = temp;

return last;
}
```

Insertion at the end of the list

To Insert a node at the end of the list, follow these step:

1. Create a node, say T.
2. Make T -> next = last -> next;
3. last -> next = T.
4. last = T.



After insertion,



Function to insert node in the end of the List,

```
struct Node *addEnd(struct Node *last, int data)
{
    if (last == NULL)
        return addToEmpty(last, data);

    // Creating a node dynamically.
    struct Node *temp =
        (struct Node *)malloc(sizeof(struct Node));

    // Assigning the data.
    temp -> data = data;

    // Adjusting the links.
    temp -> next = last -> next;
    last -> next = temp;
    last = temp;

    return last;
}
```

Insertion in between the nodes

To Insert a node at the end of the list, follow these step:

1. Create a node, say T.
2. Search the node after which T need to be insert, say that node be P.
3. Make T -> next = P -> next;
4. P -> next = T.

Suppose 12 need to be insert after node having value 10,



After searching and insertion,



Function to insert node in the end of the List,

```
struct Node *addAfter(struct Node *last, int data, int item)
{
    if (last == NULL)
        return NULL;

    struct Node *temp, *p;
    p = last -> next;

    // Searching the item.
    do
    {
        if (p -> data == item)
        {
            // Creating a node dynamically.
            temp = (struct Node *)malloc(sizeof(struct Node));

            // Assigning the data.
            temp -> data = data;

            // Adjusting the links.
            temp -> next = p -> next;

            // Checking for the last node.
            if (p == last)
                last = temp;

            return last;
        }
        p = p -> next;
    } while (p != last -> next);

    cout << item << " not present in the list." << endl;
    return last;
}
```

Following is a complete program that uses all of the above methods to create a circular singly linked list.

```
#include<bits/stdc++.h>
using namespace std;

struct Node
{
    int data;
    struct Node *next;
};

struct Node *addToEmpty(struct Node *last, int data)
{
    // This function is only for empty list
    if (last != NULL)
        return last;
}
```

```

// Creating a node dynamically.
struct Node *temp =
    (struct Node *)malloc(sizeof(struct Node));

// Assigning the data.
temp -> data = data;
last = temp;

// Creating the link.
last -> next = last;

return last;
}

struct Node *addBegin(struct Node *last, int data)
{
    if (last == NULL)
        return addToEmpty(last, data);

    struct Node *temp =
        (struct Node *)malloc(sizeof(struct Node));

    temp -> data = data;
    temp -> next = last -> next;
    last -> next = temp;

    return last;
}

struct Node *addEnd(struct Node *last, int data)
{
    if (last == NULL)
        return addToEmpty(last, data);

    struct Node *temp =
        (struct Node *)malloc(sizeof(struct Node));

    temp -> data = data;
    temp -> next = last -> next;
    last -> next = temp;
    last = temp;

    return last;
}

struct Node *addAfter(struct Node *last, int data, int item)
{
    if (last == NULL)
        return NULL;

    struct Node *temp, *p;
    p = last -> next;
    do
    {
        if (p -> data == item)
        {
            temp = (struct Node *)malloc(sizeof(struct Node));
            temp -> data = data;
            temp -> next = p -> next;
            if (p == last)
                last = temp;
            return last;
        }
        p = p -> next;
    } while (p != last -> next);

    cout << item << " not present in the list." << endl;
    return last;
}

void traverse(struct Node *last)
{
    struct Node *p;

    // If list is empty, return.
    if (last == NULL)
    {
        cout << "List is empty." << endl;
        return;
    }

    // Pointing to first Node of the list.
    p = last -> next;

    // Traversing the list.
    do
    {
        cout << p -> data << " ";
        p = p -> next;
    }
    while (p != last -> next);
}

// Driven Program
int main()
{
    struct Node *last = NULL;

    last = addToEmpty(last, 6);
    last = addBegin(last, 4);
    last = addBegin(last, 2);
    last = addEnd(last, 8);
    last = addEnd(last, 12);
    last = addAfter(last, 10, 8);

    traverse(last);

    return 0;
}

```

Output:

This article is contributed by **Anuj Chauhan**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

[Linked Lists](#)

Circular Linked List | Set 2 (Traversal)

We have discussed [Circular Linked List Introduction and Applications](#), in the previous post on Circular Linked List. In this post, traversal operation is discussed.



In a conventional linked list, we traverse the list from the head node and stop the traversal when we reach NULL. In a circular linked list, we stop traversal when we reach the first node again. Following is C code for linked list traversal.

```
/* Function to traverse a given Circular linked list and print nodes */
void printList(struct node *first)
{
    struct node *temp = first;

    // If linked list is not empty
    if (first != NULL)
    {
        // Keep printing nodes till we reach the first node again
        do
        {
            printf("%d ", temp->data);
            temp = temp->next;
        }
        while (temp != first);
    }
}
```

Complete C program to demonstrate traversal. Following is complete C program to demonstrate traversal of circular linked list.

C/C++

```
#include<stdio.h>
#include<stdlib.h>

/* structure for a node */
struct node
{
    int data;
    struct node *next;
};

/* Function to insert a node at the beginning of a Circular
linked list */
void push(struct node **head_ref, int data)
{
    struct node *ptr1 = (struct node *)malloc(sizeof(struct node));
    struct node *temp = *head_ref;
    ptr1->data = data;
    ptr1->next = *head_ref;

    /* If linked list is not NULL then set the next of last node */
    if (*head_ref != NULL)
    {
        while (temp->next != *head_ref)
            temp = temp->next;
        temp->next = ptr1;
    }
    else
        ptr1->next = ptr1; /* For the first node */

    *head_ref = ptr1;
}

/* Function to print nodes in a given Circular linked list */
void printList(struct node *head)
{
    struct node *temp = head;
    if (head != NULL)
    {
        do
        {
            printf("%d ", temp->data);
            temp = temp->next;
        }
        while (temp != head);
    }
}

/* Driver program to test above functions */
int main()
{
    /* Initialize lists as empty */
    struct node *head = NULL;

    /* Created linked list will be 11->2->56->12 */
    push(&head, 12);
    push(&head, 56);
    push(&head, 2);
    push(&head, 11);

    printf("Contents of Circular Linked List\n");
    printList(head);

    return 0;
}
```


Python

```
# Python program to demonstrate circular linked list traversal

# Structure for a Node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.next = None

class CircularLinkedList:

    # Constructor to create a empty circular linked list
    def __init__(self):
        self.head = None

    # Function to insert a node at the beginning of a
    # circular linked list
    def push(self, data):
        ptr1 = Node(data)
        temp = self.head

        ptr1.next = self.head

        # If linked list is not None then set the next of
        # last node
        if self.head is not None:
            while(temp.next != self.head):
                temp = temp.next
            temp.next = ptr1

        else:
            ptr1.next = ptr1 # For the first node

        self.head = ptr1

    # Function to print nodes in a given circular linked list
    def printList(self):
        temp = self.head
        if self.head is not None:
            while(True):
                print "%d" %(temp.data),
                temp = temp.next
                if (temp == self.head):
                    break

# Driver program to test above function

# Initialize list as empty
clist = CircularLinkedList()

# Created linked list will be 11->2->56->12
clist.push(12)
clist.push(56)
clist.push(2)
clist.push(11)

print "Contents of circular Linked List"
clist.printList()

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
Contents of Circular Linked List
11 2 56 12
```

You may like to see following posts on Circular Linked List

[Split a Circular Linked List into two halves](#)

[Sorted insert for circular linked list](#)

We will soon be discussing implementation of insert delete operations for circular linked lists.

Please write comments if you find any bug in above code/algorithm, or find other ways to solve the same problem

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: [Linked List](#)

Split a Circular Linked List into two halves

Asked by [Bharani](#)



Original Linked List



Result Linked List 1



Result Linked List 2

Thanks to [Geek4u](#) for suggesting the algorithm.

- 1) Store the mid and last pointers of the circular linked list using tortoise and hare algorithm.
- 2) Make the second half circular.
- 3) Make the first half circular.
- 4) Set head (or start) pointers of the two linked lists.

In the below implementation, if there are odd nodes in the given circular linked list then the first result list has 1 more node than the second result list.

C

```
/* Program to split a circular linked list into two halves */
#include<stdio.h>
#include<stdlib.h>

/* structure for a node */
struct node
{
    int data;
    struct node *next;
};

/* Function to split a list (starting with head) into two lists.
head1_ref and head2_ref are references to head nodes of
the two resultant linked lists */
void splitList(struct node *head, struct node **head1_ref,
              struct node **head2_ref)
{
    struct node *slow_ptr = head;
    struct node *fast_ptr = head;

    if(head == NULL)
        return;

    /* If there are odd nodes in the circular list then
    fast_ptr->next becomes head and for even nodes
    fast_ptr->next->next becomes head */
    while(fast_ptr->next != head &&
          fast_ptr->next->next != head)
    {
        fast_ptr = fast_ptr->next->next;
        slow_ptr = slow_ptr->next;
    }

    /* If there are even elements in list then move fast_ptr */
    if(fast_ptr->next->next == head)
        fast_ptr = fast_ptr->next;

    /* Set the head pointer of first half */
    *head1_ref = head;

    /* Set the head pointer of second half */
    if(head->next != head)
        *head2_ref = slow_ptr->next;

    /* Make second half circular */
    fast_ptr->next = slow_ptr->next;

    /* Make first half circular */
    slow_ptr->next = head;
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginning of a Circular
linked list */
void push(struct node **head_ref, int data)
{
    struct node *ptr1 = (struct node *)malloc(sizeof(struct node));
    struct node *temp = *head_ref;
    ptr1->data = data;
    ptr1->next = *head_ref;

    /* If linked list is not NULL then set the next of
    last node */
    if(*head_ref != NULL)
    {
        while(temp->next != *head_ref)
            temp = temp->next;
        temp->next = ptr1;
    }
    else
        ptr1->next = ptr1; /* For the first node */

    *head_ref = ptr1;
}

/* Function to print nodes in a given Circular linked list */
void printList(struct node *head)
{
    struct node *temp = head;
    if(head != NULL)
    {
        printf("\n");
        do {
            printf("%d ", temp->data);
            temp = temp->next;
        } while(temp != head);
    }
}

/* Driver program to test above functions */
int main()
{
}
```

```

int list_size, i;

/* Initialize lists as empty */
struct node *head = NULL;
struct node *head1 = NULL;
struct node *head2 = NULL;

/* Created linked list will be 12->56->2->11 */
push(&head, 12);
push(&head, 56);
push(&head, 2);
push(&head, 11);

printf("Original Circular Linked List");
printList(head);

/* Split the list */
splitList(head, &head1, &head2);

printf("\nFirst Circular Linked List");
printList(head1);

printf("\nSecond Circular Linked List");
printList(head2);

getchar();
return 0;
}

```

Java

```

// Java program to delete a node from doubly linked list

class LinkedList {

    static Node head, head1, head2;

    static class Node {

        int data;
        Node next, prev;

        Node(int d) {
            data = d;
            next = prev = null;
        }
    }

    /* Function to split a list (starting with head) into two lists.
    head1_ref and head2_ref are references to head nodes of
    the two resultant linked lists */
    void splitList() {
        Node slow_ptr = head;
        Node fast_ptr = head;

        if (head == null) {
            return;
        }

        /* If there are odd nodes in the circular list then
        fast_ptr->next becomes head and for even nodes
        fast_ptr->next->next becomes head */
        while (fast_ptr.next != head
            && fast_ptr.next.next != head) {
            fast_ptr = fast_ptr.next.next;
            slow_ptr = slow_ptr.next;
        }

        /* If there are even elements in list then move fast_ptr */
        if (fast_ptr.next.next == head) {
            fast_ptr = fast_ptr.next;
        }

        /* Set the head pointer of first half */
        head1 = head;

        /* Set the head pointer of second half */
        if (head.next != head) {
            head2 = slow_ptr.next;
        }
        /* Make second half circular */
        fast_ptr.next = slow_ptr.next;

        /* Make first half circular */
        slow_ptr.next = head;
    }

    /* Function to print nodes in a given singly linked list */
    void printList(Node node) {
        Node temp = node;
        if (node != null) {
            do {
                System.out.print(temp.data + " ");
                temp = temp.next;
            } while (temp != node);
        }
    }

    public static void main(String[] args) {
        LinkedList list = new LinkedList();

        //Created linked list will be 12->56->2->11
        list.head = new Node(12);
        list.head.next = new Node(56);
        list.head.next.next = new Node(2);
        list.head.next.next.next = new Node(11);
        list.head.next.next.next.next = list.head;

        System.out.println("Original Circular Linked list ");
        list.printList(head);

        // Split the list
        list.splitList();
    }
}

```

```

        System.out.println("");
        System.out.println("First Circular List ");
        list.printList(head1);
        System.out.println("");
        System.out.println("Second Circular List ");
        list.printList(head2);
    }
}

// This code has been contributed by Mayank Jaiswal

```

Python

Python program to split circular linked list into two halves

A node structure
class Node:

```

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.next = None

```

Class to create a new Circular Linked list
class CircularLinkedList:

```

    # Constructor to create an empty circular linked list
    def __init__(self):
        self.head = None

```

```

    # Function to insert a node at the beginning of a
    # circular linked list
    def push(self, data):
        ptr1 = Node(data)
        temp = self.head

```

```

        ptr1.next = self.head

```

```

        # If linked list is not None then set the next of
        # last node
        if self.head is not None:
            while(temp.next != self.head):
                temp = temp.next
            temp.next = ptr1

```

```

        else:
            ptr1.next = ptr1 # For the first node

```

```

        self.head = ptr1

```

```

    # Function to print nodes in a given circular linked list
    def printList(self):
        temp = self.head
        if self.head is not None:
            while(True):
                print "%d" %(temp.data),
                temp = temp.next
                if (temp == self.head):
                    break

```

```

    # Function to split a list (starting with head) into
    # two lists. head1 and head2 are the head nodes of the
    # two resultant linked lists
    def splitList(self, head1, head2):
        slow_ptr = self.head
        fast_ptr = self.head

```

```

        if self.head is None:
            return

```

```

        # If there are odd nodes in the circular list then
        # fast_ptr->next becomes head and for even nodes
        # fast_ptr->next->next becomes head
        while(fast_ptr.next != self.head and
              fast_ptr.next.next != self.head):
            fast_ptr = fast_ptr.next.next
            slow_ptr = slow_ptr.next

```

```

        # If there are even elements in list then
        # move fast_ptr
        if fast_ptr.next.next == self.head:
            fast_ptr = fast_ptr.next

```

```

        # Set the head pointer of first half
        head1.head = self.head

```

```

        # Set the head pointer of second half
        if self.head.next != self.head:
            head2.head = slow_ptr.next

```

```

        # Make second half circular
        fast_ptr.next = slow_ptr.next

```

```

        # Make first half circular
        slow_ptr.next = self.head

```

Driver program to test above functions

```

# Initialize lists as empty
head = CircularLinkedList()
head1 = CircularLinkedList()
head2 = CircularLinkedList()

```

```

head.push(12)
head.push(56)
head.push(2)
head.push(11)

```

```

print "Original Circular Linked List"
head.printList()

# Split the list
head.splitList(head1 , head2)

print "\nFirst Circular Linked List"
head1.printList()

print "\nSecond Circular Linked List"
head2.printList()

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

```

Original Circular Linked List
11 2 56 12
First Circular Linked List
11 2
Second Circular Linked List
56 12

```

Time Complexity: $O(n)$

Please write comments if you find any bug in above code/algorithm, or find other ways to solve the same problem

GATE CS Corner Company Wise Coding Practice

[Linked Lists](#)

Sorted insert for circular linked list

Difficulty Level: Rookie

Write a C function to insert a new value in a sorted Circular Linked List (CLL). For example, if the input CLL is following.



After insertion of 7, the above CLL should be changed to following



Algorithm:

Allocate memory for the newly inserted node and put data in the newly allocated node. Let the pointer to the new node be new_node. After memory allocation, following are the three cases that need to be handled.

- 1) *Linked List is empty:*
 - a) since new_node is the only node in CLL, make a self loop.
new_node->next = new_node;
 - b) change the head pointer to point to new node.
*head_ref = new_node;
- 2) *New node is to be inserted just before the head node:*
 - (a) Find out the last node using a loop.
while(current->next != *head_ref)
current = current->next;
 - (b) Change the next of last node.
current->next = new_node;
 - (c) Change next of new node to point to head.
new_node->next = *head_ref;
 - (d) change the head pointer to point to new node.
*head_ref = new_node;
- 3) *New node is to be inserted somewhere after the head:*
 - (a) Locate the node after which new node is to be inserted.
while (current->next!= *head_ref &&
current->next->data < data)
{ current = current->next; }
 - (b) Make next of new_node as next of the located pointer
new_node->next = current->next;
 - (c) Change the next of the located pointer
current->next = new_node;

C

```

#include<stdio.h>
#include<stdlib.h>

/* structure for a node */
struct node
{
    int data;
    struct node *next;
};

/* function to insert a new_node in a list in sorted way.
Note that this function expects a pointer to head node
as this can modify the head of the input linked list */
void sortedInsert(struct node** head_ref, struct node* new_node)
{
    struct node* current = *head_ref;

    // Case 1 of the above algo
    if (current == NULL)
    {
        new_node->next = new_node;
        *head_ref = new_node;
    }

    // Case 2 of the above algo
    else if (current->data >= new_node->data)
    {

```

```

/* If value is smaller than head's value then
we need to change next of last node */
while(current->next != *head_ref)
    current = current->next;
current->next = new_node;
new_node->next = *head_ref;
*head_ref = new_node;
}

// Case 3 of the above algo
else
{
    /* Locate the node before the point of insertion */
    while (current->next!= *head_ref &&
           current->next->data < new_node->data)
        current = current->next;

    new_node->next = current->next;
    current->next = new_node;
}
}

/* Function to print nodes in a given linked list */
void printList(struct node *start)
{
    struct node *temp;

    if(start != NULL)
    {
        temp = start;
        printf("\n");
        do {
            printf("%d ", temp->data);
            temp = temp->next;
        } while(temp != start);
    }
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {12, 56, 2, 11, 1, 90};
    int list_size, i;

    /* start with empty linked list */
    struct node *start = NULL;
    struct node *temp;

    /* Create linked list from the array arr[]
    Created linked list will be 1->2->11->12->56->90 */
    for (i = 0; i < 6; i++)
    {
        temp = (struct node *)malloc(sizeof(struct node));
        temp->data = arr[i];
        sortedInsert(&start, temp);
    }

    printList(start);

    return 0;
}

```

Java

```

// Java program for sorted insert in circular linked list

class Node
{
    int data;
    Node next;

    Node(int d)
    {
        data = d;
        next = null;
    }
}

class LinkedList
{
    Node head;

    // Constructor
    LinkedList() { head = null; }

    /* function to insert a new_node in a list in sorted way.
    Note that this function expects a pointer to head node
    as this can modify the head of the input linked list */
    void sortedInsert(Node new_node)
    {
        Node current = head;

        // Case 1 of the above algo
        if (current == null)
        {
            new_node.next = new_node;
            head = new_node;
        }

        // Case 2 of the above algo
        else if (current.data >= new_node.data)
        {
            /* If value is smaller than head's value then
            we need to change next of last node */
            while (current.next != head)
                current = current.next;

            current.next = new_node;
            new_node.next = head;
            head = new_node;
        }
    }
}

```

```

// Case 3 of the above algo
else
{

    /* Locate the node before the point of insertion */
    while (current.next != head &&
           current.next.data < new_node.data)
        current = current.next;

    new_node.next = current.next;
    current.next = new_node;
}
}

// Utility method to print a linked list
void printList()
{
    if (head != null)
    {
        Node temp = head;
        do
        {
            System.out.print(temp.data + " ");
            temp = temp.next;
        } while (temp != head);
    }
}

// Driver code to test above
public static void main(String[] args)
{
    LinkedList list = new LinkedList();

    // Creating the linkedlist
    int arr[] = new int[] {12, 56, 2, 11, 1, 90};

    /* start with empty linked list */
    Node temp = null;

    /* Create linked list from the array arr[]
    Created linked list will be 1->2->11->12->56->90*/
    for (int i = 0; i < 6; i++)
    {
        temp = new Node(arr[i]);
        list.sortedInsert(temp);
    }

    list.printList();
}
}

// This code has been contributed by Mayank Jaiswal

```

Python

```

# Node class
class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    # Function to insert a new node at the beginning
    def push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        self.head = new_node

    # Utility function to print the linked LinkedList
    def printList(self):
        temp = self.head
        print temp.data,
        temp = temp.next
        while(temp != self.head):
            print temp.data,
            temp = temp.next

""" function to insert a new_node in a list in sorted way.
Note that this function expects a pointer to head node
as this can modify the head of the input linked list """
def sortedInsert(self, new_node):

    current = self.head

    # Case 1 of the above algo
    if current is None:
        new_node.next = new_node
        self.head = new_node

    # Case 2 of the above algo
    elif (current.data >= new_node.data):

        # If value is smaller than head's value then we
        # need to change next of last node
        while current.next != self.head :
            current = current.next
        current.next = new_node
        new_node.next = self.head
        self.head = new_node

    # Case 3 of the above algo
    else:

        # Locate the node before the point of insertion

```

```

while (current.next != self.head and
      current.next.data < new_node.data):
    current = current.next

new_node.next = current.next
current.next = new_node

# Driver program to test the above function
#l1list = LinkedList()
arr = [12, 56, 2, 11, 1, 90]

list_size = len(arr)

# start with empty linked list
start = LinkedList()

# Create linked list from the array arr[]
# Created linked list will be 1->2->11->12->56->90
for i in range(list_size):
    temp = Node(arr[i])
    start.sortedInsert(temp)

start.printList()

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

```
1 2 11 12 56 90
```

Time Complexity: $O(n)$ where n is the number of nodes in the given linked list.

Case 2 of the above algorithm/code can be optimized. Please see [this comment](#) from Pavan. To implement the suggested change we need to modify the case 2 to following.

```

// Case 2 of the above algo
else if (current->data >= new_node->data)
{
    // swap the data part of head node and new node
    // assuming that we have a function swap(int *, int *)
    swap(&(current->data), &(new_node->data));

    new_node->next = (*head_ref)->next;
    (*head_ref)->next = new_node;
}

```

Please write comments if you find the above code/algorithm incorrect, or find other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

[Linked Lists](#)

Doubly Linked List | Set 1 (Introduction and Insertion)

We strongly recommend to refer following post as a prerequisite of this post.

[Linked List Introduction](#)

[Inserting a node in Singly Linked List](#)

A **Doubly Linked List (DLL)** contains an extra pointer, typically called *previous pointer*, together with next pointer and data which are there in singly linked list.

dll



Following is representation of a DLL node in C language.

```

/* Node of a doubly linked list */
struct node
{
    int data;
    struct node *next; // Pointer to next node in DLL
    struct node *prev; // Pointer to previous node in DLL
};

```

Following are advantages/disadvantages of doubly linked list over singly linked list.

Advantages over singly linked list

- 1) A DLL can be traversed in both forward and backward direction.
- 2) The delete operation in DLL is more efficient if pointer to the node to be deleted is given.

In singly linked list, to delete a node, pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we can get the previous node using previous pointer.

Disadvantages over singly linked list

- 1) Every node of DLL Require extra space for an previous pointer. It is possible to implement DLL with single pointer though (See [this](#) and [this](#)).
- 2) All operations require an extra pointer previous to be maintained. For example, in insertion, we need to modify previous pointers together with next pointers. For example in following functions for insertions at different positions, we need 1 or 2 extra steps to set previous pointer.

Insertion

A node can be added in four ways

- 1) At the front of the DLL
- 2) After a given node.
- 3) At the end of the DLL
- 4) Before a given node.

1) Add a node at the front: (A 5 steps process)

The new node is always added before the head of the given Linked List. And newly added node becomes the new head of DLL. For example if the given Linked List is 10152025 and we add an item 5 at the front, then the Linked List becomes 510152025. Let us call the function that adds at the front of the list is push(). The push() must receive a pointer to the head pointer, because push must change the head pointer to point to the new node (See [this](#))

dll_add_front



Following are the 5 steps to add node at the front.

```
/* Given a reference (pointer to pointer) to the head of a list
and an int, inserts a new node on the front of the list. */
void push(struct node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct node* new_node = (struct node*) malloc(sizeof(struct node));

    /* 2. put in the data */
    new_node->data = new_data;

    /* 3. Make next of new node as head and previous as NULL */
    new_node->next = (*head_ref);
    new_node->prev = NULL;

    /* 4. change prev of head node to new node */
    if((*head_ref) != NULL)
        (*head_ref)->prev = new_node ;

    /* 5. move the head to point to the new node */
    (*head_ref) = new_node;
}
```

Four steps of the above five steps are same as [the 4 steps used for inserting at the front in singly linked list](#). The only extra step is to change previous of head.

2) Add a node after a given node.: (A 7 steps process)

We are given pointer to a node as prev_node, and the new node is inserted after the given node.

dll_add_middle



```
/* Given a node as prev_node, insert a new node after the given node */
void insertAfter(struct node* prev_node, int new_data)
{
    /* 1. check if the given prev_node is NULL */
    if (prev_node == NULL)
    {
        printf("the given previous node cannot be NULL");
        return;
    }

    /* 2. allocate new node */
    struct node* new_node = (struct node*) malloc(sizeof(struct node));

    /* 3. put in the data */
    new_node->data = new_data;

    /* 4. Make next of new node as next of prev_node */
    new_node->next = prev_node->next;

    /* 5. Make the next of prev_node as new_node */
    prev_node->next = new_node;

    /* 6. Make prev_node as previous of new_node */
    new_node->prev = prev_node;

    /* 7. Change previous of new_node's next node */
    if (new_node->next != NULL)
        new_node->next->prev = new_node;
}
```

Five of the above steps step process are same as [the 5 steps used for inserting after a given node in singly linked list](#). The two extra steps are needed to change previous pointer of new node and previous pointer of new node's next node.

3) Add a node at the end: (7 steps process)

The new node is always added after the last node of the given Linked List. For example if the given DLL is 510152025 and we add an item 30 at the end, then the DLL becomes 51015202530.

Since a Linked List is typically represented by the head of it, we have to traverse the list till end and then change the next of last node to new node.

dll_add_end



Following are the 7 steps to add node at the end.

```
/* Given a reference (pointer to pointer) to the head
of a DLL and an int, appends a new node at the end */
void append(struct node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct node* new_node = (struct node*) malloc(sizeof(struct node));

    struct node *last = *head_ref; /* used in step 5*/

    /* 2. put in the data */
    new_node->data = new_data;

    /* 3. This new node is going to be the last node, so
    make next of it as NULL*/
    new_node->next = NULL;

    /* 4. If the Linked List is empty, then make the new
    node as head */
    if (*head_ref == NULL)
    {
        new_node->prev = NULL;
        *head_ref = new_node;
        return;
    }

    /* 5. Else traverse till the last node */
    while (last->next != NULL)
        last = last->next;

    /* 6. Change the next of last node */
    last->next = new_node;

    /* 7. Make last node as previous of new node */
    new_node->prev = last;

    return;
}
```

Six of the above 7 steps are same as [the 6 steps used for inserting after a given node in singly linked list](#). The one extra step is needed to change previous pointer of new node.

4) Add a node before a given node

This is left as an exercise for the readers.

A complete working program to test above functions.

Following is complete C program to test above functions.

C

```
// A complete working C program to demonstrate all insertion methods
#include <stdio.h>
#include <stdlib.h>

// A linked list node
struct node
{
    int data;
    struct node *next;
    struct node *prev;
};

/* Given a reference (pointer to pointer) to the head of a list
and an int, inserts a new node on the front of the list */
void push(struct node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct node* new_node = (struct node*) malloc(sizeof(struct node));

    /* 2. put in the data */
    new_node->data = new_data;

    /* 3. Make next of new node as head and previous as NULL */
    new_node->next = (*head_ref);
    new_node->prev = NULL;

    /* 4. change prev of head node to new node */
    if((*head_ref) != NULL)
        (*head_ref)->prev = new_node;

    /* 5. move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Given a node as prev_node, insert a new node after the given node */
void insertAfter(struct node* prev_node, int new_data)
{
    /* 1. check if the given prev_node is NULL */
    if (prev_node == NULL)
    {
        printf("the given previous node cannot be NULL");
        return;
    }

    /* 2. allocate new node */
    struct node* new_node = (struct node*) malloc(sizeof(struct node));
```

```

/* 3. put in the data */
new_node->data = new_data;

/* 4. Make next of new node as next of prev_node */
new_node->next = prev_node->next;

/* 5. Make the next of prev_node as new_node */
prev_node->next = new_node;

/* 6. Make prev_node as previous of new_node */
new_node->prev = prev_node;

/* 7. Change previous of new_node's next node */
if (new_node->next != NULL)
    new_node->next->prev = new_node;
}

/* Given a reference (pointer to pointer) to the head
of a DLL and an int, appends a new node at the end */
void append(struct node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct node* new_node = (struct node*) malloc(sizeof(struct node));

    struct node *last = *head_ref; /* used in step 5*/

    /* 2. put in the data */
    new_node->data = new_data;

    /* 3. This new node is going to be the last node, so
    make next of it as NULL*/
    new_node->next = NULL;

    /* 4. If the Linked List is empty, then make the new
    node as head */
    if (*head_ref == NULL)
    {
        new_node->prev = NULL;
        *head_ref = new_node;
        return;
    }

    /* 5. Else traverse till the last node */
    while (last->next != NULL)
        last = last->next;

    /* 6. Change the next of last node */
    last->next = new_node;

    /* 7. Make last node as previous of new node */
    new_node->prev = last;

    return;
}

// This function prints contents of linked list starting from the given node
void printList(struct node *node)
{
    struct node *last;
    printf("nTraversal in forward direction \n");
    while (node != NULL)
    {
        printf("%d ", node->data);
        last = node;
        node = node->next;
    }

    printf("nTraversal in reverse direction \n");
    while (last != NULL)
    {
        printf("%d ", last->data);
        last = last->prev;
    }
}

/* Driver program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

    // Insert 6. So linked list becomes 6->NULL
    append(&head, 6);

    // Insert 7 at the beginning. So linked list becomes 7->6->NULL
    push(&head, 7);

    // Insert 1 at the beginning. So linked list becomes 1->7->6->NULL
    push(&head, 1);

    // Insert 4 at the end. So linked list becomes 1->7->6->4->NULL
    append(&head, 4);

    // Insert 8, after 7. So linked list becomes 1->7->8->6->4->NULL
    insertAfter(head->next, 8);

    printf("Created DLL is: ");
    printList(head);

    getchar();
    return 0;
}

```

Python

```

# A complete working Python program to demonstrate all
# insertion methods

```

```

# A linked list node
class Node:

```

```

    # Constructor to create a new node

```

```

def __init__(self, data):
    self.data = data
    self.next = None
    self.prev = None

# Class to create a Doubly Linked List
class DoublyLinkedList:

    # Constructor for empty Doubly Linked List
    def __init__(self):
        self.head = None

    # Given a reference to the head of a list and an
    # integer, inserts a new node on the front of list
    def push(self, new_data):

        # 1. Allocates node
        # 2. Put the data in it
        new_node = Node(new_data)

        # 3. Make next of new node as head and
        # previous as None (already None)
        new_node.next = self.head

        # 4. change prev of head node to new_node
        if self.head is not None:
            self.head.prev = new_node

        # 5. move the head to point to the new node
        self.head = new_node

    # Given a node as prev_node, insert a new node after
    # the given node
    def insertAfter(self, prev_node, new_data):

        # 1. Check if the given prev_node is None
        if prev_node is None:
            print "the given previous node cannot be NULL"
            return

        # 2. allocate new node
        # 3. put in the data
        new_node = Node(new_data)

        # 4. Make next of new node as next of prev node
        new_node.next = prev_node.next

        # 5. Make prev_node as previous of new_node
        prev_node.next = new_node

        # 6. Make prev_node as previous of new_node
        new_node.prev = prev_node

        # 7. Change previous of new_node's next node
        if new_node.next is not None:
            new_node.next.prev = new_node

    # Given a reference to the head of DLL and integer,
    # appends a new node at the end
    def append(self, new_data):

        # 1. Allocates node
        # 2. Put in the data
        new_node = Node(new_data)

        # 3. This new node is going to be the last node,
        # so make next of it as None
        new_node.next = None

        # 4. If the Linked List is empty, then make the
        # new node as head
        if self.head is None:
            new_node.prev = None
            self.head = new_node
            return

        # 5. Else traverse till the last node
        last = self.head
        while(last.next is not None):
            last = last.next

        # 6. Change the next of last node
        last.next = new_node

        # 7. Make last node as previous of new node
        new_node.prev = last

    def returnList(self):
        return self.head

    # This function prints contents of linked list
    # starting from the given node
    def printList(self, node):

        print "\nTraversal in forward direction"
        while(node is not None):
            print " %d" %(node.data),
            last = node
            node = node.next

        print "\nTraversal in reverse direction"
        while(last is not None):
            print " %d" %(last.data),
            last = last.prev

# Driver program to test above functions

# Start with empty list
l1 = DoublyLinkedList()

# Insert 6. So the list becomes 6->None
l1.append(6)

# Insert 7 at the beginning.
# So linked list becomes 7->6->None
l1.push(7)

```

```
# Insert 1 at the beginning.
# So linked list becomes 1->7->6->None
l1.push(1)

# Insert 4 at the end.
# So linked list becomes 1->7->6->4->None
l1.append(4)

# Insert 8, after 7.
# So linked list becomes 1->7->8->6->4->None
l1.insertAfter(l1.head.next, 8)

print "Created DLL is: ",
l1.printList(l1.head)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
Created DLL is:
Traversal in forward direction
1 7 8 6 4
Traversal in reverse direction
4 6 8 7 1
```

Also see – [Delete a node in double Link List](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: [Linked List](#)

Delete a node in a Doubly Linked List

[Doubly Link List Set 1](#) | [Introduction and Insertion](#)

Write a function to delete a given node in a doubly linked list.

(a) Original Doubly Linked List



(a) After deletion of head node



(a) After deletion of middle node



(a) After deletion of last node



Algorithm

Let the node to be deleted is *del*.

- 1) If node to be deleted is head node, then change the head pointer to next current head.
- 2) Set *next* of previous to *del*, if previous to *del* exists.
- 3) Set *prev* of next to *del*, if next to *del* exists.

C

```
#include <stdio.h>
#include <stdlib.h>

/* a node of the doubly linked list */
struct node
{
    int data;
    struct node *next;
    struct node *prev;
};

/* Function to delete a node in a Doubly Linked List.
   head_ref -> pointer to head node pointer.
```

```

del --> pointer to node to be deleted. */
void deleteNode(struct node **head_ref, struct node *del)
{
    /* base case */
    if(*head_ref == NULL || del == NULL)
        return;

    /* If node to be deleted is head node */
    if(*head_ref == del)
        *head_ref = del->next;

    /* Change next only if node to be deleted is NOT the last node */
    if(del->next != NULL)
        del->next->prev = del->prev;

    /* Change prev only if node to be deleted is NOT the first node */
    if(del->prev != NULL)
        del->prev->next = del->next;

    /* Finally, free the memory occupied by del */
    free(del);
    return;
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginning of the Doubly Linked List */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data */
    new_node->data = new_data;

    /* since we are adding at the beginning,
    prev is always NULL */
    new_node->prev = NULL;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* change prev of head node to new node */
    if((*head_ref) != NULL)
        (*head_ref)->prev = new_node ;

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Function to print nodes in a given doubly linked list
This function is same as printList() of singly linked list */
void printList(struct node *node)
{
    while(node!=NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Driver program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

    /* Let us create the doubly linked list 10<->8<->4<->2 */
    push(&head, 2);
    push(&head, 4);
    push(&head, 8);
    push(&head, 10);

    printf("\n Original Linked list ");
    printList(head);

    /* delete nodes from the doubly linked list */
    deleteNode(&head, head); /*delete first node*/
    deleteNode(&head, head->next); /*delete middle node*/
    deleteNode(&head, head->next); /*delete last node*/

    /* Modified linked list will be NULL<-8->NULL */
    printf("\n Modified Linked list ");
    printList(head);

    getchar();
}

```

Java

```

// Java program to delete a node from doubly linked list

class LinkedList {

    static Node head = null;

    class Node {

        int data;
        Node next, prev;

        Node(int d) {
            data = d;
            next = prev = null;
        }
    }

    /*Function to delete a node in a Doubly Linked List.
    head_ref --> pointer to head node pointer.
    del --> pointer to node to be deleted.*/
    void deleteNode(Node head_ref, Node del) {

        /* base case */

```

```

if (head == null || del == null) {
    return;
}

/* If node to be deleted is head node */
if (head == del) {
    head = del.next;
}

/* Change next only if node to be deleted is NOT the last node */
if (del.next != null) {
    del.next.prev = del.prev;
}

/* Change prev only if node to be deleted is NOT the first node */
if (del.prev != null) {
    del.prev.next = del.next;
}

/* Finally, free the memory occupied by del */
return;
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginning of the Doubly Linked List */
void push(Node head_ref, int new_data) {

    /* allocate node */
    Node new_node = new Node(new_data);

    /* since we are adding at the beginning,
    prev is always NULL */
    new_node.prev = null;

    /* link the old list off the new node */
    new_node.next = (head);

    /* change prev of head node to new node */
    if ((head) != null) {
        (head).prev = new_node;
    }

    /* move the head to point to the new node */
    (head) = new_node;
}

/*Function to print nodes in a given doubly linked list
This function is same as printList() of singly linked list */
void printList(Node node) {
    while (node != null) {
        System.out.print(node.data + " ");
        node = node.next;
    }
}

public static void main(String[] args) {
    LinkedList list = new LinkedList();

    /* Let us create the doubly linked list 10<->8<->4<->2 */
    list.push(head, 2);
    list.push(head, 4);
    list.push(head, 8);
    list.push(head, 10);

    System.out.println("Original Linked list ");
    list.printList(head);

    /* delete nodes from the doubly linked list */
    list.deleteNode(head, head); /*delete first node*/

    list.deleteNode(head, head.next); /*delete middle node*/

    list.deleteNode(head, head.next); /*delete last node*/
    System.out.println("");

    /* Modified linked list will be NULL<->8<->NULL */
    System.out.println("Modified Linked List");
    list.printList(head);
}
}

// This code has been contributed by Mayank Jaiswal

```

Python

```

# Program to delete a node in doubly linked list

# for Garbage collection
import gc

# A node of the doubly linked list
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:
    # Constructor for empty Doubly Linked List
    def __init__(self):
        self.head = None

    # Function to delete a node in a Doubly Linked List.
    # head_ref -> pointer to head node pointer.
    # dele -> pointer to node to be deleted

    def deleteNode(self, dele):

        # Base Case
        if self.head is None or dele is None:

```

```

return

# If node to be deleted is head node
if self.head == dele:
    self.head = dele.next

# Change next only if node to be deleted is NOT
# the last node
if dele.next is not None:
    dele.next.prev = dele.prev

# Change prev only if node to be deleted is NOT
# the first node
if dele.prev is not None:
    dele.prev.next = dele.next
# Finally, free the memory occupied by dele
# Call python garbage collector
gc.collect()

# Given a reference to the head of a list and an
# integer, inserts a new node on the front of list
def push(self, new_data):

    # 1. Allocates node
    # 2. Put the data in it
    new_node = Node(new_data)

    # 3. Make next of new node as head and
    # previous as None (already None)
    new_node.next = self.head

    # 4. change prev of head node to new_node
    if self.head is not None:
        self.head.prev = new_node

    # 5. move the head to point to the new node
    self.head = new_node

def printList(self, node):
    while(node is not None):
        print node.data,
        node = node.next

# Driver program to test the above functions

# Start with empty list
dll = DoublyLinkedList()

# Let us create the doubly linked list 10<->8<->4<->2
dll.push(2);
dll.push(4);
dll.push(8);
dll.push(10);

print "\n Original Linked List",
dll.printList(dll.head)

# delete nodes from doubly linked list
dll.deleteNode(dll.head)
dll.deleteNode(dll.head.next)
dll.deleteNode(dll.head.next)
# Modified linked list will be NULL<-8->NULL
print "\n Modified Linked List",
dll.printList(dll.head)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Time Complexity: O(1)

Time Complexity: O(1)

Please write comments if you find any of the above codes/algorithms incorrect, or find better ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

[Linked Lists](#)

Reverse a Doubly Linked List

Write a C function to reverse a given Doubly Linked List

See below diagrams for example.

(a) Original Doubly Linked List



(b) Reversed Doubly Linked List



Here is a simple method for reversing a Doubly Linked List. All we need to do is swap prev and next pointers for all nodes, change prev of the head (or start) and change the head pointer in the end.

C


```

/* Program to reverse a doubly linked list */
#include <stdio.h>
#include <stdlib.h>

/* a node of the doubly linked list */
struct node
{
    int data;
    struct node *next;
    struct node *prev;
};

/* Function to reverse a Doubly Linked List */
void reverse(struct node **head_ref)
{
    struct node *temp = NULL;
    struct node *current = *head_ref;

    /* swap next and prev for all nodes of
    doubly linked list */
    while (current != NULL)
    {
        temp = current->prev;
        current->prev = current->next;
        current->next = temp;
        current = current->prev;
    }

    /* Before changing head, check for the cases like empty
    list and list with only one node */
    if(temp != NULL )
        *head_ref = temp->prev;
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginning of the Doubly Linked List */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data */
    new_node->data = new_data;

    /* since we are adding at the beginning,
    prev is always NULL */
    new_node->prev = NULL;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* change prev of head node to new node */
    if((*head_ref) != NULL)
        (*head_ref)->prev = new_node ;

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Function to print nodes in a given doubly linked list
This function is same as printList() of singly linked list */
void printList(struct node *node)
{
    while(node!=NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Driver program to test above functions */
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

    /* Let us create a sorted linked list to test the functions
    Created linked list will be 10->8->4->2 */
    push(&head, 2);
    push(&head, 4);
    push(&head, 8);
    push(&head, 10);

    printf("\n Original Linked list ");
    printList(head);

    /* Reverse doubly linked list */
    reverse(&head);

    printf("\n Reversed Linked list ");
    printList(head);

    getchar();
}

```

Java

```

// Java program to reverse a doubly linked list

class LinkedList {

    static Node head;

    static class Node {

        int data;
        Node next, prev;

        Node(int d) {
            data = d;
        }
    }
}

```

```

        next = prev = null;
    }
}

/* Function to reverse a Doubly Linked List */
void reverse() {
    Node temp = null;
    Node current = head;

    /* swap next and prev for all nodes of
    doubly linked list */
    while (current != null) {
        temp = current.prev;
        current.prev = current.next;
        current.next = temp;
        current = current.prev;
    }

    /* Before changing head, check for the cases like empty
    list and list with only one node */
    if (temp != null) {
        head = temp.prev;
    }
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginning of the Doubly Linked List */
void push(int new_data) {
    /* allocate node */
    Node new_node = new Node(new_data);

    /* since we are adding at the beginning,
    prev is always NULL */
    new_node.prev = null;

    /* link the old list off the new node */
    new_node.next = head;

    /* change prev of head node to new node */
    if (head != null) {
        head.prev = new_node;
    }

    /* move the head to point to the new node */
    head = new_node;
}

/* Function to print nodes in a given doubly linked list
This function is same as printList() of singly linked list */
void printList(Node node) {
    while (node != null) {
        System.out.print(node.data + " ");
        node = node.next;
    }
}

public static void main(String[] args) {
    LinkedList list = new LinkedList();

    /* Let us create a sorted linked list to test the functions
    Created linked list will be 10->8->4->2 */
    list.push(2);
    list.push(4);
    list.push(8);
    list.push(10);

    System.out.println("Original linked list ");
    list.printList(head);

    list.reverse();
    System.out.println("");
    System.out.println("The reversed Linked List is ");
    list.printList(head);
}
}

// This code has been contributed by Mayank Jaiswal

```

Python

```

# Program to reverse a doubly linked list

# A node of the doubly linked list
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:
    # Constructor for empty Doubly Linked List
    def __init__(self):
        self.head = None

    # Function reverse a Doubly Linked List
    def reverse(self):
        temp = None
        current = self.head

        # Swap next and prev for all nodes of
        # doubly linked list
        while current is not None:
            temp = current.prev
            current.prev = current.next
            current.next = temp
            current = current.prev

    # Before changing head, check for the cases like
    # empty list and list with only one node

```

```

if temp is not None:
    self.head = temp.prev

# Given a reference to the head of a list and an
# integer, inserts a new node on the front of list
def push(self, new_data):

    # 1. Allocates node
    # 2. Put the data in it
    new_node = Node(new_data)

    # 3. Make next of new node as head and
    # previous as None (already None)
    new_node.next = self.head

    # 4. change prev of head node to new_node
    if self.head is not None:
        self.head.prev = new_node

    # 5. move the head to point to the new node
    self.head = new_node

def printList(self, node):
    while (node is not None):
        print node.data,
        node = node.next

# Driver program to test the above functions
dll = DoublyLinkedList()
dll.push(2);
dll.push(4);
dll.push(8);
dll.push(10);

print "nOriginal Linked List"
dll.printList(dll.head)

# Reverse doubly linked list
dll.reverse()

print "n Reversed Linked List"
dll.printList(dll.head)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Time Complexity: O(n)

We can also swap data instead of pointers to reverse the Doubly Linked List. [Method used for reversing array](#) can be used to swap data. Swapping data can be costly compared to pointers if size of data item(s) is more.

Please write comments if you find any of the above codes/algorithms incorrect, or find better ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

[Linked Lists](#)

The Great Tree-List Recursion Problem.

Asked by Varun Bhatia.

Question:

Write a recursive function `treeToList(Node root)` that takes an ordered binary tree and rearranges the internal pointers to make a circular doubly linked list out of the tree nodes. The "previous" pointers should be stored in the "small" field and the "next" pointers should be stored in the "large" field. The list should be arranged so that the nodes are in increasing order. Return the head pointer to the new list.

This is very well explained and implemented at: [Convert a Binary Tree to a Circular Doubly Link List](#)

References:

<http://cslibrary.stanford.edu/109/TreeListRecursion.html>

GATE CS Corner Company Wise Coding Practice

[Linked Lists](#)
[Trees](#)

QuickSort on Doubly Linked List

Following is a typical recursive implementation of [QuickSort](#) for arrays. The implementation uses last element as pivot.

```

/* A typical recursive implementation of Quicksort for array */

/* This function takes last element as pivot, places the pivot element at its
correct position in sorted array, and places all smaller (smaller than
pivot) to left of pivot and all greater elements to right of pivot */
int partition (int arr[], int l, int h)
{
    int x = arr[h];
    int i = (l - 1);

    for (int j = l; j <= h - 1; j++)
    {
        if (arr[j] <= x)
        {
            i++;
            swap (&arr[i], &arr[j]);
        }
    }
    swap (&arr[i + 1], &arr[h]);
    return (i + 1);
}

/* A[] --> Array to be sorted, l --> Starting index, h --> Ending index */
void quickSort(int A[], int l, int h)
{
    if (l < h)
    {
        int p = partition(A, l, h); /* Partitioning index */
        quickSort(A, l, p - 1);
        quickSort(A, p + 1, h);
    }
}

```

```
}
```

Can we use same algorithm for Linked List?

Following is C++ implementation for doubly linked list. The idea is simple, we first find out pointer to last node. Once we have pointer to last node, we can recursively sort the linked list using pointers to first and last nodes of linked list, similar to the above recursive function where we pass indexes of first and last array elements. The partition function for linked list is also similar to partition for arrays. Instead of returning index of the pivot element, it returns pointer to the pivot element. In the following implementation, quickSort() is just a wrapper function, the main recursive function is _quickSort() which is similar to quickSort() for array implementation.



C++

```
// A C++ program to sort a linked list using Quicksort
#include <iostream>
#include <stdio.h>
using namespace std;

/* a node of the doubly linked list */
struct node
{
    int data;
    struct node *next;
    struct node *prev;
};

/* A utility function to swap two elements */
void swap ( int* a, int* b )
{
    int t = *a;    *a = *b;    *b = t; }

// A utility function to find last node of linked list
struct node *lastNode(struct node *root)
{
    while (root && root->next)
        root = root->next;
    return root;
}

/* Considers last element as pivot, places the pivot element at its
correct position in sorted array, and places all smaller (smaller than
pivot) to left of pivot and all greater elements to right of pivot */
node* partition(struct node *l, struct node *h)
{
    // set pivot as h element
    int x = h->data;

    // similar to i = l-1 for array implementation
    struct node *i = l->prev;

    // Similar to "for (int j = l; j <= h-1; j++)"
    for (struct node *j = l; j != h; j = j->next)
    {
        if (j->data <= x)
        {
            // Similar to i++ for array
            i = (i == NULL)? l : i->next;

            swap(&(i->data), &(j->data));
        }
    }
    i = (i == NULL)? l : i->next; // Similar to i++
    swap(&(i->data), &(h->data));
    return i;
}

/* A recursive implementation of quicksort for linked list */
void _quickSort(struct node * l, struct node *h)
{
    if (h != NULL && l != h && l != h->next)
    {
        struct node *p = partition(l, h);
        _quickSort(l, p->prev);
        _quickSort(p->next, h);
    }
}

// The main function to sort a linked list. It mainly calls _quickSort()
void quickSort(struct node *head)
{
    // Find last node
    struct node *h = lastNode(head);

    // Call the recursive QuickSort
    _quickSort(head, h);
}

// A utility function to print contents of arr
void printList(struct node *head)
{
    while (head)
    {
        cout << head->data << " ";
        head = head->next;
    }
    cout << endl;
}

/* Function to insert a node at the beginning of the Doubly Linked List */
void push(struct node** head_ref, int new_data)
{
    struct node* new_node = new node; /* allocate node */
    new_node->data = new_data;

    /* since we are adding at the beginning, prev is always NULL */
    new_node->prev = NULL;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* change prev of head node to new node */
    if ((*head_ref) != NULL) (*head_ref)->prev = new_node ;
}
```

```

/* move the head to point to the new node */
(*head_ref) = new_node;
}

/* Driver program to test above function */
int main()
{
    struct node *a = NULL;
    push(&a, 5);
    push(&a, 20);
    push(&a, 4);
    push(&a, 3);
    push(&a, 30);

    cout << "Linked List before sorting \n";
    printList(a);

    quickSort(a);

    cout << "Linked List after sorting \n";
    printList(a);

    return 0;
}

```

Java

```

// A Java program to sort a linked list using Quicksort
class QuickSort_using_Doubly_LinkedList{
    Node head;

/* a node of the doubly linked list */
    static class Node{
        private int data;
        private Node next;
        private Node prev;

        Node(int d){
            data = d;
            next = null;
            prev = null;
        }
    }

// A utility function to find last node of linked list
    Node lastNode(Node node){
        while(node.next!=null)
            node = node.next;
        return node;
    }

/* Considers last element as pivot, places the pivot element at its
correct position in sorted array, and places all smaller (smaller than
pivot) to left of pivot and all greater elements to right of pivot */
    Node partition(Node l, Node h)
    {
        // set pivot as h element
        int x = h.data;

        // similar to i = l-1 for array implementation
        Node i = l.prev;

        // Similar to "for (int j = l; j <= h-1; j++)"
        for(Node j=l; j!=h; j=j.next)
        {
            if(j.data <= x)
            {
                // Similar to i++ for array
                i = (i==null) ? l : i.next;
                int temp = i.data;
                i.data = j.data;
                j.data = temp;
            }
        }
        i = (i==null) ? l : i.next; // Similar to i++
        int temp = i.data;
        i.data = h.data;
        h.data = temp;
        return i;
    }

/* A recursive implementation of quicksort for linked list */
    void _quickSort(Node l, Node h)
    {
        if(h!=null && l!=h && l!=h.next){
            Node temp = partition(l, h);
            _quickSort(l, temp.prev);
            _quickSort(temp.next, h);
        }
    }

// The main function to sort a linked list. It mainly calls _quickSort()
    public void quickSort(Node node)
    {
        // Find last node
        Node head = lastNode(node);

        // Call the recursive QuickSort
        _quickSort(node, head);
    }

// A utility function to print contents of arr
    public void printList(Node head)
    {
        while(head!=null){
            System.out.print(head.data+" ");
            head = head.next;
        }
    }

/* Function to insert a node at the beginning of the Doubly Linked List */

```

```

void push(int new_Data)
{
    Node new_Node = new Node(new_Data); /* allocate node */

    // if head is null, head = new_Node
    if(head==null){
        head = new_Node;
        return;
    }

    /* link the old list off the new node */
    new_Node.next = head;

    /* change prev of head node to new node */
    head.prev = new_Node;

    /* since we are adding at the beginning, prev is always NULL */
    new_Node.prev = null;

    /* move the head to point to the new node */
    head = new_Node;
}

/* Driver program to test above function */
public static void main(String[] args){
    QuickSort_using_Doubly_LinkedList list = new QuickSort_using_Doubly_LinkedList();

    list.push(5);
    list.push(20);
    list.push(4);
    list.push(3);
    list.push(30);

    System.out.println("Linked List before sorting ");
    list.printList(list.head);
    System.out.println("\nLinked List after sorting");
    list.quickSort(list.head);
    list.printList(list.head);
}
}

// This code has been contributed by Amit Khandelwal

```

Output :

```

Linked List before sorting
30 3 4 20 5
Linked List after sorting
3 4 5 20 30

```

Time Complexity: Time complexity of the above implementation is same as time complexity of QuickSort() for arrays. It takes $O(n^2)$ time in worst case and $O(n \log n)$ in average and best cases. The worst case occurs when the linked list is already sorted.

Can we implement random quick sort for linked list?

QuickSort can be implemented for Linked List only when we can pick a fixed point as pivot (like last element in above implementation). Random QuickSort cannot be efficiently implemented for Linked Lists by picking random pivot.

Exercise:

The above implementation is for doubly linked list. Modify it for singly linked list. Note that we don't have prev pointer in singly linked list.

Refer [QuickSort on Singly Linked List](#) for solution.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Linked Lists
Sorting
Quick Sort

Merge Sort for Doubly Linked List

Given a doubly linked list, write a function to sort the doubly linked list in increasing order using merge sort.

For example, the following doubly linked list should be changed to 24810



We strongly recommend to minimize your browser and try this yourself first.

[Merge sort for singly linked list](#) is already discussed. The important change here is to modify the previous pointers also when merging two lists.

Below is the implementation of merge sort for doubly linked list.

C

```

// C program for merge sort on doubly linked list
#include<stdio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node *next, *prev;
};

struct node *split(struct node *head);

// Function to merge two linked lists
struct node *merge(struct node *first, struct node *second)
{
    // If first linked list is empty
    if (!first)
        return second;

    // If second linked list is empty

```

```

if (second)
    return first;

// Pick the smaller value
if (first->data < second->data)
{
    first->next = merge(first->next,second);
    first->next->prev = first;
    first->prev = NULL;
    return first;
}
else
{
    second->next = merge(first,second->next);
    second->next->prev = second;
    second->prev = NULL;
    return second;
}
}

// Function to do merge sort
struct node *mergeSort(struct node *head)
{
    if (!head || !head->next)
        return head;
    struct node *second = split(head);

    // Recur for left and right halves
    head = mergeSort(head);
    second = mergeSort(second);

    // Merge the two sorted halves
    return merge(head,second);
}

// A utility function to insert a new node at the
// beginning of doubly linked list
void insert(struct node **head, int data)
{
    struct node *temp =
        (struct node *)malloc(sizeof(struct node));
    temp->data = data;
    temp->next = temp->prev = NULL;
    if (!(*head))
        (*head) = temp;
    else
    {
        temp->next = *head;
        (*head)->prev = temp;
        (*head) = temp;
    }
}

// A utility function to print a doubly linked list in
// both forward and backward directions
void print(struct node *head)
{
    struct node *temp = head;
    printf("Forward Traversal using next pointer\n");
    while (head)
    {
        printf("%d ",head->data);
        temp = head;
        head = head->next;
    }
    printf("\nBackward Traversal using prev pointer\n");
    while (temp)
    {
        printf("%d ", temp->data);
        temp = temp->prev;
    }
}

// Utility function to swap two integers
void swap(int *A, int *B)
{
    int temp = *A;
    *A = *B;
    *B = temp;
}

// Split a doubly linked list (DLL) into 2 DLLs of
// half sizes
struct node *split(struct node *head)
{
    struct node *fast = head,*slow = head;
    while (fast->next && fast->next->next)
    {
        fast = fast->next->next;
        slow = slow->next;
    }
    struct node *temp = slow->next;
    slow->next = NULL;
    return temp;
}

// Driver program
int main(void)
{
    struct node *head = NULL;
    insert(&head,5);
    insert(&head,20);
    insert(&head,4);
    insert(&head,3);
    insert(&head,30);
    insert(&head,10);
    head = mergeSort(head);
    printf("\n\nLinked List after sorting\n");
    print(head);
    return 0;
}

```

```
// Java program to implement merge sort in singly linked list
```

```
// Linked List Class
```

```
class LinkedList {
```

```
    static Node head; // head of list
```

```
    /* Node Class */
```

```
    static class Node {
```

```
        int data;
```

```
        Node next, prev;
```

```
        // Constructor to create a new node
```

```
        Node(int d) {
```

```
            data = d;
```

```
            next = prev = null;
```

```
        }
```

```
    }
```

```
    void print(Node node) {
```

```
        Node temp = node;
```

```
        System.out.println("Forward Traversal using next pointer");
```

```
        while (node != null) {
```

```
            System.out.print(node.data + " ");
```

```
            temp = node;
```

```
            node = node.next;
```

```
        }
```

```
        System.out.println("\nBackward Traversal using prev pointer");
```

```
        while (temp != null) {
```

```
            System.out.print(temp.data + " ");
```

```
            temp = temp.prev;
```

```
        }
```

```
    }
```

```
    // Split a doubly linked list (DLL) into 2 DLLs of
```

```
    // half sizes
```

```
    Node split(Node head) {
```

```
        Node fast = head, slow = head;
```

```
        while (fast.next != null && fast.next.next != null) {
```

```
            fast = fast.next.next;
```

```
            slow = slow.next;
```

```
        }
```

```
        Node temp = slow.next;
```

```
        slow.next = null;
```

```
        return temp;
```

```
    }
```

```
    Node mergeSort(Node node) {
```

```
        if (node == null || node.next == null) {
```

```
            return node;
```

```
        }
```

```
        Node second = split(node);
```

```
        // Recur for left and right halves
```

```
        node = mergeSort(node);
```

```
        second = mergeSort(second);
```

```
        // Merge the two sorted halves
```

```
        return merge(node, second);
```

```
    }
```

```
    // Function to merge two linked lists
```

```
    Node merge(Node first, Node second) {
```

```
        // If first linked list is empty
```

```
        if (first == null) {
```

```
            return second;
```

```
        }
```

```
        // If second linked list is empty
```

```
        if (second == null) {
```

```
            return first;
```

```
        }
```

```
        // Pick the smaller value
```

```
        if (first.data < second.data) {
```

```
            first.next = merge(first.next, second);
```

```
            first.next.prev = first;
```

```
            first.prev = null;
```

```
            return first;
```

```
        } else {
```

```
            second.next = merge(first, second.next);
```

```
            second.next.prev = second;
```

```
            second.prev = null;
```

```
            return second;
```

```
        }
```

```
    }
```

```
    // Driver program to test above functions
```

```
    public static void main(String[] args) {
```

```
        LinkedList list = new LinkedList();
```

```
        list.head = new Node(10);
```

```
        list.head.next = new Node(30);
```

```
        list.head.next.next = new Node(3);
```

```
        list.head.next.next.next = new Node(4);
```

```
        list.head.next.next.next.next = new Node(20);
```

```
        list.head.next.next.next.next.next = new Node(5);
```

```
        Node node = null;
```

```
        node = list.mergeSort(head);
```

```
        System.out.println("Linked list after sorting -");
```

```
        list.print(node);
```

```
    }
```

```
}
```

```
// This code has been contributed by Mayank Jaiswal
```



```

# Program for merge sort on doubly linked list

# A node of the doubly linked list
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:

    # Constructor for empty Doubly Linked List
    def __init__(self):
        self.head = None

    # Function to merge two linked list
    def merge(self, first, second):

        # If first linked list is empty
        if first is None:
            return second

        # If second linked list is empty
        if second is None:
            return first

        # Pick the smaller value
        if first.data < second.data:
            first.next = self.merge(first.next, second)
            first.next.prev = first
            first.prev = None
            return first
        else:
            second.next = self.merge(first, second.next)
            second.next.prev = second
            second.prev = None
            return second

    # Function to do merge sort
    def mergeSort(self, tempHead):
        if tempHead is None:
            return tempHead
        if tempHead.next is None:
            return tempHead

        second = self.split(tempHead)

        # Recur for left and right halves
        tempHead = self.mergeSort(tempHead)
        second = self.mergeSort(second)

        # Merge the two sorted halves
        return self.merge(tempHead, second)

    # Split the doubly linked list (DLL) into two DLLs
    # of half sizes
    def split(self, tempHead):
        fast = slow = tempHead
        while(True):
            if fast.next is None:
                break
            if fast.next.next is None:
                break
            fast = fast.next.next
            slow = slow.next

        temp = slow.next
        slow.next = None
        return temp

    # Given a reference to the head of a list and an
    # integer, inserts a new node on the front of list
    def push(self, new_data):

        # 1. Allocates node
        # 2. Put the data in it
        new_node = Node(new_data)

        # 3. Make next of new node as head and
        # previous as None (already None)
        new_node.next = self.head

        # 4. change prev of head node to new_node
        if self.head is not None:
            self.head.prev = new_node

        # 5. move the head to point to the new node
        self.head = new_node

    def printList(self, node):
        temp = node
        print "Forward Traversal using next pointer"
        while(node is not None):
            print node.data,
            temp = node
            node = node.next
        print "\nBackward Traversal using prev pointer"
        while(temp):
            print temp.data,
            temp = temp.prev

# Driver program to test the above functions
dll = DoublyLinkedList()
dll.push(5)
dll.push(20);
dll.push(4);
dll.push(3);
dll.push(30)
dll.push(10);
dll.head = dll.mergeSort(dll.head)

```

```
print "Linked List after sorting"
dll.printList(dll.head)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
Linked List after sorting
Forward Traversal using next pointer
3 4 5 10 20 30
Backward Traversal using prev pointer
30 20 10 5 4 3
```

Thanks to Goku for providing above implementation in a comment [here](#).

Time Complexity: Time complexity of the above implementation is same as time complexity of [MergeSort for arrays](#). It takes $\Theta(n \log n)$ time.

You may also like to see [QuickSort for doubly linked list](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

[Linked Lists](#)
[Sorting](#)
[Merge Sort](#)

Stack Data Structure

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO (Last In First Out) or FILO (First In Last Out).

Mainly the following three basic operations are performed in the stack:

- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- **Peek or Top:** Returns top element of stack.
- **isEmpty:** Returns true if stack is empty, else false.

stack



How to understand a stack practically?

There are many real life examples of stack. Consider the simple example of plates stacked over one another in canteen. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO/FILO order.

Time Complexities of operations on stack:

push(), pop(), isEmpty() and peek() all take $O(1)$ time. We do not run any loop in any of these operations.

Applications of stack:

- [Balancing of symbols](#)
- [Infix to Postfix /Prefix conversion](#)
- Redo-undo features at many places like editors, photoshop.
- Forward and backward feature in web browsers
- Used in many algorithms like [Tower of Hanoi](#), [tree traversals](#), [stock span problem](#), [histogram problem](#).
- Other applications can be [Backtracking](#), [Knight tour problem](#), [rat in a maze](#), [N queen problem](#) and [sudoku solver](#)

Implementation:

There are two ways to implement a stack:

- Using array
- Using linked list

Implementing Stack using Arrays

C++

```
/* C++ program to implement basic stack
operations */
#include<bits/stdc++.h>
using namespace std;

#define MAX 1000

class Stack
{
    int top;
public:
    int a[MAX]; //Maximum size of Stack

    Stack() { top = -1; }
    bool push(int x);
    int pop();
    bool isEmpty();
};

bool Stack::push(int x)
{
    if (top >= MAX)
    {
        cout << "Stack Overflow";
        return false;
    }
    else
```

```

    {
        a[++top] = x;
        return true;
    }
}

int Stack::pop()
{
    if (top < 0)
    {
        cout << "Stack Underflow";
        return 0;
    }
    else
    {
        int x = a[top--];
        return x;
    }
}

bool Stack::isEmpty()
{
    return (top < 0);
}

// Driver program to test above functions
int main()
{
    struct Stack s;
    s.push(10);
    s.push(20);
    s.push(30);

    cout << s.pop() << " Popped from stack\n";

    return 0;
}

```

Java

```

/* Java program to implement basic stack
operations */
class Stack
{
    static final int MAX = 1000;
    int top;
    int a[] = new int[MAX]; // Maximum size of Stack

    boolean isEmpty()
    {
        return (top < 0);
    }
    Stack()
    {
        top = -1;
    }

    boolean push(int x)
    {
        if (top >= MAX)
        {
            System.out.println("Stack Overflow");
            return false;
        }
        else
        {
            a[++top] = x;
            return true;
        }
    }

    int pop()
    {
        if (top < 0)
        {
            System.out.println("Stack Underflow");
            return 0;
        }
        else
        {
            int x = a[top--];
            return x;
        }
    }
}

// Driver code
class Main
{
    public static void main(String args[])
    {
        Stack s = new Stack();
        s.push(10);
        s.push(20);
        s.push(30);
        System.out.println(s.pop() + " Popped from stack");
    }
}

```

C

```

// C program for array implementation of stack
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

// A structure to represent a stack
struct Stack
{
    int top;
    unsigned capacity;

```

```

int* array;
};

// function to create a stack of given capacity. It initializes size of
// stack as 0
struct Stack* createStack(unsigned capacity)
{
    struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));
    stack->capacity = capacity;
    stack->top = -1;
    stack->array = (int*) malloc(stack->capacity * sizeof(int));
    return stack;
}

// Stack is full when top is equal to the last index
int isFull(struct Stack* stack)
{ return stack->top == stack->capacity - 1; }

// Stack is empty when top is equal to -1
int isEmpty(struct Stack* stack)
{ return stack->top == -1; }

// Function to add an item to stack. It increases top by 1
void push(struct Stack* stack, int item)
{
    if (isFull(stack))
        return;
    stack->array[++stack->top] = item;
    printf("%d pushed to stack\n", item);
}

// Function to remove an item from stack. It decreases top by 1
int pop(struct Stack* stack)
{
    if (isEmpty(stack))
        return INT_MIN;
    return stack->array[stack->top--];
}

// Driver program to test above functions
int main()
{
    struct Stack* stack = createStack(100);

    push(stack, 10);
    push(stack, 20);
    push(stack, 30);

    printf("%d popped from stack\n", pop(stack));

    return 0;
}

```

Python

```

# Python program for implementation of stack

# import maxsize from sys module
# Used to return -infinite when stack is empty
from sys import maxsize

# Function to create a stack. It initializes size of stack as 0
def createStack():
    stack = []
    return stack

# Stack is empty when stack size is 0
def isEmpty(stack):
    return len(stack) == 0

# Function to add an item to stack. It increases size by 1
def push(stack, item):
    stack.append(item)
    print("pushed to stack " + item)

# Function to remove an item from stack. It decreases size by 1
def pop(stack):
    if (isEmpty(stack)):
        return str(-maxsize - 1) #return minus infinite

    return stack.pop()

# Driver program to test above functions
stack = createStack()
push(stack, str(10))
push(stack, str(20))
push(stack, str(30))
print(pop(stack) + " popped from stack")

```

Pros: Easy to implement. Memory is saved as pointers are not involved.

Cons: It is not dynamic. It doesn't grow and shrink depending on needs at runtime.

```

10 pushed to stack
20 pushed to stack
30 pushed to stack
30 popped from stack
Top item is 20

```

Implementing Stack using Linked List

C

```

// C program for linked list implementation of stack
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

// A structure to represent a stack
struct StackNode
{
    int data;

```

```

    struct StackNode* next;
};

struct StackNode* newNode(int data)
{
    struct StackNode* stackNode =
        (struct StackNode*) malloc(sizeof(struct StackNode));
    stackNode->data = data;
    stackNode->next = NULL;
    return stackNode;
}

int isEmpty(struct StackNode *root)
{
    return !root;
}

void push(struct StackNode** root, int data)
{
    struct StackNode* stackNode = newNode(data);
    stackNode->next = *root;
    *root = stackNode;
    printf("%d pushed to stack\n", data);
}

int pop(struct StackNode** root)
{
    if (isEmpty(*root))
        return INT_MIN;
    struct StackNode* temp = *root;
    *root = (*root)->next;
    int popped = temp->data;
    free(temp);

    return popped;
}

int peek(struct StackNode* root)
{
    if (isEmpty(root))
        return INT_MIN;
    return root->data;
}

int main()
{
    struct StackNode* root = NULL;

    push(&root, 10);
    push(&root, 20);
    push(&root, 30);

    printf("%d popped from stack\n", pop(&root));

    printf("Top element is %d\n", peek(root));

    return 0;
}

```

Python

```

# Python program for linked list implementation of stack

# Class to represent a node
class StackNode:

    # Constructor to initialize a node
    def __init__(self, data):
        self.data = data
        self.next = None

class Stack:

    # Constructor to initialize the root of linked list
    def __init__(self):
        self.root = None

    def isEmpty(self):
        return True if self.root is None else False

    def push(self, data):
        newNode = StackNode(data)
        newNode.next = self.root
        self.root = newNode
        print "%d pushed to stack" %(data)

    def pop(self):
        if (self.isEmpty()):
            return float("-inf")
        temp = self.root
        self.root = self.root.next
        popped = temp.data
        return popped

    def peek(self):
        if self.isEmpty():
            return float("-inf")
        return self.root.data

# Driver program to test above class
stack = Stack()
stack.push(10)
stack.push(20)
stack.push(30)

print "%d popped from stack" %(stack.pop())
print "Top element is %d " %(stack.peek())

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

```
10 pushed to stack
20 pushed to stack
30 pushed to stack
30 popped from stack
Top element is 20
```

Pros: The linked list implementation of stack can grow and shrink according to the needs at runtime.

Cons: Requires extra memory due to involvement of pointers.

We will cover the implementation of applications of stack in separate posts.

[Stack Set -2 \(Infix to Postfix\)](#)

Quiz: [Stack Questions](#)

References:

http://en.wikipedia.org/wiki/Stack_%28abstract_data_type%29#Problem_Description

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: Stack

Stack | Set 2 (Infix to Postfix)

Prerequisite – [Stack | Set 1 \(Introduction\)](#)

Infix expression: The expression of the form a op b. When an operator is in-between every pair of operands.

Postfix expression: The expression of the form a b op. When an operator is followed for every pair of operands.

Why postfix representation of the expression?

The compiler scans the expression either from left to right or from right to left.

Consider the below expression: a op1 b op2 c op3 d

If op1 = +, op2 = *, op3 = +

The compiler first scans the expression to evaluate the expression b * c, then again scan the expression to add a to it. The result is then added to d after another scan.

The repeated scanning makes it very in-efficient. It is better to convert the expression to postfix(or prefix) form before evaluation.

The corresponding expression in postfix form is: abc*+d+. The postfix expressions can be evaluated easily using a stack. We will cover postfix expression evaluation in a separate post.

Algorithm

1. Scan the infix expression from left to right.

2. If the scanned character is an operand, output it.

3. Else,

.....3.1 If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty), push it.

.....3.2 Else, Pop the operator from the stack until the precedence of the scanned operator is less-equal to the precedence of the operator residing on the top of the stack. Push the scanned operator to the stack.

4. If the scanned character is an '(', push it to the stack.

5. If the scanned character is an ')', pop and output from the stack until an '(' is encountered.

6. Repeat steps 2-6 until infix expression is scanned.

7. Pop and output from the stack until it is not empty.

Following is C implementation of the above algorithm

C

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

// Stack type
struct Stack
{
    int top;
    unsigned capacity;
    int* array;
};

// Stack Operations
struct Stack* createStack( unsigned capacity )
{
    struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));

    if (!stack)
        return NULL;

    stack->top = -1;
    stack->capacity = capacity;
```

```

stack->array = (int*) malloc(stack->capacity * sizeof(int));

if (!stack->array)
    return NULL;
return stack;
}

int isEmpty(struct Stack* stack)
{
    return stack->top == -1 ;
}

char peek(struct Stack* stack)
{
    return stack->array[stack->top];
}

char pop(struct Stack* stack)
{
    if (!isEmpty(stack))
        return stack->array[stack->top--];
    return '$';
}

void push(struct Stack* stack, char op)
{
    stack->array[++stack->top] = op;
}

// A utility function to check if the given character is operand
int isOperand(char ch)
{
    return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z');
}

// A utility function to return precedence of a given operator
// Higher returned value means higher precedence
int Prec(char ch)
{
    switch (ch)
    {
        case '+':
        case '-':
            return 1;

        case '*':
        case '/':
            return 2;

        case '^':
            return 3;
    }
    return -1;
}

// The main function that converts given infix expression
// to postfix expression.
int infixToPostfix(char* exp)
{
    int i, k;

    // Create a stack of capacity equal to expression size
    struct Stack* stack = createStack(strlen(exp));
    if(!stack) // See if stack was created successfully
        return -1 ;

    for (i = 0, k = -1; exp[i]; ++i)
    {
        // If the scanned character is an operand, add it to output.
        if (isOperand(exp[i]))
            exp[++k] = exp[i];

        // If the scanned character is an '(', push it to the stack.
        else if (exp[i] == '(')
            push(stack, exp[i]);

        // If the scanned character is an ')', pop and output from the stack
        // until an '(' is encountered.
        else if (exp[i] == ')')
        {
            while (!isEmpty(stack) && peek(stack) != '(')
                exp[++k] = pop(stack);
            if (!isEmpty(stack) && peek(stack) != '(')
                return -1; // invalid expression
            else
                pop(stack);
        }
        else // an operator is encountered
        {
            while (!isEmpty(stack) && Prec(exp[i]) <= Prec(peek(stack)))
                exp[++k] = pop(stack);
            push(stack, exp[i]);
        }
    }

    // pop all the operators from the stack
    while (!isEmpty(stack))
        exp[++k] = pop(stack);

    exp[++k] = '\0';
    printf( "%s\n", exp );
}

// Driver program to test above functions
int main()
{
    char exp[] = "a+b*(c^d-e)^(f+g*h)-i";
    infixToPostfix(exp);
    return 0;
}

```

Python

```
# Python program to convert infix expression to postfix
```

```
# Class to convert the expression
class Conversion:
```

```
    # Constructor to initialize the class variables
    def __init__(self, capacity):
        self.top = -1
        self.capacity = capacity
        # This array is used as a stack
        self.array = []
        # Precedence setting
        self.output = []
        self.precedence = {'+':1, '-':1, '*':2, '/':2, '^':3}
```

```
    # check if the stack is empty
    def isEmpty(self):
        return True if self.top == -1 else False
```

```
    # Return the value of the top of the stack
    def peek(self):
        return self.array[-1]
```

```
    # Pop the element from the stack
    def pop(self):
        if not self.isEmpty():
            self.top -= 1
            return self.array.pop()
        else:
            return "$"
```

```
    # Push the element to the stack
    def push(self, op):
        self.top += 1
        self.array.append(op)
```

```
    # A utility function to check if the given character
    # is operand
    def isOperand(self, ch):
        return ch.isalpha()
```

```
    # Check if the precedence of operator is strictly
    # less than top of stack or not
    def notGreater(self, i):
        try:
            a = self.precedence[i]
            b = self.precedence[self.peek()]
            return True if a <= b else False
        except KeyError:
            return False
```

```
    # The main function that converts given infix expression
    # to postfix expression
    def infixToPostfix(self, exp):
```

```
        # Iterate over the expression for conversion
        for i in exp:
            # If the character is an operand,
            # add it to output
            if self.isOperand(i):
                self.output.append(i)
```

```
            # If the character is an '(', push it to stack
            elif i == '(':
                self.push(i)
```

```
            # If the scanned character is an ')', pop and
            # output from the stack until and '(' is found
            elif i == ')':
                while (not self.isEmpty() and self.peek() != '('):
                    a = self.pop()
                    self.output.append(a)
                if (not self.isEmpty() and self.peek() != '('):
                    return -1
                else:
                    self.pop()
```

```
            # An operator is encountered
            else:
                while(not self.isEmpty() and self.notGreater(i)):
                    self.output.append(self.pop())
                self.push(i)
```

```
        # pop all the operator from the stack
        while not self.isEmpty():
            self.output.append(self.pop())
```

```
        print "".join(self.output)
```

```
# Driver program to test above function
```

```
exp = "a+b*(c^d-e)^(f+g*h)-i"
obj = Conversion(len(exp))
obj.infixToPostfix(exp)
```

```
# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
abcd^e-fgh*+^*-i-
```


Quiz: Stack Questions

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: [Stack](#)

Stack | Set 4 (Evaluation of Postfix Expression)

The Postfix notation is used to represent algebraic expressions. The expressions written in postfix form are evaluated faster compared to infix notation as parenthesis are not required in postfix. We have discussed [infix to postfix conversion](#). In this post, evaluation of postfix expressions is discussed.

Following is algorithm for evaluation postfix expressions.

- 1) Create a stack to store operands (or values).
- 2) Scan the given expression and do following for every scanned element.
.....a) If the element is a number, push it into the stack
.....b) If the element is an operator, pop operands for the operator from stack. Evaluate the operator and push the result back to the stack
- 3) When the expression is ended, the number in the stack is the final answer

Example:

Let the given expression be "2 3 1 * + 9 -". We scan all elements one by one.

- 1) Scan '2', it's a number, so push it to stack. Stack contains '2'
- 2) Scan '3', again a number, push it to stack, stack now contains '2 3' (from bottom to top)
- 3) Scan '1', again a number, push it to stack, stack now contains '2 3 1'
- 4) Scan '*', it's an operator, pop two operands from stack, apply the * operator on operands, we get 3*1 which results in 3. We push the result '3' to stack. Stack now becomes '2 3'.
- 5) Scan '+', it's an operator, pop two operands from stack, apply the + operator on operands, we get 3 + 2 which results in 5. We push the result '5' to stack. Stack now becomes '5'.
- 6) Scan '9', it's a number, we push it to the stack. Stack now becomes '5 9'.
- 7) Scan '-', it's an operator, pop two operands from stack, apply the - operator on operands, we get 5 - 9 which results in -4. We push the result '-4' to stack. Stack now becomes '-4'.
- 8) There are no more elements to scan, we return the top element from stack (which is the only element left in stack).

Following is C implementation of above algorithm.

C

```
// C program to evaluate value of a postfix expression
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>

// Stack type
struct Stack
{
    int top;
    unsigned capacity;
    int* array;
};

// Stack Operations
struct Stack* createStack( unsigned capacity )
{
    struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));

    if (!stack) return NULL;

    stack->top = -1;
    stack->capacity = capacity;
    stack->array = (int*) malloc(stack->capacity * sizeof(int));

    if (!stack->array) return NULL;

    return stack;
}

int isEmpty(struct Stack* stack)
{
    return stack->top == -1 ;
}

char peek(struct Stack* stack)
{
    return stack->array[stack->top];
}

char pop(struct Stack* stack)
{
    if (!isEmpty(stack))
        return stack->array[stack->top--];
    return '$';
}
```

```

}

void push(struct Stack* stack, char op)
{
    stack->array[++stack->top] = op;
}

// The main function that returns value of a given postfix expression
int evaluatePostfix(char* exp)
{
    // Create a stack of capacity equal to expression size
    struct Stack* stack = createStack(strlen(exp));
    int i;

    // See if stack was created successfully
    if (!stack) return -1;

    // Scan all characters one by one
    for (i = 0; exp[i]; ++i)
    {
        // If the scanned character is an operand (number here),
        // push it to the stack.
        if (isdigit(exp[i]))
            push(stack, exp[i] - '0');

        // If the scanned character is an operator, pop two
        // elements from stack apply the operator
        else
        {
            int val1 = pop(stack);
            int val2 = pop(stack);
            switch (exp[i])
            {
                case '+': push(stack, val2 + val1); break;
                case '-': push(stack, val2 - val1); break;
                case '*': push(stack, val2 * val1); break;
                case '/': push(stack, val2/val1); break;
            }
        }
    }
    return pop(stack);
}

// Driver program to test above functions
int main()
{
    char exp[] = "231*+9-";
    printf ("Value of %s is %d", exp, evaluatePostfix(exp));
    return 0;
}

```

Python

Python program to evaluate value of a postfix expression

Class to convert the expression
class Evaluate:

```

    # Constructor to initialize the class variables
    def __init__(self, capacity):
        self.top = -1
        self.capacity = capacity
        # This array is used a stack
        self.array = []

```

```

    # check if the stack is empty
    def isEmpty(self):
        return True if self.top == -1 else False

```

```

    # Return the value of the top of the stack
    def peek(self):
        return self.array[-1]

```

```

    # Pop the element from the stack
    def pop(self):
        if not self.isEmpty():
            self.top -= 1
            return self.array.pop()
        else:
            return "$"

```

```

    # Push the element to the stack
    def push(self, op):
        self.top += 1
        self.array.append(op)

```

The main function that converts given infix expression
to postfix expression
def evaluatePostfix(self, exp):

```

    # Iterate over the expression for conversion
    for i in exp:

```

```

        # If the scanned character is an operand
        # (number here) push it to the stack
        if i.isdigit():
            self.push(i)

```

```

        # If the scanned character is an operator,
        # pop two elements from stack and apply it.
        else:
            val1 = self.pop()
            val2 = self.pop()
            self.push(str(eval(val2 + i + val1)))

```

```

    return int(self.pop())

```

Driver program to test above function

```
exp = "231*+9-"  
obj = Evaluate(len(exp))  
print "Value of %s is %d" %(exp, obj.evaluatePostfix(exp))  
  
# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
Value of 231*+9- is -4
```

Time complexity of evaluation algorithm is $O(n)$ where n is number of characters in input expression.

There are following limitations of above implementation.

- 1) It supports only 4 binary operators '+', '-', '*' and '/'. It can be extended for more operators by adding more switch cases.
- 2) The allowed operands are only single digit operands. The program can be extended for multiple digits by adding a separator like space between all elements (operators and operands) of given expression.

References:

<http://www.cs.nthu.edu.tw/~wkhon/ds/ds10/tutorial/tutorial2.pdf>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: [Stack](#) Tags: [Stack](#)

Stack | Set 3 (Reverse a string using stack)

Given a string, reverse it using stack. For example "GeeksQuiz" should be converted to "ziuQskeeG".

Following is simple algorithm to reverse a string using stack.

- 1) Create an empty stack.
- 2) One by one push all characters of string to stack.
- 3) One by one pop all characters from stack and put them back to string.

Following are C and Python programs that implements above algorithm.

C

```
// C program to reverse a string using stack  
#include <stdio.h>  
#include <string.h>  
#include <stdlib.h>  
#include <limits.h>  
  
// A structure to represent a stack  
struct Stack  
{  
    int top;  
    unsigned capacity;  
    char* array;  
};  
  
// function to create a stack of given capacity. It initializes size of  
// stack as 0  
struct Stack* createStack(unsigned capacity)  
{  
    struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));  
    stack->capacity = capacity;  
    stack->top = -1;  
    stack->array = (char*) malloc(stack->capacity * sizeof(char));  
    return stack;  
}  
  
// Stack is full when top is equal to the last index  
int isFull(struct Stack* stack)  
{ return stack->top == stack->capacity - 1; }  
  
// Stack is empty when top is equal to -1  
int isEmpty(struct Stack* stack)  
{ return stack->top == -1; }  
  
// Function to add an item to stack. It increases top by 1  
void push(struct Stack* stack, char item)  
{  
    if (isFull(stack))  
        return;  
}
```

```

    stack->array[++stack->top] = item;
}

// Function to remove an item from stack. It decreases top by 1
char pop(struct Stack* stack)
{
    if (isEmpty(stack))
        return INT_MIN;
    return stack->array[stack->top--];
}

// A stack based function to reverse a string
void reverse(char str[])
{
    // Create a stack of capacity equal to length of string
    int n = strlen(str);
    struct Stack* stack = createStack(n);

    // Push all characters of string to stack
    int i;
    for (i = 0; i < n; i++)
        push(stack, str[i]);

    // Pop all characters of string and put them back to str
    for (i = 0; i < n; i++)
        str[i] = pop(stack);
}

// Driver program to test above functions
int main()
{
    char str[] = "GeeksQuiz";

    reverse(str);
    printf("Reversed string is %s", str);

    return 0;
}

```

Python

```

# Python program to reverse a string using stack

# Function to create an empty stack. It initializes size of stack as 0
def createStack():
    stack=[]
    return stack

# Function to determine the size of the stack
def size(stack):
    return len(stack)

# Stack is empty if the size is 0
def isEmpty(stack):
    if size(stack) == 0:
        return True

# Function to add an item to stack . It increases size by 1
def push(stack,item):
    stack.append(item)

#Function to remove an item from stack. It decreases size by 1
def pop(stack):
    if isEmpty(stack): return
    return stack.pop()

# A stack based function to reverse a string
def reverse(string):
    n = len(string)

    # Create a empty stack
    stack = createStack()

    # Push all characters of string to stack
    for i in range(0,n,1):
        push(stack,string[i])

    # Making the string empty since all characters are saved in stack
    string=""

    # Pop all characters of string and put them back to string
    for i in range(0,n,1):
        string+=pop(stack)

    return string

# Driver program to test above functions
string="GeeksQuiz"
string = reverse(string)
print("Reversed string is " + string)

# This code is contributed by Sunny Karira

```

Output:

```
Reversed string is ziuQskeeG
```

Time Complexity: $O(n)$ where n is number of characters in stack.

Auxiliary Space: $O(n)$ for stack.

A string can also be reversed without using any auxiliary space. Following C and Python programs to implement reverse without using stack.

C

```

// C program to reverse a string without using stack
#include <stdio.h>
#include <string.h>

// A utility function to swap two characters

```

```

void swap(char *a, char *b)
{
    char temp = *a;
    *a = *b;
    *b = temp;
}

// A stack based function to reverse a string
void reverse(char str[])
{
    // get size of string
    int n = strlen(str);

    for (i = 0; i < n/2; i++)
        swap(&str[i], &str[n-i-1]);
}

// Driver program to test above functions
int main()
{
    char str[] = "abc";

    reverse(str);
    printf("Reversed string is %s", str);

    return 0;
}

```

Python

```

# Python program to reverse a string without stack

# Function to reverse a string
def reverse(string):
    string = string[::-1]
    return string

# Driver program to test above functions
string = "abc"
string = reverse(string)
print("Reversed string is " + string)

# This code is contributed by Sunny Karira

```

Output:

```
Reversed string is cba
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: Stack

Implement two stacks in an array

Create a data structure *twoStacks* that represents two stacks. Implementation of *twoStacks* should use only one array, i.e., both stacks should use the same array for storing elements. Following functions must be supported by *twoStacks*.

```

push1(int x) -> pushes x to first stack
push2(int x) -> pushes x to second stack

```

```

pop1() -> pops an element from first stack and return the popped element
pop2() -> pops an element from second stack and return the popped element

```

Implementation of *twoStack* should be space efficient.

Method 1 (Divide the space in two halves)

A simple way to implement two stacks is to divide the array in two halves and assign the half half space to two stacks, i.e., use `arr[0]` to `arr[n/2]` for stack1, and `arr[n/2+1]` to `arr[n-1]` for stack2 where `arr[]` is the array to be used to implement two stacks and size of array be `n`.

The problem with this method is inefficient use of array space. A stack push operation may result in stack overflow even if there is space available in `arr[]`. For example, say the array size is 6 and we push 3 elements to stack1 and do not push anything to second stack2. When we push 4th element to stack1, there will be overflow even if we have space for 3 more elements in array.

Method 2 (A space efficient implementation)

This method efficiently utilizes the available space. It doesn't cause an overflow if there is space available in `arr[]`. The idea is to start two stacks from two extreme corners of `arr[]`. stack1 starts from the leftmost element, the first element in stack1 is pushed at index 0. The stack2 starts from the rightmost corner, the first element in stack2 is pushed at index (n-1). Both stacks grow (or shrink) in opposite direction. To check for overflow, all we need to check is for space between top elements of both stacks. This check is highlighted in the below code.

C++

```
#include<iostream>
#include<stdlib.h>

using namespace std;

class twoStacks
{
    int *arr;
    int size;
    int top1 , top2;
public:
    twoStacks(int n) // constructor
    {
        size = n;
        arr = new int[n];
        top1 = -1;
        top2 = size;
    }

    // Method to push an element x to stack1
    void push1(int x)
    {
        // There is at least one empty space for new element
        if (top1 < top2 - 1)
        {
            top1++;
            arr[top1] = x;
        }
        else
        {
            cout << "Stack Overflow";
            exit(1);
        }
    }

    // Method to push an element x to stack2
    void push2(int x)
    {
        // There is at least one empty space for new element
        if (top1 < top2 - 1)
        {
            top2--;
            arr[top2] = x;
        }
        else
        {
            cout << "Stack Overflow";
            exit(1);
        }
    }

    // Method to pop an element from first stack
    int pop1()
    {
        if (top1 >= 0)
        {
            int x = arr[top1];
            top1--;
            return x;
        }
        else
        {
            cout << "Stack UnderFlow";
            exit(1);
        }
    }

    // Method to pop an element from second stack
    int pop2()
    {
        if (top2 < size)
        {
            int x = arr[top2];
            top2++;
            return x;
        }
        else
        {
            cout << "Stack UnderFlow";
            exit(1);
        }
    }
};

/* Driver program to test twStacks class */
int main()
{
    twoStacks ts(5);
    ts.push1(5);
    ts.push2(10);
    ts.push2(15);
    ts.push1(11);
    ts.push2(7);
    cout << "Popped element from stack1 is " << ts.pop1();
    ts.push2(40);
    cout << "\nPopped element from stack2 is " << ts.pop2();
    return 0;
}
```

Java

```
// Java program to implement two stacks in a
// single array
class TwoStacks
{
    int size;
    int top1 , top2;
    int arr[];
```

```

// Constructor
TwoStacks(int n)
{
    arr = new int[n];
    size = n;
    top1 = -1;
    top2 = size;
}

// Method to push an element x to stack1
void push1(int x)
{
    // There is at least one empty space for
    // new element
    if (top1 < top2 - 1)
    {
        top1++;
        arr[top1] = x;
    }
    else
    {
        System.out.println("Stack Overflow");
        System.exit(1);
    }
}

// Method to push an element x to stack2
void push2(int x)
{
    // There is at least one empty space for
    // new element
    if (top1 < top2 - 1)
    {
        top2--;
        arr[top2] = x;
    }
    else
    {
        System.out.println("Stack Overflow");
        System.exit(1);
    }
}

// Method to pop an element from first stack
int pop1()
{
    if (top1 >= 0)
    {
        int x = arr[top1];
        top1--;
        return x;
    }
    else
    {
        System.out.println("Stack Underflow");
        System.exit(1);
    }
    return 0;
}

// Method to pop an element from second stack
int pop2()
{
    if (top2 < size)
    {
        int x = arr[top2];
        top2++;
        return x;
    }
    else
    {
        System.out.println("Stack Underflow");
        System.exit(1);
    }
    return 0;
}

// Driver program to test twoStack class
public static void main(String args[])
{
    TwoStacks ts = new TwoStacks(5);
    ts.push1(5);
    ts.push2(10);
    ts.push2(15);
    ts.push1(11);
    ts.push2(7);
    System.out.println("Popped element from" +
        " stack1 is " + ts.pop1());
    ts.push2(40);
    System.out.println("Popped element from" +
        " stack2 is " + ts.pop2());
}
}
// This code has been contributed by
// Amit Khandelwal(Amit Khandelwal 1).

```

Python

Python Script to Implement two stacks in a list
class twoStacks:

```

def __init__(self, n):    #constructor
    self.size = n
    self.arr = [None] * n
    self.top1 = -1
    self.top2 = self.size

```

```

# Method to push an element x to stack1
def push1(self, x):

```

```

    # There is at least one empty space for new element
    if self.top1 < self.top2 - 1 :

```

```

        self.top1 = self.top1 + 1
        self.arr[self.top1] = x

    else:
        print("Stack Overflow ")
        exit(1)

# Method to push an element x to stack2
def push2(self, x):

    # There is at least one empty space for new element
    if self.top1 < self.top2 - 1:
        self.top2 = self.top2 - 1
        self.arr[self.top2] = x

    else :
        print("Stack Overflow ")
        exit(1)

# Method to pop an element from first stack
def pop1(self):
    if self.top1 >= 0:
        x = self.arr[self.top1]
        self.top1 = self.top1 - 1
        return x
    else:
        print("Stack Underflow ")
        exit(1)

# Method to pop an element from second stack
def pop2(self):
    if self.top2 < self.size:
        x = self.arr[self.top2]
        self.top2 = self.top2 + 1
        return x
    else:
        print("Stack Underflow ")
        exit()

# Driver program to test twoStacks class
ts = twoStacks(5)
ts.push1(5)
ts.push2(10)
ts.push2(15)
ts.push1(11)
ts.push2(7)

print("Popped element from stack1 is " + str(ts.pop1 ()))
ts.push2(40)
print("Popped element from stack2 is " + str(ts.pop2()))

# This code is contributed by Sunny Karira

```

Output:

```

Popped element from stack1 is 11
Popped element from stack2 is 40

```

Time complexity of all 4 operations of *twoStack* is $O(1)$.
We will extend to 3 stacks in an array in a separate post.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Stack
stack

Check for balanced parentheses in an expression

Given an expression string *exp* , write a program to examine whether the pairs and the orders of “(”, “)”, “{”, “}”, “[”, “]” are correct in *exp*. For example, the program should print true for *exp* = “{()}{}{()()()()}” and false for *exp* = “{()”

check-for-balanced-parentheses-in-an-expression



We strongly recommend that you click here and practice it, before moving on to the solution.

Algorithm:

- 1) Declare a character stack *S*.
- 2) Now traverse the expression string *exp*.

- a) If the current character is a starting bracket ('(' or '[' or '{') then push it to stack.
 - b) If the current character is a closing bracket (')' or ']' or '}') then pop from stack and if the popped character is the matching starting bracket then fine else parenthesis are not balanced.
- 3) After complete traversal, if there is some starting bracket left in stack then "not balanced"

Implementation:

```
#include<stdio.h>
#include<stdlib.h>
#define bool int

/* structure of a stack node */
struct sNode
{
    char data;
    struct sNode *next;
};

/* Function to push an item to stack*/
void push(struct sNode** top_ref, int new_data);

/* Function to pop an item from stack*/
int pop(struct sNode** top_ref);

/* Returns 1 if character1 and character2 are matching left
and right Parenthesis */
bool isMatchingPair(char character1, char character2)
{
    if (character1 == '(' && character2 == ')')
        return 1;
    else if (character1 == '[' && character2 == ']')
        return 1;
    else if (character1 == '{' && character2 == '}')
        return 1;
    else
        return 0;
}

/*Return 1 if expression has balanced Parenthesis */
bool areParenthesisBalanced(char exp[])
{
    int i = 0;

    /* Declare an empty character stack */
    struct sNode *stack = NULL;

    /* Traverse the given expression to check matching parenthesis */
    while (exp[i])
    {
        /*If the exp[i] is a starting parenthesis then push it*/
        if (exp[i] == '(' || exp[i] == '[' || exp[i] == '{')
            push(&stack, exp[i]);

        /* If exp[i] is a ending parenthesis then pop from stack and
        check if the popped parenthesis is a matching pair*/
        if (exp[i] == ')' || exp[i] == ']' || exp[i] == '}')
        {
            /*If we see an ending parenthesis without a pair then return false*/
            if (stack == NULL)
                return 0;

            /* Pop the top element from stack, if it is not a pair
            parenthesis of character then there is a mismatch.
            This happens for expressions like {}{} */
            else if (!isMatchingPair(pop(&stack), exp[i]))
                return 0;
        }
        i++;
    }

    /* If there is something left in expression then there is a starting
    parenthesis without a closing parenthesis */
    if (stack == NULL)
        return 1; /*balanced*/
    else
        return 0; /*not balanced*/
}

/* UTILITY FUNCTIONS */
/*driver program to test above functions*/
int main()
{
    char exp[100] = "{()[]}";
    if (areParenthesisBalanced(exp))
        printf("n Balanced ");
    else
        printf("n Not Balanced ");
    return 0;
}

/* Function to push an item to stack*/
void push(struct sNode** top_ref, int new_data)
{
    /* allocate node */
    struct sNode* new_node =
        (struct sNode*) malloc(sizeof(struct sNode));

    if (new_node == NULL)
    {
        printf("Stack overflow \n");
        getchar();
        exit(0);
    }

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*top_ref);

    /* move the head to point to the new node */
    (*top_ref) = new_node;
}

/* Function to pop an item from stack*/
int pop(struct sNode** top_ref)
```

```

{
    char res;
    struct sNode *top;

    /*If stack is empty then error */
    if (*top_ref == NULL)
    {
        printf("Stack overflow \n");
        getchar();
        exit(0);
    }
    else
    {
        top = *top_ref;
        res = top->data;
        *top_ref = top->next;
        free(top);
        return res;
    }
}

```

Time Complexity: O(n)

Auxiliary Space: O(n) for stack.

Please write comments if you find any bug in above codes/algorithms, or find other ways to solve the same problem

GATE CS Corner Company Wise Coding Practice

Stack

Next Greater Element

Given an array, print the Next Greater Element (NGE) for every element. The Next greater Element for an element x is the first greater element on the right side of x in array. Elements for which no greater element exist, consider next greater element as -1.

Examples:

- a) For any array, rightmost element always has next greater element as -1.
- b) For an array which is sorted in decreasing order, all elements have next greater element as -1.
- c) For the input array [4, 5, 2, 25], the next greater elements for each element are as follows.

```

Element   NGE
4    -->  5
5    --> 25
2    --> 25
25   --> -1

```

- d) For the input array [13, 7, 6, 12], the next greater elements for each element are as follows.

```

Element   NGE
13   -->  -1
7    --> 12
6    --> 12
12   --> -1

```

Method 1 (Simple)

Use two loops: The outer loop picks all the elements one by one. The inner loop looks for the first greater element for the element picked by outer loop. If a greater element is found then that element is printed as next, otherwise -1 is printed.

Thanks to [Sachin](#) for providing following code.

C/C++

```

// Simple C program to print next greater elements
// in a given array
#include<stdio.h>

/* prints element and NGE pair for all elements of
arr[] of size n */
void printNGE(int arr[], int n)
{
    int next, i, j;
    for (i=0; i<n; i++)
    {
        next = -1;
        for (j = i+1; j<n; j++)
        {
            if (arr[j] < arr[i])
            {
                next = arr[j];
                break;
            }
        }
        printf("%d -- %d\n", arr[i], next);
    }
}

int main()
{
    int arr[] = {11, 13, 21, 3};
    int n = sizeof(arr)/sizeof(arr[0]);
    printNGE(arr, n);
    getchar();
    return 0;
}

```

Python

```

# Function to print element and NGE pair for all elements of list
def printNGE(arr):

    for i in range(0, len(arr), 1):

        next = -1
        for j in range(i+1, len(arr), 1):

```

```

        if arr[i] < arr[j]:
            next = arr[j]
            break

    print(str(arr[i]) + " -- " + str(next))

# Driver program to test above function
arr = [11,13,21,3]
printNGE(arr)

# This code is contributed by Sunny Karira

```

Output:

```

11 -- 13
13 -- 21
21 -- -1
3 -- -1

```

Time Complexity: $O(n^2)$. The worst case occurs when all elements are sorted in decreasing order.

Method 2 (Using Stack)

Thanks to [pchild](#) for suggesting following approach.

- 1) Push the first element to stack.
- 2) Pick rest of the elements one by one and follow following steps in loop.
 -a) Mark the current element as *next*.
 -b) If stack is not empty, then pop an element from stack and compare it with *next*.
 -c) If *next* is greater than the popped element, then *next* is the next greater element for the popped element.
 -d) Keep popping from the stack while the popped element is smaller than *next*. *next* becomes the next greater element for all such popped elements
 -g) If *next* is smaller than the popped element, then push the popped element back.
- 3) After the loop in step 2 is over, pop all the elements from stack and print -1 as next element for them.

C

```

// A Stack based C program to find next greater element
// for all array elements.
#include<stdio.h>
#include<stdlib.h>
#define STACKSIZE 100

// stack structure
struct stack
{
    int top;
    int items[STACKSIZE];
};

// Stack Functions to be used by printNGE()
void push(struct stack *ps, int x)
{
    if (ps->top == STACKSIZE-1)
    {
        printf("Error: stack overflow\n");
        getchar();
        exit(0);
    }
    else
    {
        ps->top += 1;
        int top = ps->top;
        ps->items [top] = x;
    }
}

bool isEmpty(struct stack *ps)
{
    return (ps->top == -1)? true : false;
}

int pop(struct stack *ps)
{
    int temp;
    if (ps->top == -1)
    {
        printf("Error: stack underflow \n");
        getchar();
        exit(0);
    }
    else
    {
        int top = ps->top;
        temp = ps->items [top];
        ps->top -= 1;
        return temp;
    }
}

/* prints element and NGE pair for all elements of
arr[] of size n */
void printNGE(int arr[], int n)
{
    int i = 0;
    struct stack s;
    s.top = -1;
    int element, next;

    /* push the first element to stack */
    push(&s, arr[0]);

    // Iterate for rest of the elements
    for (i=1; i<n; i++)
    {
        next = arr[i];

        if (isEmpty(&s) == false)
        {
            // if stack is not empty, then pop an element from stack
            element = pop(&s);

            /* If the popped element is smaller than next, then

```

```

    a) print the pair
    b) keep popping while elements are smaller and
    stack is not empty */
while (element < next)
{
    printf("\n %d --> %d", element, next);
    if(isEmpty(&s) == true)
        break;
    element = pop(&s);
}

/* If element is greater than next, then push
the element back */
if (element > next)
    push(&s, element);
}

/* push next to stack so that we can find
next greater for it */
push(&s, next);
}

/* After iterating over the loop, the remaining
elements in stack do not have the next greater
element, so print -1 for them */
while (isEmpty(&s) == false)
{
    element = pop(&s);
    next = -1;
    printf("\n %d -- %d", element, next);
}
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {11, 13, 21, 3};
    int n = sizeof(arr)/sizeof(arr[0]);
    printNGE(arr, n);
    getchar();
    return 0;
}

```

Python

```

# Python program to print next greater element using stack

# Stack Functions to be used by printNGE()
def createStack():
    stack = []
    return stack

def isEmpty(stack):
    return len(stack) == 0

def push(stack, x):
    stack.append(x)

def pop(stack):
    if isEmpty(stack):
        print("Error : stack underflow")
    else:
        return stack.pop()

"""prints element and NGE pair for all elements of
arr"""
def printNGE(arr):
    s = createStack()
    element = 0
    next = 0

    # push the first element to stack
    push(s, arr[0])

    # iterate for rest of the elements
    for i in range(1, len(arr), 1):
        next = arr[i]

        if isEmpty(s) == False:

            # if stack is not empty, then pop an element from stack
            element = pop(s)

            """If the popped element is smaller than next, then
            a) print the pair
            b) keep popping while elements are smaller and
            stack is not empty """
            while element < next :
                print(str(element)+ " -- " + str(next))
                if isEmpty(s) == True :
                    break
                element = pop(s)

            """If element is greater than next, then push
            the element back """
            if element > next:
                push(s, element)

        """push next to stack so that we can find
        next greater for it """
        push(s, next)

    """After iterating over the loop, the remaining
    elements in stack do not have the next greater
    element, so print -1 for them """

    while isEmpty(s) == False:
        element = pop(s)
        next = -1
        print(str(element) + " -- " + str(next))

# Driver program to test above functions
arr = [11, 13, 21, 3]

```

```
printNGE(arr)
```

```
# This code is contributed by Sunny Karira
```

Output:

```
11 -- 13
13 -- 21
3 -- -1
21 -- -1
```

Time Complexity: $O(n)$. The worst case occurs when all elements are sorted in decreasing order. If elements are sorted in decreasing order, then every element is processed at most 4 times.

- a) Initially pushed to the stack.
- b) Popped from the stack when next element is being processed.
- c) Pushed back to the stack because next element is smaller.
- d) Popped from the stack in step 3 of algo.

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Stack

Reverse a stack using recursion

You are not allowed to use loop constructs like while, for...etc, and you can only use the following ADT functions on Stack S:

isEmpty(S)

push(S)

pop(S)

Solution:

The idea of the solution is to hold all values in Function Call Stack until the stack becomes empty. When the stack becomes empty, insert all held items one by one at the bottom of the stack.

For example, let the input stack be

1

First 4 is inserted at the bottom.

4

So we need a function that inserts at the bottom of a stack using the above given basic stack function.

void insertAtBottom(): First pops all stack items and stores the popped item in function call stack using recursion. And when stack becomes empty, pushes new item and all items stored in call stack.

void reverse(): This function mainly uses insertAtBottom() to pop all items one by one and insert the popped items at the bottom.

C

```
// C program to reverse a stack using recursion
#include<stdio.h>
#include<stdlib.h>
#define bool int

/* structure of a stack node */
struct sNode
{
    char data;
    struct sNode *next;
};

/* Function Prototypes */
void push(struct sNode** top_ref, int new_data);
int pop(struct sNode** top_ref);
bool isEmpty(struct sNode* top);
void print(struct sNode* top);

// Below is a recursive function that inserts an element
// at the bottom of a stack.
void insertAtBottom(struct sNode** top_ref, int item)
{
    if (isEmpty(*top_ref))
        push(top_ref, item);
    else
    {
        /* Hold all items in Function Call Stack until we
        reach end of the stack. When the stack becomes
        empty, the isEmpty(*top_ref) becomes true, the
        above if part is executed and the item is inserted
        at the bottom */
        int temp = pop(top_ref);
```

```

insertAtBottom(top_ref, item);

/* Once the item is inserted at the bottom, push all
the items held in Function Call Stack */
push(top_ref, temp);
}
}

// Below is the function that reverses the given stack using
// insertAtBottom()
void reverse(struct sNode** top_ref)
{
    if (!isEmpty(*top_ref))
    {
        /* Hold all items in Function Call Stack until we
        reach end of the stack */
        int temp = pop(top_ref);
        reverse(top_ref);

        /* Insert all the items (held in Function Call Stack)
        one by one from the bottom to top. Every item is
        inserted at the bottom */
        insertAtBottom(top_ref, temp);
    }
}

/* Driver program to test above functions */
int main()
{
    struct sNode *s = NULL;
    push(&s, 4);
    push(&s, 3);
    push(&s, 2);
    push(&s, 1);

    printf("\n Original Stack ");
    print(s);
    reverse(&s);
    printf("\n Reversed Stack ");
    print(s);
    return 0;
}

/* Function to check if the stack is empty */
bool isEmpty(struct sNode* top)
{
    return (top == NULL)? 1 : 0;
}

/* Function to push an item to stack*/
void push(struct sNode** top_ref, int new_data)
{
    /* allocate node */
    struct sNode* new_node =
        (struct sNode*) malloc(sizeof(struct sNode));

    if (new_node == NULL)
    {
        printf("Stack overflow \n");
        exit(0);
    }

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*top_ref);

    /* move the head to point to the new node */
    (*top_ref) = new_node;
}

/* Function to pop an item from stack*/
int pop(struct sNode** top_ref)
{
    char res;
    struct sNode *top;

    /* If stack is empty then error */
    if (*top_ref == NULL)
    {
        printf("Stack overflow \n");
        exit(0);
    }
    else
    {
        top = *top_ref;
        res = top->data;
        *top_ref = top->next;
        free(top);
        return res;
    }
}

/* Function to print a linked list */
void print(struct sNode* top)
{
    printf("\n");
    while (top != NULL)
    {
        printf("%d ", top->data);
        top = top->next;
    }
}

```

Python

```
# Python program to reverse a stack using recursion
```

```
# Below is a recursive function that inserts an element
# at the bottom of a stack.
def insertAtBottom(stack, item):
    if isEmpty(stack):
        push(stack, item)
    else:
        temp = pop(stack)
        insertAtBottom(stack, item)
        push(stack, temp)
```

```
# Below is the function that reverses the given stack
# using insertAtBottom()
def reverse(stack):
    if not isEmpty(stack):
        temp = pop(stack)
        reverse(stack)
        insertAtBottom(stack, temp)
```

```
# Below is a complete running program for testing above
# functions.
```

```
# Function to create a stack. It initializes size of stack
# as 0
def createStack():
    stack = []
    return stack
```

```
# Function to check if the stack is empty
def isEmpty( stack ):
    return len(stack) == 0
```

```
#Function to push an item to stack
def push( stack, item ):
    stack.append( item )
```

```
# Function to pop an item from stack
def pop( stack ):
```

```
#If stack is empty then error
if( isEmpty( stack ) ):
    print("Stack Underflow ")
    exit(1)
```

```
return stack.pop()
```

```
# Function to print the stack
def prints(stack):
    for i in range(len(stack)-1, -1, -1):
        print(stack[i], end = ' ')
    print()
```

```
# Driver program to test above functions
```

```
stack = createStack()
push( stack, str(4) )
push( stack, str(3) )
push( stack, str(2) )
push( stack, str(1) )
print("Original Stack ")
prints(stack)
```

```
reverse(stack)
```

```
print("Reversed Stack ")
prints(stack)
```

```
# This code is contributed by Sunny Karira
```

Output:

```
Original Stack
1 2 3 4
Reversed Stack
4 3 2 1
```

Please write comments if you find any bug in above code/algorithm, or find other ways to solve the same problem.

GATE CS Corner
Company Wise Coding Practice

Sort a stack using recursion

Given a stack, sort it using recursion. Use of any loop constructs like while, for...etc is not allowed. We can only use the following ADT functions on Stack S:

```
is_empty(S) : Tests whether stack is empty or not.
push(S)    : Adds new element to the stack.
pop(S)     : Removes top element from the stack.
top(S)     : Returns value of the top element. Note that this
            function does not remove element from the stack.
```

Example:

Input: -3

We strongly recommend you to minimize your browser and try this yourself first.

This problem is mainly a variant of [Reverse stack using recursion](#).

The idea of the solution is to hold all values in Function Call Stack until the stack becomes empty. When the stack becomes empty, insert all held items one by one in sorted order. Here sorted order is important.

Algorithm

We can use below algorithm to sort stack elements:

```
sortStack(stack S)
if stack is not empty:
    temp = pop(S);
    sortStack(S);
    sortedInsert(S, temp);
```

Below algorithm is to insert element in sorted order:

```
sortedInsert(Stack S, element)
if stack is empty OR element > top element
    push(S, elem)
else
    temp = pop(S)
    sortedInsert(S, element)
    push(S, temp)
```

Illustration:

Let given stack be
-3

Let us illustrate sorting of stack using above example:

First pop all the elements from the stack and store popped element in variable 'temp'. After popping all the elements function's stack frame will look like:

```
temp = -3 --> stack frame #1
temp = 14 --> stack frame #2
temp = 18 --> stack frame #3
temp = -5 --> stack frame #4
temp = 30 --> stack frame #5
```

Now stack is empty and 'insert_in_sorted_order()' function is called and it inserts 30 (from stack frame #5) at the bottom of the stack. Now stack looks like below:

```
30
Now next element i.e. -5 (from stack frame #4) is picked. Since -5
30 -5
```

Next 18 (from stack frame #3) is picked. Since 18

```
30 18
-5
```

Next 14 (from stack frame #2) is picked. Since 14

```
30 14
-5
```

Now -3 (from stack frame #1) is picked, as -3 30 **-3** -5

Implementation:

Below is C implementation of above algorithm.

```
// C program to sort a stack using recursion
#include <stdio.h>
#include <stdlib.h>

// Stack is represented using linked list
struct stack
{
    int data;
    struct stack *next;
};

// Utility function to initialize stack
void initStack(struct stack **s)
{
    *s = NULL;
}

// Utility function to check if stack is empty
int isEmpty(struct stack *s)
{
    return s == NULL;
}
```



```

if (s == NULL)
    return 1;
return 0;
}

// Utility function to push an item to stack
void push(struct stack **s, int x)
{
    struct stack *p = (struct stack *)malloc(sizeof("p));

    if (p == NULL)
    {
        fprintf(stderr, "Memory allocation failed.\n");
        return;
    }

    p->data = x;
    p->next = *s;
    *s = p;
}

// Utility function to remove an item from stack
int pop(struct stack **s)
{
    int x;
    struct stack *temp;

    x = (*s)->data;
    temp = *s;
    (*s) = (*s)->next;
    free(temp);

    return x;
}

// Function to find top item
int top(struct stack *s)
{
    return (s->data);
}

// Recursive function to insert an item x in sorted way
void sortedInsert(struct stack **s, int x)
{
    // Base case: Either stack is empty or newly inserted
    // item is greater than top (more than all existing)
    if (isEmpty(*s) || x > top(*s))
    {
        push(s, x);
        return;
    }

    // If top is greater, remove the top item and recur
    int temp = pop(s);
    sortedInsert(s, x);

    // Put back the top item removed earlier
    push(s, temp);
}

// Function to sort stack
void sortStack(struct stack **s)
{
    // If stack is not empty
    if (!isEmpty(*s))
    {
        // Remove the top item
        int x = pop(s);

        // Sort remaining stack
        sortStack(s);

        // Push the top item back in sorted stack
        sortedInsert(s, x);
    }
}

// Utility function to print contents of stack
void printStack(struct stack *s)
{
    while (s)
    {
        printf("%d ", s->data);
        s = s->next;
    }
    printf("\n");
}

// Driver Program
int main(void)
{
    struct stack *top;

    initStack(&top);
    push(&top, 30);
    push(&top, -5);
    push(&top, 10);
    push(&top, 14);
    push(&top, -3);

    printf("Stack elements before sorting:\n");
    printStack(top);

    sortStack(&top);
    printf("\n\n");

    printf("Stack elements after sorting:\n");
    printStack(top);

    return 0;
}

```

Output:

```
Stack elements before sorting:
```

-3 14 18 -5 30

Stack elements after sorting:
30 18 14 -3 -5

Exercise: Modify above code to reverse stack in descending order.

This article is contributed by **Narendra Kangraikar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Stack
Recursion
stack

The Stock Span Problem

The **stock span problem** is a financial problem where we have a series of n daily price quotes for a stock and we need to calculate span of stock's price for all n days.

The span S_i of the stock's price on a given day i is defined as the maximum number of consecutive days just before the given day, for which the price of the stock on the current day is less than or equal to its price on the given day.

For example, if an array of 7 days prices is given as {100, 80, 60, 70, 60, 75, 85}, then the span values for corresponding 7 days are {1, 1, 1, 2, 1, 4, 6}



We strongly recommend that you click here and practice it, before moving on to the solution.

A Simple but inefficient method

Traverse the input price array. For every element being visited, traverse elements on left of it and increment the span value of it while elements on the left side are smaller.

Following is implementation of this method.

C

```
// C program for brute force method to calculate stock span values
#include <stdio.h>

// Fills array S[] with span values
void calculateSpan(int price[], int n, int S[])
{
    // Span value of first day is always 1
    S[0] = 1;

    // Calculate span value of remaining days by linearly checking
    // previous days
    for (int i = 1; i < n; i++)
    {
        S[i] = 1; // Initialize span value

        // Traverse left while the next element on left is smaller
        // than price[i]
        for (int j = i-1; (j >= 0) && (price[j] >= price[i]); j--)
            S[i]++;
    }
}

// A utility function to print elements of array
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
}

// Driver program to test above function
int main()
{
    int price[] = {10, 4, 5, 90, 120, 80};
    int n = sizeof(price)/sizeof(price[0]);
    int S[n];

    // Fill the span values in array S[]
    calculateSpan(price, n, S);

    // print the calculated span values
    printArray(S, n);

    return 0;
}
```

Python

```
# Python program for brute force method to calculate stock span values

# Fills list S[] with span values
def calculateSpan(price, n, S):

    # Span value of first day is always 1
    S[0] = 1

    # Calculate span value of remaining days by linearly
    # checking previous days
    for i in range(1, n, 1):
        S[i] = 1 # Initialize span value

        # Traverse left while the next element on left is
        # smaller than price[i]
        j = i - 1
```

```

while (j>=0) and (price[i] >= price[j]) :
    S[i] += 1
    j -= 1

# A utility function to print elements of array
def printArray(arr, n):

    for i in range(n):
        print(arr[i], end = " ")

# Driver program to test above function
price = [10, 4, 5, 90, 120, 80]
n = len(price)
S = [None] * n

# Fill the span values in list S[]
calculateSpan(price, n, S)

# print the calculated span values
printArray(S, n)

# This code is contributed by Sunny Karira

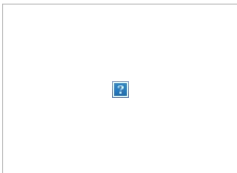
```

Time Complexity of the above method is $O(n^2)$. We can calculate stock span values in $O(n)$ time.

A Linear Time Complexity Method

We see that $S[i]$ on day i can be easily computed if we know the closest day preceding i , such that the price is greater than on that day than the price on day i . If such a day exists, let's call it $h(i)$, otherwise, we define $h(i) = -1$.

The span is now computed as $S[i] = i - h(i)$. See the following diagram.



To implement this logic, we use a stack as an abstract data type to store the days i , $h(i)$, $h(h(i))$ and so on. When we go from day $i-1$ to i , we pop the days when the price of the stock was less than or equal to $price[i]$ and then push the value of day i back into the stack.

Following is C++ implementation of this method.

C++

```

// a linear time solution for stock span problem
#include <iostream>
#include <stack>
using namespace std;

// A stack based efficient method to calculate stock span values
void calculateSpan(int price[], int n, int S[])
{
    // Create a stack and push index of first element to it
    stack<int> st;
    st.push(0);

    // Span value of first element is always 1
    S[0] = 1;

    // Calculate span values for rest of the elements
    for (int i = 1; i < n; i++)
    {
        // Pop elements from stack while stack is not empty and top of
        // stack is smaller than price[i]
        while (!st.empty() && price[st.top()] <= price[i])
            st.pop();

        // If stack becomes empty, then price[i] is greater than all elements
        // on left of it, i.e., price[0], price[1]..price[i-1]. Else price[i]
        // is greater than elements after top of stack
        S[i] = (st.empty()) ? (i + 1) : (i - st.top());

        // Push this element to stack
        st.push(i);
    }
}

// A utility function to print elements of array
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
}

// Driver program to test above function
int main()
{
    int price[] = {10, 4, 5, 90, 120, 80};
    int n = sizeof(price)/sizeof(price[0]);
    int S[n];

    // Fill the span values in array S[]
    calculateSpan(price, n, S);

    // print the calculated span values
    printArray(S, n);

    return 0;
}

```

Python

```

# A linear time solution for stock span problem

# A stack based efficient method to calculate s

```

```
def calculateSpan(price, S):

    n = len(price)
    # Create a stack and push index of first element to it
    st = []
    st.append(0)

    # Span value of first element is always 1
    S[0] = 1

    # Calculate span values for rest of the elements
    for i in range(1, n):

        # Pop elements from stack while stack is not
        # empty and top of stack is smaller than price[i]
        while( len(st) > 0 and price[st[0]] <= price[i]):
            st.pop()

        # If stack becomes empty, then price[i] is greater
        # than all elements on left of it, i.e. price[0],
        # price[1], ..price[i-1]. Else the price[i] is
        # greater than elements after top of stack
        S[i] = i+1 if len(st) <= 0 else (i - st[0])

        # Push this element to stack
        st.append(i)

# A utility function to print elements of array
def printArray(arr, n):
    for i in range(0,n):
        print arr[i],

# Driver program to test above function
price = [10, 4, 5, 90, 120, 80]
S = [0 for i in range(len(price)+1)]

# Fill the span values in array S[]
calculateSpan(price, S)

# Print the calculated span values
printArray(S, len(price))

# This code is contributed by Nikhil Kumar Singh (nickzuck_007)
```

Output:

```
1 1 2 4 5 1
```

Time Complexity: $O(n)$. It seems more than $O(n)$ at first look. If we take a closer look, we can observe that every element of array is added and removed from stack at most once. So there are total $2n$ operations at most. Assuming that a stack operation takes $O(1)$ time, we can say that the time complexity is $O(n)$.

Auxiliary Space: $O(n)$ in worst case when all elements are sorted in decreasing order.

References:

[http://en.wikipedia.org/wiki/Stack_\(abstract_data_type\)#The_Stack_Span_Problem](http://en.wikipedia.org/wiki/Stack_(abstract_data_type)#The_Stack_Span_Problem)
<http://crypto.cs.mcgill.ca/~crepeau/CS250/2004/Stack-1.pdf>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Stack
stack

Design and Implement Special Stack Data Structure | Added Space Optimized Version

Question: Design a Data Structure SpecialStack that supports all the stack operations like push(), pop(), isEmpty(), isFull() and an additional operation getMin() which should return minimum element from the SpecialStack. All these operations of SpecialStack must be $O(1)$. To implement SpecialStack, you should only use standard Stack data structure and no other data structure like arrays, list, .. etc.

Example:

```
Consider the following SpecialStack
16 --> TOP
15
29
19
18

When getMin() is called it should return 15, which is the minimum
element in the current stack.

If we do pop two times on stack, the stack becomes
29 --> TOP
19
18

When getMin() is called, it should return 18 which is the minimum
```

in the current stack.

Solution: Use two stacks: one to store actual stack elements and other as an auxiliary stack to store minimum values. The idea is to do push() and pop() operations in such a way that the top of auxiliary stack is always the minimum. Let us see how push() and pop() operations work.

Push(int x) // inserts an element x to Special Stack

- 1) push x to the first stack (the stack with actual elements)
- 2) compare x with the top element of the second stack (the auxiliary stack). Let the top element be y.
.....a) If x is smaller than y then push x to the auxiliary stack.
.....b) If x is greater than y then push y to the auxiliary stack.

int Pop() // removes an element from Special Stack and return the removed element

- 1) pop the top element from the auxiliary stack.
- 2) pop the top element from the actual stack and return it.

The step 1 is necessary to make sure that the auxiliary stack is also updated for future operations.

int getMin() // returns the minimum element from Special Stack

- 1) Return the top element of auxiliary stack.

We can see that **all above operations are O(1)**.

Let us see an example. Let us assume that both stacks are initially empty and 18, 19, 29, 15 and 16 are inserted to the SpecialStack.

When we insert 18, both stacks change to following.

Actual Stack

18

Following is C++ implementation for SpecialStack class. In the below implementation, SpecialStack inherits from Stack and has one Stack object *min* which work as auxiliary stack.

```
#include<iostream>
#include<stdlib.h>

using namespace std;

/* A simple stack class with basic stack functionalities */
class Stack
{
private:
    static const int max = 100;
    int arr[max];
    int top;
public:
    Stack() { top = -1; }
    bool isEmpty();
    bool isFull();
    int pop();
    void push(int x);
};

/* Stack's member method to check if the stack is empty */
bool Stack::isEmpty()
{
    if(top == -1)
        return true;
    return false;
}

/* Stack's member method to check if the stack is full */
bool Stack::isFull()
{
    if(top == max - 1)
        return true;
    return false;
}

/* Stack's member method to remove an element from it */
int Stack::pop()
{
    if(isEmpty())
    {
        cout<<"Stack Underflow";
        abort();
    }
    int x = arr[top];
    top--;
    return x;
}

/* Stack's member method to insert an element to it */
void Stack::push(int x)
{
    if(isFull())
    {
        cout<<"Stack Overflow";
        abort();
    }
    top++;
    arr[top] = x;
}

/* A class that supports all the stack operations and one additional
operation getMin() that returns the minimum element from stack at
any time. This class inherits from the stack class and uses an
auxiliary stack that holds minimum elements */
class SpecialStack: public Stack
{
    Stack min;
public:
    int pop();
    void push(int x);
    int getMin();
};

/* SpecialStack's member method to insert an element to it. This method
makes sure that the min stack is also updated with appropriate minimum
values */
void SpecialStack::push(int x)
{
    if(isEmpty()==true)
    {
        Stack::push(x);
        min.push(x);
    }
}
```

```

else
{
    Stack::push(x);
    int y = min.pop();
    min.push(y);
    if( x < y )
        min.push(x);
    else
        min.push(y);
}
}

/* SpecialStack's member method to remove an element from it. This method
removes top element from min stack also. */
int SpecialStack::pop()
{
    int x = Stack::pop();
    min.pop();
    return x;
}

/* SpecialStack's member method to get minimum element from it. */
int SpecialStack::getMin()
{
    int x = min.pop();
    min.push(x);
    return x;
}

/* Driver program to test SpecialStack methods */
int main()
{
    SpecialStack s;
    s.push(10);
    s.push(20);
    s.push(30);
    cout<<s.getMin()<<endl;
    s.push(5);
    cout<<s.getMin();
    return 0;
}

```

Output:

10

5

Space Optimized Version

The above approach can be optimized. We can limit the number of elements in auxiliary stack. We can push only when the incoming element of main stack is smaller than or equal to top of auxiliary stack. Similarly during pop, if the pop of

```

/* SpecialStack's member method to insert an element to it. This method
makes sure that the min stack is also updated with appropriate minimum
values */
void SpecialStack::push(int x)
{
    if(isEmpty()==true)
    {
        Stack::push(x);
        min.push(x);
    }
    else
    {
        Stack::push(x);
        int y = min.pop();
        min.push(y);

        /* push only when the incoming element of main stack is smaller
        than or equal to top of auxiliary stack */
        if( x <= y )
            min.push(x);
    }
}

/* SpecialStack's member method to remove an element from it. This method
removes top element from min stack also. */
int SpecialStack::pop()
{
    int x = Stack::pop();
    int y = min.pop();

    /* Push the popped element y back only if it is not equal to x */
    if ( y != x )
        min.push(y);

    return x;
}

```

Thanks to @Venki, @swarup and @Jing Huang for their inputs.

Please write comments if you find the above code incorrect, or find other ways to solve the same problem.

GATE CS Corner

Company Wise Coding Practice

Stack
stack
Stack-Queue
StackAndQueue

Implement Stack using Queues

The problem is opposite of [this](#) post. We are given a Queue data structure that supports standard operations like enqueue() and dequeue(). We need to implement a Stack data structure using only instances of Queue and queue operations allowed on the instances.

Stack and Queue with insert and delete operations



A stack can be implemented using two queues. Let stack to be implemented be 's' and queues used to implement be 'q1' and 'q2'. Stack 's' can be implemented in two ways:

Method 1 (By making push operation costly)

This method makes sure that newly entered element is always at the front of 'q1', so that pop operation just dequeues from 'q1'. 'q2' is used to put every new element at front of 'q1'.

```
push(s, x) // x is the element to be pushed and s is stack
1) Enqueue x to q2
2) One by one dequeue everything from q1 and enqueue to q2.
3) Swap the names of q1 and q2
// Swapping of names is done to avoid one more movement of all elements
// from q2 to q1.

pop(s)
1) Dequeue an item from q1 and return it.
```

Method 2 (By making pop operation costly)

In push operation, the new element is always enqueued to q1. In pop() operation, if q2 is empty then all the elements except the last, are moved to q2. Finally the last element is dequeued from q1 and returned.

```
push(s, x)
1) Enqueue x to q1 (assuming size of q1 is unlimited).

pop(s)
1) One by one dequeue everything except the last element from q1 and enqueue to q2.
2) Dequeue the last item of q1, the dequeued item is result, store it.
3) Swap the names of q1 and q2
4) Return the item stored in step 2.
// Swapping of names is done to avoid one more movement of all elements
// from q2 to q1.
```

References:

[Implement Stack using Two Queues](#)

This article is compiled by **Sumit Jain** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Queue
Stack
Queue
stack

Design a stack with operations on middle element

How to implement a stack which will support following operations in **O(1) time complexity**?

- 1) push() which adds an element to the top of stack.
- 2) pop() which removes an element from top of stack.
- 3) findMiddle() which will return middle element of the stack.
- 4) deleteMiddle() which will delete the middle element.

Push and pop are standard stack operations.

The important question is, whether to use a linked list or array for implementation of stack?

Please note that, we need to find and delete middle element. Deleting an element from middle is not $O(1)$ for array. Also, we may need to move the middle pointer up when we push an element and move down when we pop(). In singly linked list, moving middle pointer in both directions is not possible.

The idea is to use Doubly Linked List (DLL). We can delete middle element in $O(1)$ time by maintaining mid pointer. We can move mid pointer in both directions using previous and next pointers.

Following is C implementation of push(), pop() and findMiddle() operations. Implementation of deleteMiddle() is left as an exercise. If there are even elements in stack, findMiddle() returns the first middle element. For example, if stack contains {1, 2, 3, 4}, then findMiddle() would return 2.

```
/* Program to implement a stack that supports findMiddle() and deleteMiddle
   in O(1) time */
#include <stdio.h>
#include <stdlib.h>

/* A Doubly Linked List Node */
struct DLLNode
{
    struct DLLNode *prev;
    int data;
    struct DLLNode *next;
};

/* Representation of the stack data structure that supports findMiddle()
   in O(1) time. The Stack is implemented using Doubly Linked List. It
   maintains pointer to head node, pointer to middle node and count of
   nodes */
struct myStack
{
    struct DLLNode *head;
    struct DLLNode *mid;
    int count;
};

/* Function to create the stack data structure */
struct myStack *createMyStack()
{
    struct myStack *ms =
        (struct myStack*) malloc(sizeof(struct myStack));
    ms->count = 0;
    return ms;
};

/* Function to push an element to the stack */
void push(struct myStack *ms, int new_data)
{
    /* allocate DLLNode and put in data */
    struct DLLNode* new_DLLNode =
        (struct DLLNode*) malloc(sizeof(struct DLLNode));
    new_DLLNode->data = new_data;

    /* Since we are adding at the beginning,
       prev is always NULL */
    new_DLLNode->prev = NULL;

    /* link the old list off the new DLLNode */
    new_DLLNode->next = ms->head;

    /* Increment count of items in stack */
    ms->count += 1;

    /* Change mid pointer in two cases
       1) Linked List is empty
       2) Number of nodes in linked list is odd */
    if (ms->count == 1)
    {
        ms->mid = new_DLLNode;
    }
    else
    {
        ms->head->prev = new_DLLNode;

        if (ms->count & 1) // Update mid if ms->count is odd
            ms->mid = ms->mid->prev;
    }

    /* move head to point to the new DLLNode */
    ms->head = new_DLLNode;
}

/* Function to pop an element from stack */
int pop(struct myStack *ms)
{
    /* Stack underflow */
    if (ms->count == 0)
    {
        printf("Stack is empty\n");
        return -1;
    }

    struct DLLNode *head = ms->head;
    int item = head->data;
    ms->head = head->next;

    /* If linked list doesn't become empty, update prev
       // of new head as NULL
       if (ms->head != NULL)
           ms->head->prev = NULL;

    ms->count -= 1;

    // update the mid pointer when we have even number of
    // elements in the stack, i.e move down the mid pointer.
    if (!(ms->count & 1))
        ms->mid = ms->mid->next;

    free(head);

    return item;
}

// Function for finding middle of the stack
int findMiddle(struct myStack *ms)
{
    if (ms->count == 0)
    {

```



```

    printf("Stack is empty now\n");
    return -1;
}

return ms->mid->data;
}

// Driver program to test functions of myStack
int main()
{
    /* Let us create a stack using push() operation*/
    struct myStack *ms = createMyStack();
    push(ms, 11);
    push(ms, 22);
    push(ms, 33);
    push(ms, 44);
    push(ms, 55);
    push(ms, 66);
    push(ms, 77);

    printf("Item popped is %d\n", pop(ms));
    printf("Item popped is %d\n", pop(ms));
    printf("Middle Element is %d\n", findMiddle(ms));
    return 0;
}

```

Output:

```

Item popped is 77
Item popped is 66
Middle Element is 33

```

This article is contributed by **Chandra Prakash**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Stack
stack

How to efficiently implement k stacks in a single array?

We have discussed [space efficient implementation of 2 stacks in a single array](#). In this post, a general solution for k stacks is discussed. Following is the detailed problem statement.

Create a data structure *kStacks* that represents *k* stacks. Implementation of *kStacks* should use only one array, i.e., *k* stacks should use the same array for storing elements. Following functions must be supported by *kStacks*.

push(int x, int sn) → pushes x to stack number 'sn' where sn is from 0 to k-1

pop(int sn) → pops an element from stack number 'sn' where sn is from 0 to k-1

Method 1 (Divide the array in slots of size n/k)

A simple way to implement k stacks is to divide the array in k slots of size n/k each, and fix the slots for different stacks, i.e., use arr[0] to arr[n/k-1] for first stack, and arr[n/k] to arr[2n/k-1] for stack2 where arr[] is the array to be used to implement two stacks and size of array be n.

The problem with this method is inefficient use of array space. A stack push operation may result in stack overflow even if there is space available in arr[]. For example, say the k is 2 and array size (n) is 6 and we push 3 elements to first and do not push anything to second second stack. When we push 4th element to first, there will be overflow even if we have space for 3 more elements in array.

Method 2 (A space efficient implementation)

The idea is to use two extra arrays for efficient implementation of k stacks in an array. This may not make much sense for integer stacks, but stack items can be large for example stacks of employees, students, etc where every item is of hundreds of bytes. For such large stacks, the extra space used is comparatively very less as we use two *integer* arrays as extra space.

Following are the two extra arrays are used:

1) top[]: This is of size k and stores indexes of top elements in all stacks.

2) next[]: This is of size n and stores indexes of next item for the items in array arr[]. Here arr[] is actual array that stores k stacks.

Together with k stacks, a stack of free slots in arr[] is also maintained. The top of this stack is stored in a variable 'free'.

All entries in top[] are initialized as -1 to indicate that all stacks are empty. All entries next[i] are initialized as i+1 because all slots are free initially and pointing to next slot. Top of free stack, 'free' is initialized as 0.

Following is C++ implementation of the above idea.

```

// A C++ program to demonstrate implementation of k stacks in a single
// array in time and space efficient way
#include<iostream>
#include<climits>
using namespace std;

// A C++ class to represent k stacks in a single array of size n
class kStacks
{
    int *arr; // Array of size n to store actual content to be stored in stacks
    int *top; // Array of size k to store indexes of top elements of stacks
    int *next; // Array of size n to store next entry in all stacks
    // and free list
    int n, k;
    int free; // To store beginning index of free list
public:
    //constructor to create k stacks in an array of size n
    kStacks(int k, int n);

    // A utility function to check if there is space available
    bool isFull() { return (free == -1); }

    // To push an item in stack number 'sn' where sn is from 0 to k-1
    void push(int item, int sn);

    // To pop an from stack number 'sn' where sn is from 0 to k-1
    int pop(int sn);

    // To check whether stack number 'sn' is empty or not
    bool isEmpty(int sn) { return (top[sn] == -1); }
};

//constructor to create k stacks in an array of size n
kStacks::kStacks(int k1, int n1)
{
    // Initialize n and k, and allocate memory for all arrays
    k = k1, n = n1;
    arr = new int[n];
    top = new int[k];
    next = new int[n];

    // Initialize all stacks as empty
    for (int i = 0; i < k; i++)

```

```

top[i] = -1;

// Initialize all spaces as free
free = 0;
for (int i=0; i<n-1; i++)
    next[i] = i+1;
next[n-1] = -1; // -1 is used to indicate end of free list
}

// To push an item in stack number 'sn' where sn is from 0 to k-1
void kStacks::push(int item, int sn)
{
    // Overflow check
    if (isFull())
    {
        cout << "nStack Overflow\n";
        return;
    }

    int i = free;    // Store index of first free slot

    // Update index of free slot to index of next slot in free list
    free = next[i];

    // Update next of top and then top for stack number 'sn'
    next[i] = top[sn];
    top[sn] = i;

    // Put the item in array
    arr[i] = item;
}

// To pop an from stack number 'sn' where sn is from 0 to k-1
int kStacks::pop(int sn)
{
    // Underflow check
    if (isEmpty(sn))
    {
        cout << "nStack Underflow\n";
        return INT_MAX;
    }

    // Find index of top item in stack number 'sn'
    int i = top[sn];

    top[sn] = next[i]; // Change top to store next of previous top

    // Attach the previous top to the beginning of free list
    next[i] = free;
    free = i;

    // Return the previous top item
    return arr[i];
}

/* Driver program to test twStacks class */
int main()
{
    // Let us create 3 stacks in an array of size 10
    int k = 3, n = 10;
    kStacks ks(k, n);

    // Let us put some items in stack number 2
    ks.push(15, 2);
    ks.push(45, 2);

    // Let us put some items in stack number 1
    ks.push(17, 1);
    ks.push(49, 1);
    ks.push(39, 1);

    // Let us put some items in stack number 0
    ks.push(11, 0);
    ks.push(9, 0);
    ks.push(7, 0);

    cout << "Popped element from stack 2 is " << ks.pop(2) << endl;
    cout << "Popped element from stack 1 is " << ks.pop(1) << endl;
    cout << "Popped element from stack 0 is " << ks.pop(0) << endl;

    return 0;
}

```

Output:

```

Popped element from stack 2 is 45
Popped element from stack 1 is 39
Popped element from stack 0 is 7

```

Time complexities of operations push() and pop() is O(1).

The best part of above implementation is, if there is a slot available in stack, then an item can be pushed in any of the stacks, i.e., no wastage of space.

This article is contributed by **Sachin**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Stack
stack
Stack-Queue

Sort a stack using recursion

Given a stack, sort it using recursion. Use of any loop constructs like while, for...etc is not allowed. We can only use the following ADT functions on Stack S:

```

is_empty(S) : Tests whether stack is empty or not.
push(S)      : Adds new element to the stack.
pop(S)       : Removes top element from the stack.
top(S)       : Returns value of the top element. Note that this
              function does not remove element from the stack.

```

Example:

Input: -3

We strongly recommend you to minimize your browser and try this yourself first.

This problem is mainly a variant of [Reverse stack using recursion](#).

The idea of the solution is to hold all values in Function Call Stack until the stack becomes empty. When the stack becomes empty, insert all held items one by one in sorted order. Here sorted order is important.

Algorithm

We can use below algorithm to sort stack elements:

```
sortStack(stack S)
if stack is not empty:
    temp = pop(S);
    sortStack(S);
    sortedInsert(S, temp);
```

Below algorithm is to insert element in sorted order:

```
sortedInsert(Stack S, element)
if stack is empty OR element > top element
    push(S, elem)
else
    temp = pop(S)
    sortedInsert(S, element)
    push(S, temp)
```

Illustration:

Let given stack be
-3
Let us illustrate sorting of stack using above example:

First pop all the elements from the stack and store popped element in variable 'temp'. After popping all the elements function's stack frame will look like:

```
temp = -3 --> stack frame #1
temp = 14 --> stack frame #2
temp = 18 --> stack frame #3
temp = -5 --> stack frame #4
temp = 30 --> stack frame #5
```

Now stack is empty and 'insert_in_sorted_order()' function is called and it inserts 30 (from stack frame #5) at the bottom of the stack. Now stack looks like below:

```
30
Now next element i.e. -5 (from stack frame #4) is picked. Since -5
30 -5
```

Next 18 (from stack frame #3) is picked. Since 18
30 **18**
-5

Next 14 (from stack frame #2) is picked. Since 14
30 **14**
-5

Now -3 (from stack frame #1) is picked, as -3 30 **-3** -5

Implementation:

Below is C implementation of above algorithm.

```
// C program to sort a stack using recursion
#include <stdio.h>
#include <stdlib.h>

// Stack is represented using linked list
struct stack
{
    int data;
    struct stack *next;
};

// Utility function to initialize stack
void initStack(struct stack **s)
{
    *s = NULL;
}

// Utility function to check if stack is empty
int isEmpty(struct stack *s)
{
    if (s == NULL)
        return 1;
    return 0;
}

// Utility function to push an item to stack
void push(struct stack **s, int x)
{
    struct stack *p = (struct stack *)malloc(sizeof(*p));

    if (p == NULL)
    {
        fprintf(stderr, "Memory allocation failed.\n");
    }
}
```

```

        return;
    }

    p->data = x;
    p->next = *s;
    *s = p;
}

// Utility function to remove an item from stack
int pop(struct stack **s)
{
    int x;
    struct stack *temp;

    x = (*s)->data;
    temp = *s;
    (*s) = (*s)->next;
    free(temp);

    return x;
}

// Function to find top item
int top(struct stack *s)
{
    return (s->data);
}

// Recursive function to insert an item x in sorted way
void sortedInsert(struct stack **s, int x)
{
    // Base case: Either stack is empty or newly inserted
    // item is greater than top (more than all existing)
    if (isEmpty(*s) || x > top(*s))
    {
        push(s, x);
        return;
    }

    // If top is greater, remove the top item and recur
    int temp = pop(s);
    sortedInsert(s, x);

    // Put back the top item removed earlier
    push(s, temp);
}

// Function to sort stack
void sortStack(struct stack **s)
{
    // If stack is not empty
    if (!isEmpty(*s))
    {
        // Remove the top item
        int x = pop(s);

        // Sort remaining stack
        sortStack(s);

        // Push the top item back in sorted stack
        sortedInsert(s, x);
    }
}

// Utility function to print contents of stack
void printStack(struct stack *s)
{
    while (s)
    {
        printf("%d ", s->data);
        s = s->next;
    }
    printf("\n");
}

// Driver Program
int main(void)
{
    struct stack *top;

    initStack(&top);
    push(&top, 30);
    push(&top, -5);
    push(&top, 18);
    push(&top, 14);
    push(&top, -3);

    printf("Stack elements before sorting:\n");
    printStack(top);

    sortStack(&top);
    printf("\n\n");

    printf("Stack elements after sorting:\n");
    printStack(top);

    return 0;
}

```

Output:

```
Stack elements before sorting:
-3 14 18 -5 30
```

```
Stack elements after sorting:
30 18 14 -3 -5
```

Exercise: Modify above code to reverse stack in descending order.

This article is contributed by **Narendra Kangralkar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner **Company Wise Coding Practice**

Stack

Queue | Set 1 (Introduction and Array Implementation)

Like [Stack](#), [Queue](#) is a linear structure which follows a particular order in which the operations are performed. The order is **First In First Out** (FIFO). A good example of queue is any queue of consumers for a resource where the consumer that came first is served first.

The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

Operations on Queue:

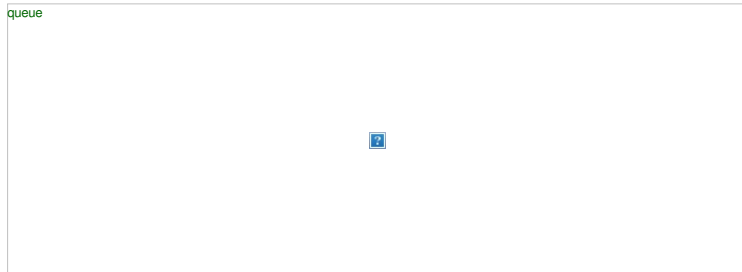
Mainly the following four basic operations are performed on queue:

Enqueue: Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.

Dequeue: Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.

Front: Get the front item from queue.

Rear: Get the last item from queue.



Applications of Queue:

Queue is used when things don't have to be processed immediately, but have to be processed in **First In First Out** order like [Breadth First Search](#). This property of Queue makes it also useful in following kind of scenarios.

- 1) When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
- 2) When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.

See [this](#) for more detailed applications of Queue and Stack.

Array implementation Of Queue

For implementing queue, we need to keep track of two indices, front and rear. We enqueue an item at the rear and dequeue an item from front. If we simply increment front and rear indices, then there may be problems, front may reach end of the array. The solution to this problem is to increase front and rear in circular manner (See [this](#) for details)

```
// C program for array implementation of queue
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

// A structure to represent a queue
struct Queue
{
    int front, rear, size;
    unsigned capacity;
    int* array;
};

// function to create a queue of given capacity. It initializes size of
// queue as 0
struct Queue* createQueue(unsigned capacity)
{
    struct Queue* queue = (struct Queue*) malloc(sizeof(struct Queue));
    queue->capacity = capacity;
    queue->front = queue->size = 0;
    queue->rear = capacity - 1; // This is important, see the enqueue
    queue->array = (int*) malloc(queue->capacity * sizeof(int));
    return queue;
}

// Queue is full when size becomes equal to the capacity
int isFull(struct Queue* queue)
{ return (queue->size == queue->capacity); }

// Queue is empty when size is 0
int isEmpty(struct Queue* queue)
{ return (queue->size == 0); }

// Function to add an item to the queue. It changes rear and size
void enqueue(struct Queue* queue, int item)
{
    if (isFull(queue))
        return;
    queue->rear = (queue->rear + 1) % queue->capacity;
    queue->array[queue->rear] = item;
    queue->size = queue->size + 1;
    printf("%d enqueued to queue\n", item);
}

// Function to remove an item from queue. It changes front and size
int dequeue(struct Queue* queue)
{
    if (isEmpty(queue))
        return INT_MIN;
    int item = queue->array[queue->front];
    queue->front = (queue->front + 1) % queue->capacity;
    queue->size = queue->size - 1;
    return item;
}

// Function to get front of queue
int front(struct Queue* queue)
{
    if (isEmpty(queue))
        return INT_MIN;
    return queue->array[queue->front];
}

// Function to get rear of queue
int rear(struct Queue* queue)
{
    if (isEmpty(queue))
        return INT_MIN;
    return queue->array[queue->rear];
}
```

```

}

// Driver program to test above functions.
int main()
{
    struct Queue* queue = createQueue(1000);

    enqueue(queue, 10);
    enqueue(queue, 20);
    enqueue(queue, 30);
    enqueue(queue, 40);

    printf("%d dequeued from queue\n", dequeue(queue));

    printf("Front item is %d\n", front(queue));
    printf("Rear item is %d\n", rear(queue));

    return 0;
}

```

Output:

```

10 enqueued to queue
20 enqueued to queue
30 enqueued to queue
40 enqueued to queue
10 dequeued from queue
Front item is 20
Rear item is 40

```

Time Complexity: Time complexity of all operations like enqueue(), dequeue(), isFull(), isEmpty(), front() and rear() is O(1). There is no loop in any of the operations.

Linked list implementation is easier, it is discussed here: [Queue | Set 2 \(Linked List Implementation\)](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: [Queue](#)

Queue | Set 2 (Linked List Implementation)

In the [previous post](#), we introduced Queue and discussed array implementation. In this post, linked list implementation is discussed. The following two main operations must be implemented efficiently.

In a Queue data structure, we maintain two pointers, *front* and *rear*. The *front* points the first item of queue and *rear* points to last item.

enQueue() This operation adds a new node after *rear* and moves *rear* to the next node.

deQueue() This operation removes the front node and moves *front* to the next node.

```

// A C program to demonstrate linked list based implementation of queue
#include <stdlib.h>
#include <stdio.h>

// A linked list (LL) node to store a queue entry
struct QNode
{
    int key;
    struct QNode *next;
};

// The queue, front stores the front node of LL and rear stores the
// last node of LL
struct Queue
{
    struct QNode *front, *rear;
};

// A utility function to create a new linked list node.
struct QNode* newNode(int k)
{
    struct QNode *temp = (struct QNode*)malloc(sizeof(struct QNode));
    temp->key = k;
    temp->next = NULL;
    return temp;
}

// A utility function to create an empty queue
struct Queue *createQueue()
{
    struct Queue *q = (struct Queue*)malloc(sizeof(struct Queue));
    q->front = q->rear = NULL;
    return q;
}

```

```

}

// The function to add a key k to q
void enqueue(struct Queue *q, int k)
{
    // Create a new LL node
    struct QNode *temp = newNode(k);

    // If queue is empty, then new node is front and rear both
    if (q->rear == NULL)
    {
        q->front = q->rear = temp;
        return;
    }

    // Add the new node at the end of queue and change rear
    q->rear->next = temp;
    q->rear = temp;
}

// Function to remove a key from given queue q
struct QNode *dequeue(struct Queue *q)
{
    // If queue is empty, return NULL.
    if (q->front == NULL)
        return NULL;

    // Store previous front and move front one node ahead
    struct QNode *temp = q->front;
    q->front = q->front->next;

    // If front becomes NULL, then change rear also as NULL
    if (q->front == NULL)
        q->rear = NULL;
    return temp;
}

// Driver Program to test above functions
int main()
{
    struct Queue *q = createQueue();
    enqueue(q, 10);
    enqueue(q, 20);
    dequeue(q);
    dequeue(q);
    enqueue(q, 30);
    enqueue(q, 40);
    enqueue(q, 50);
    struct QNode *n = dequeue(q);
    if (n != NULL)
        printf("Dequeued item is %d", n->key);
    return 0;
}

```

Output:

```
Dequeued item is 30
```

Time Complexity: Time complexity of both operations enqueue() and dequeue() is $O(1)$ as we only change few pointers in both operations. There is no loop in any of the operations.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: [Linked List Queue](#)

Applications of Queue Data Structure

[Queue](#) is used when things don't have to be processed immediately, but have to be processed in **First In First Out** order like [Breadth First Search](#). This property of Queue makes it also useful in following kind of scenarios.

- 1) When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
- 2) When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.

See [this](#) for more detailed applications of Queue and Stack.

References:

<http://introcs.cs.princeton.edu/43stack/>

GATE CS Corner Company Wise Coding Practice

Queue

Priority Queue | Set 1 (Introduction)

Priority Queue is an extension of [queue](#) with following properties.

- 1) Every item has a priority associated with it.
- 2) An element with high priority is dequeued before an element with low priority.
- 3) If two elements have the same priority, they are served according to their order in the queue.

A typical priority queue supports following operations.

insert(item, priority): Inserts an item with given priority.

getHighestPriority(): Returns the highest priority item.

deleteHighestPriority(): Removes the highest priority item.

How to implement priority queue?

Using Array: A simple implementation is to use array of following structure.

```
struct item {
    int item;
    int priority;
}
```

insert() operation can be implemented by adding an item at end of array in $O(1)$ time.

getHighestPriority() operation can be implemented by linearly searching the highest priority item in array. This operation takes $O(n)$ time.

deleteHighestPriority() operation can be implemented by first linearly searching an item, then removing the item by moving all subsequent items one position back.

We can also use Linked List, time complexity of all operations with linked list remains same as array. The advantage with linked list is deleteHighestPriority() can be more efficient as we don't have to move items.

Using Heaps:

Heap is generally preferred for priority queue implementation because heaps provide better performance compared arrays or linked list. In a Binary Heap, getHighestPriority() can be implemented in $O(1)$ time, insert() can be implemented in $O(\log n)$ time and deleteHighestPriority() can also be implemented in $O(\log n)$ time.

With [Fibonacci heap](#), insert() and getHighestPriority() can be implemented in $O(1)$ amortized time and deleteHighestPriority() can be implemented in $O(\log n)$ amortized time.

Applications of Priority Queue:

- 1) CPU Scheduling
- 2) Graph algorithms like Dijkstra's shortest path algorithm, Prim's Minimum Spanning Tree, etc
- 3) All [queue applications](#) where priority is involved.

We will soon be discussing array and heap implementations of priority queue.

References:

http://en.wikipedia.org/wiki/Priority_queue

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: [Queue](#)

Deque | Set 1 (Introduction and Applications)

[Deque](#) or [Double Ended Queue](#) is a generalized version of [Queue data structure](#) that allows insert and delete at both ends.

Operations on Deque:

Mainly the following four basic operations are performed on queue:

insetFront(): Adds an item at the front of Deque.

insertLast(): Adds an item at the rear of Deque.

deleteFront(): Deletes an item from front of Deque.

deleteLast(): Deletes an item from rear of Deque.

In addition to above operations, following operations are also supported

getFront(): Gets the front item from queue.

getRear(): Gets the last item from queue.

isEmpty(): Checks whether Deque is empty or not.

isFull(): Checks whether Deque is full or not.

Applications of Deque:

Since Deque supports both stack and queue operations, it can be used as both. The Deque data structure supports clockwise and anticlockwise rotations in $O(1)$ time which can be useful in certain applications.

Also, the problems where elements need to be removed and or added both ends can be efficiently solved using Deque. For example see [Maximum of all subarrays of size k problem](#)..

See [wiki page](#) for another example of A-Steal job scheduling algorithm where Deque is used as deletions operation is required at both ends.

Language Support:

C++ STL provides implementation of Deque as [std::deque](#) and Java provides [Deque interface](#). See [this](#) for more details.

Implementation:

A Deque can be implemented either using a [doubly linked list](#) or circular array. In both implementation, we can implement all operations in $O(1)$ time. We will soon be discussing C/C++ implementation of Deque Data structure.

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: [Queue](#)

Implement Queue using Stacks

The problem is opposite of [this](#) post. We are given a stack data structure with push and pop operations, the task is to implement a queue using instances of stack data structure and operations on them.

Stack and Queue with insert and delete operations



A queue can be implemented using two stacks. Let queue to be implemented be q and stacks used to implement q be stack1 and stack2. q can be implemented in two ways:

Method 1 (By making enqueue operation costly) This method makes sure that oldest entered element is always at the top of stack 1, so that dequeue operation just pops from stack1. To put the element at top of stack1, stack2 is used.

```
enqueue(q, x)
1) While stack1 is not empty, push everything from stack1 to stack2.
2) Push x to stack1 (assuming size of stacks is unlimited).
3) Push everything back to stack1.

dequeue(q)
1) If stack1 is empty then error
2) Pop an item from stack1 and return it
```

Method 2 (By making dequeue operation costly) In this method, in enqueue operation, the new element is entered at the top of stack1. In dequeue operation, if stack2 is empty then all the elements are moved to stack2 and finally top of stack2 is returned.

```
enqueue(q, x)
1) Push x to stack1 (assuming size of stacks is unlimited).

dequeue(q)
1) If both stacks are empty then error.
2) If stack2 is empty
   While stack1 is not empty, push everything from stack1 to stack2.
3) Pop the element from stack2 and return it.
```

Method 2 is definitely better than method 1.

Method 1 moves all the elements twice in enqueue operation, while method 2 (in enqueue operation) moves the elements once and moves elements only if stack2 empty.

Implementation of method 2:

```
/* Program to implement a queue using two stacks */
#include<stdio.h>
#include<stdlib.h>

/* structure of a stack node */
struct sNode
{
    int data;
    struct sNode *next;
};

/* Function to push an item to stack */
void push(struct sNode** top_ref, int new_data);

/* Function to pop an item from stack */
int pop(struct sNode** top_ref);

/* structure of queue having two stacks */
struct queue
{
    struct sNode *stack1;
    struct sNode *stack2;
};

/* Function to enqueue an item to queue */
void enqueue(struct queue *q, int x)
{
    push(&q->stack1, x);
}

/* Function to dequeue an item from queue */
int dequeue(struct queue *q)
{
    int x;
    /* If both stacks are empty then error */
    if(q->stack1 == NULL && q->stack2 == NULL)
    {
        printf("Q is empty");
        getchar();
        exit(0);
    }

    /* Move elements from stack1 to stack2 only if
    stack2 is empty */
    if(q->stack2 == NULL)
    {
        while(q->stack1 != NULL)
        {
            x = pop(&q->stack1);
            push(&q->stack2, x);
        }
    }

    x = pop(&q->stack2);
    return x;
}

/* Function to push an item to stack */
void push(struct sNode** top_ref, int new_data)
{
    /* allocate node */
    struct sNode* new_node =
        (struct sNode*) malloc(sizeof(struct sNode));
    if(new_node == NULL)
```

```

    {
        printf("Stack overflow \n");
        getchar();
        exit(0);
    }

/* put in the data */
new_node->data = new_data;

/* link the old list off the new node */
new_node->next = (*top_ref);

/* move the head to point to the new node */
(*top_ref) = new_node;
}

/* Function to pop an item from stack*/
int pop(struct sNode** top_ref)
{
    int res;
    struct sNode *top;

    /* If stack is empty then error */
    if(*top_ref == NULL)
    {
        printf("Stack overflow \n");
        getchar();
        exit(0);
    }
    else
    {
        top = *top_ref;
        res = top->data;
        *top_ref = top->next;
        free(top);
        return res;
    }
}

/* Driver function to test above functions */
int main()
{
    /* Create a queue with items 1 2 3*/
    struct queue *q = (struct queue*)malloc(sizeof(struct queue));
    q->stack1 = NULL;
    q->stack2 = NULL;
    enqueue(q, 1);
    enqueue(q, 2);
    enqueue(q, 3);

    /* Dequeue items */
    printf("%d ", dequeue(q));
    printf("%d ", dequeue(q));
    printf("%d ", dequeue(q));

    getchar();
}

```

Output:

```
1 2 3
```

Queue can also be implemented using one user stack and one Function Call Stack. Below is modified Method 2 where recursion (or Function Call Stack) is used to implement queue using only one user defined stack.

```

enqueue(x)
1) Push x to stack1.

deQueue:
1) If stack1 is empty then error.
2) If stack1 has only one element then return it.
3) Recursively pop everything from the stack1, store the popped item
   in a variable res, push the res back to stack1 and return res

```

The step 3 makes sure that the last popped item is always returned and since the recursion stops when there is only one item in *stack1* (step 2), we get the last element of *stack1* in *dequeue()* and all other items are pushed back in step

3. Implementation of method 2 using Function Call Stack:

```

/* Program to implement a queue using one user defined stack
and one Function Call Stack */
#include<stdio.h>
#include<stdlib.h>

/* structure of a stack node */
struct sNode
{
    int data;
    struct sNode *next;
};

/* structure of queue having two stacks */
struct queue
{
    struct sNode *stack1;
};

/* Function to push an item to stack*/
void push(struct sNode** top_ref, int new_data);

/* Function to pop an item from stack*/
int pop(struct sNode** top_ref);

/* Function to enqueue an item to queue */
void enqueue(struct queue *q, int x)
{
    push(&q->stack1, x);
}

/* Function to dequeue an item from queue */
int dequeue(struct queue *q)

```

```

{
    int x, res;

    /* If both stacks are empty then error */
    if(q->stack1 == NULL)
    {
        printf("Q is empty");
        getchar();
        exit(0);
    }
    else if(q->stack1->next == NULL)
    {
        return pop(&q->stack1);
    }
    else
    {
        /* pop an item from the stack1 */
        x = pop(&q->stack1);

        /* store the last dequeued item */
        res = deQueue(q);

        /* push everything back to stack1 */
        push(&q->stack1, x);
        return res;
    }
}

/* Function to push an item to stack */
void push(struct sNode** top_ref, int new_data)
{
    /* allocate node */
    struct sNode* new_node =
        (struct sNode*) malloc(sizeof(struct sNode));

    if(new_node == NULL)
    {
        printf("Stack overflow\n");
        getchar();
        exit(0);
    }

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*top_ref);

    /* move the head to point to the new node */
    (*top_ref) = new_node;
}

/* Function to pop an item from stack */
int pop(struct sNode** top_ref)
{
    int res;
    struct sNode *top;

    /* If stack is empty then error */
    if(*top_ref == NULL)
    {
        printf("Stack overflow\n");
        getchar();
        exit(0);
    }
    else
    {
        top = *top_ref;
        res = top->data;
        *top_ref = top->next;
        free(top);
        return res;
    }
}

/* Driver function to test above functions */
int main()
{
    /* Create a queue with items 1 2 3 */
    struct queue *q = (struct queue*) malloc(sizeof(struct queue));
    q->stack1 = NULL;

    enqueue(q, 1);
    enqueue(q, 2);
    enqueue(q, 3);

    /* Dequeue items */
    printf("%d ", deQueue(q));
    printf("%d ", deQueue(q));
    printf("%d ", deQueue(q));

    getchar();
}

```

Output:

```
1 2 3
```

Please write comments if you find any of the above codes/algorithms incorrect, or find better ways to solve the same problem.

GATE CS Corner **Company Wise Coding Practice**

Queue
Stack

Find the first circular tour that visits all petrol pumps

Suppose there is a circle. There are n petrol pumps on that circle. You are given two sets of data.

1. The amount of petrol that every petrol pump has.
2. Distance from that petrol pump to the next petrol pump.

Calculate the first point from where a truck will be able to complete the circle (The truck will stop at each petrol pump and it has infinite capacity). Expected time complexity is $O(n)$. Assume for 1 litre petrol, the truck can go 1 unit of distance.

For example, let there be 4 petrol pumps with amount of petrol and distance to next petrol pump value pairs as {4, 6}, {6, 5}, {7, 3} and {4, 5}. The first point from where truck can make a circular tour is 2nd petrol pump. Output should be "start = 1" (index of 2nd petrol pump).

A **Simple Solution** is to consider every petrol pumps as starting point and see if there is a possible tour. If we find a starting point with feasible solution, we return that starting point. The worst case time complexity of this solution is $O(n^2)$.

We can **use a Queue** to store the current tour. We first enqueue first petrol pump to the queue, we keep enqueueing petrol pumps till we either complete the tour, or current amount of petrol becomes negative. If the amount becomes negative, then we keep dequeuing petrol pumps till the current amount becomes positive or queue becomes empty.

Instead of creating a separate queue, we use the given array itself as queue. We maintain two index variables start and end that represent rear and front of queue.

C

```
// C program to find circular tour for a truck
#include <stdio.h>

// A petrol pump has petrol and distance to next petrol pump
struct petrolPump
{
    int petrol;
    int distance;
};

// The function returns starting point if there is a possible solution,
// otherwise returns -1
int printTour(struct petrolPump arr[], int n)
{
    // Consider first petrol pump as a starting point
    int start = 0;
    int end = 1;

    int curr_petrol = arr[start].petrol - arr[start].distance;

    /* Run a loop while all petrol pumps are not visited.
    And we have reached first petrol pump again with 0 or more petrol */
    while (end != start || curr_petrol < 0)
    {
        // If current amount of petrol in truck becomes less than 0, then
        // remove the starting petrol pump from tour
        while (curr_petrol < 0 && start != end)
        {
            // Remove starting petrol pump. Change start
            curr_petrol -= arr[start].petrol - arr[start].distance;
            start = (start + 1) % n;

            // If 0 is being considered as start again, then there is no
            // possible solution
            if (start == 0)
                return -1;
        }

        // Add a petrol pump to current tour
        curr_petrol += arr[end].petrol - arr[end].distance;

        end = (end + 1) % n;
    }

    // Return starting point
    return start;
}

// Driver program to test above functions
int main()
{
    struct petrolPump arr[] = {{6, 4}, {3, 6}, {7, 3}};

    int n = sizeof(arr)/sizeof(arr[0]);
    int start = printTour(arr, n);

    (start == -1)? printf("No solution"): printf("Start = %d", start);

    return 0;
}
```

Python

```
# Python program to find circular tour for a track

# A petrol pump has petrol and distance to next petrol pump
class PetrolPump:

    # Constructor to create a new node
    def __init__(self, petrol, distance):
        self.petrol = petrol
        self.distance = distance

# The function return starting point if there is a possible
# solution otherwise returns -1
def printTour(arr):

    n = len(arr)
    # Consider first petrol pump as starting point
    start = 0
    end = 1

    curr_petrol = arr[start].petrol - arr[start].distance

    # Run a loop while all petrol pumps are not visited
    # And we have reached first petrol pump again with 0
    # or more petrol
    while (end != start or curr_petrol < 0):
```

```

# If current amount of petrol pumps are not visited
# And we have reached first petrol pump again with
# 0 or more petrol
while(curr_petrol < 0 and start != end):

    # Remove starting petrol pump. Change start
    curr_petrol -= arr[start].petrol - arr[start].distance
    start = (start + 1) % n

    # If 0 is being considered as start again, then
    # there is no possible solution
    if start == 0:
        return -1

    # Add a petrol pump to current tour
    curr_petrol += arr[end].petrol - arr[end].distance

end = (end + 1) % n

return start

# Driver program to test above function
arr = [PetrolPump(6,4), PetrolPump(3,6), PetrolPump(7,3)]
start = printTour(arr)

print "No solution" if start == -1 else "start =", start

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

```
start = 2
```

Time Complexity: Seems to be more than linear at first look. If we consider the items between start and end as part of a circular queue, we can observe that every item is enqueued at most two times to the queue. The total number of operations is proportional to total number of enqueue operations. Therefore the time complexity is $O(n)$.

Auxiliary Space: $O(1)$

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

[Queue](#)
[Queue](#)
[Stack-Queue](#)
[StackAndQueue](#)

Sliding Window Maximum (Maximum of all subarrays of size k)

Given an array and an integer k, find the maximum for each and every contiguous subarray of size k.

Examples:

Input :
arr[] = {1, 2, 3, 1, 4, 5, 2, 3, 6}
k = 3
Output :
3 3 4 5 5 5 6

Input :
arr[] = {8, 5, 10, 7, 9, 4, 15, 12, 90, 13}
k = 4
Output :
10 10 10 15 15 90 90

We strongly recommend that you click here and practice it, before moving on to the solution.

Method 1 (Simple)

Run two loops. In the outer loop, take all subarrays of size k. In the inner loop, get the maximum of the current subarray.

```

#include<stdio.h>

void printKMax(int arr[], int n, int k)
{
    int j, max;

    for (int i = 0; i <= n-k; i++)
    {
        max = arr[i];

        for (j = i+1; j < i+k; j++)
        {
            if (arr[j] > max)
                max = arr[j];
        }
        printf("%d ", max);
    }
}

int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int n = sizeof(arr)/sizeof(arr[0]);
    int k = 3;
    printKMax(arr, n, k);
    return 0;
}

```

Time Complexity: The outer loop runs $n-k+1$ times and the inner loop runs k times for every iteration of outer loop. So time complexity is $O((n-k+1)*k)$ which can also be written as $O(nk)$.

Method 2 (Use Self-Balancing BST)

- 1) Pick first k elements and create a Self-Balancing Binary Search Tree (BST) of size k.
- 2) Run a loop for $i = 0$ to $n - k$

.....a) Get the maximum element from the BST, and print it.
.....b) Search for arr[i] in the BST and delete it from the BST.
.....c) Insert arr[i+k] into the BST.

Time Complexity: Time Complexity of step 1 is $O(k \log k)$. Time Complexity of steps 2(a), 2(b) and 2(c) is $O(\log k)$. Since steps 2(a), 2(b) and 2(c) are in a loop that runs $n-k+1$ times, time complexity of the complete algorithm is $O(k \log k + (n-k+1) \log k)$ which can also be written as $O(n \log k)$.

Method 3 (A $O(n)$ method: use Dequeue)

We create a [Deque](#), Q of capacity k , that stores only useful elements of current window of k elements. An element is useful if it is in current window and is greater than all other elements on left side of it in current window. We process all array elements one by one and maintain Q to contain useful elements of current window and these useful elements are maintained in sorted order. The element at front of the Q is the largest and element at rear of Q is the smallest of current window. Thanks to [Aashish](#) for suggesting this method.

Following is C++ implementation of this method.

```
#include <iostream>
#include <deque>

using namespace std;

// A Dequeue (Double ended queue) based method for printing maximum element of
// all subarrays of size k
void printKMax(int arr[], int n, int k)
{
    // Create a Double Ended Queue, Qi that will store indexes of array elements
    // The queue will store indexes of useful elements in every window and it will
    // maintain decreasing order of values from front to rear in Qi, i.e.,
    // arr[Qi.front()] to arr[Qi.rear()] are sorted in decreasing order
    std::deque<int> Qi(k);

    /* Process first k (or first window) elements of array */
    int i;
    for (i = 0; i < k; ++i)
    {
        // For very element, the previous smaller elements are useless so
        // remove them from Qi
        while ( (!Qi.empty()) && arr[i] >= arr[Qi.back()])
            Qi.pop_back(); // Remove from rear

        // Add new element at rear of queue
        Qi.push_back(i);
    }

    // Process rest of the elements, i.e., from arr[k] to arr[n-1]
    for (; i < n; ++i)
    {
        // The element at the front of the queue is the largest element of
        // previous window, so print it
        cout << arr[Qi.front()] << " ";

        // Remove the elements which are out of this window
        while ( (!Qi.empty()) && Qi.front() <= i - k )
            Qi.pop_front(); // Remove from front of queue

        // Remove all elements smaller than the currently
        // being added element (remove useless elements)
        while ( (!Qi.empty()) && arr[i] >= arr[Qi.back()])
            Qi.pop_back();

        // Add current element at the rear of Qi
        Qi.push_back(i);
    }

    // Print the maximum element of last window
    cout << arr[Qi.front()];
}

// Driver program to test above functions
int main()
{
    int arr[] = {12, 1, 78, 90, 57, 89, 56};
    int n = sizeof(arr)/sizeof(arr[0]);
    int k = 3;
    printKMax(arr, n, k);
    return 0;
}
```

Output:

```
78 90 90 90 89
```

Time Complexity: $O(n)$. It seems more than $O(n)$ at first look. If we take a closer look, we can observe that every element of array is added and removed at most once. So there are total $2n$ operations.

Auxiliary Space: $O(k)$

Below is an extension of this problem.

[Sum of minimum and maximum elements of all subarrays of size k.](#)

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

[Queue](#)
[Queue](#)
[sliding-window](#)
[Stack-Queue](#)
[StackAndQueue](#)

An Interesting Method to Generate Binary Numbers from 1 to n

Given a number n , write a function that generates and prints all binary numbers with decimal values from 1 to n .

Examples:

```
Input: n = 2
Output: 1, 10

Input: n = 5
Output: 1, 10, 11, 100, 101
```

A simple method is to run a loop from 1 to n , call decimal to binary inside the loop.

Following is an interesting method that uses [queue data structure](#) to print binary numbers. Thanks to [Vivek](#) for suggesting this approach.

- 1) Create an empty queue of strings
- 2) Enqueue the first binary number "1" to queue.
- 3) Now run a loop for generating and printing n binary numbers.
 -a) Dequeue and Print the front of queue.
 -b) Append "0" at the end of front item and enqueue it.
 -c) Append "1" at the end of front item and enqueue it.

Following is C++ implementation of above algorithm.

C++

```
// C++ program to generate binary numbers from 1 to n
#include <iostream>
#include <queue>
using namespace std;

// This function uses queue data structure to print binary numbers
void generatePrintBinary(int n)
{
    // Create an empty queue of strings
    queue<string> q;

    // Enqueue the first binary number
    q.push("1");

    // This loop is like BFS of a tree with 1 as root
    // 0 as left child and 1 as right child and so on
    while (n--)
    {
        // print the front of queue
        string s1 = q.front();
        q.pop();
        cout << s1 << " ";

        string s2 = s1; // Store s1 before changing it

        // Append "0" to s1 and enqueue it
        q.push(s1.append("0"));

        // Append "1" to s2 and enqueue it. Note that s2 contains
        // the previous front
        q.push(s2.append("1"));
    }
}

// Driver program to test above function
int main()
{
    int n = 10;
    generatePrintBinary(n);
    return 0;
}
```

Python

```
# Python program to generate binary numbers from
# 1 to n

# This function uses queue data structure to print binary numbers
def generatePrintBinary(n):

    # Create an empty queue
    from Queue import Queue
    q = Queue()

    # Enqueue the first binary number
    q.put("1")

    # This loop is like BFS of a tree with 1 as root
    # 0 as left child and 1 as right child and so on
    while(n>0):
        n-= 1
        # Print the front of queue
        s1 = q.get()
        print s1

        s2 = s1 # Store s1 before changing it

        # Append "0" to s1 and enqueue it
        q.put(s1+"0")

        # Append "1" to s2 and enqueue it. Note that s2
        # contains the previous front
        q.put(s2+"1")

# Driver program to test above function
n = 10
generatePrintBinary(n)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
1
10
11
100
101
110
111
1000
1001
1010
```

How to efficiently implement k Queues in a single array?

We have discussed [efficient implementation of k stack in an array](#). In this post, same for queue is discussed. Following is the detailed problem statement.

Create a data structure *kQueues* that represents *k* queues. Implementation of *kQueues* should use only one array, i.e., *k* queues should use the same array for storing elements. Following functions must be supported by *kQueues*.

enqueue(int x, int qn) → adds x to queue number 'qn' where qn is from 0 to k-1

dequeue(int qn) → deletes an element from queue number 'qn' where qn is from 0 to k-1

Method 1 (Divide the array in slots of size n/k)

A simple way to implement k queues is to divide the array in k slots of size n/k each, and fix the slots for different queues, i.e., use arr[0] to arr[n/k-1] for first queue, and arr[n/k] to arr[2n/k-1] for queue2 where arr[] is the array to be used to implement two queues and size of array be n.

The problem with this method is inefficient use of array space. An enqueue operation may result in overflow even if there is space available in arr[]. For example, consider k as 2 and array size n as 6. Let we enqueue 3 elements to first and do not enqueue anything to second second queue. When we enqueue 4th element to first queue, there will be overflow even if we have space for 3 more elements in array.

Method 2 (A space efficient implementation)

The idea is similar to the [stack post](#), here we need to use three extra arrays. In stack post, we needed to extra arrays, one more array is required because in queues, enqueue() and dequeue() operations are done at different ends.

Following are the three extra arrays are used:

- 1) **front[]**: This is of size k and stores indexes of front elements in all queues.
- 2) **rear[]**: This is of size k and stores indexes of rear elements in all queues.
- 2) **next[]**: This is of size n and stores indexes of next item for all items in array arr[].

Here arr[] is actual array that stores k stacks.

Together with k queues, a stack of free slots in arr[] is also maintained. The top of this stack is stored in a variable 'free'.

All entries in front[] are initialized as -1 to indicate that all queues are empty. All entries next[i] are initialized as i+1 because all slots are free initially and pointing to next slot. Top of free stack, 'free' is initialized as 0.

Following is C++ implementation of the above idea.

```
// A C++ program to demonstrate implementation of k queues in a single
// array in time and space efficient way
#include<iostream>
#include<climits>
using namespace std;

// A C++ class to represent k queues in a single array of size n
class kQueues
{
    int *arr; // Array of size n to store actual content to be stored in queue
    int *front; // Array of size k to store indexes of front elements of queue
    int *rear; // Array of size k to store indexes of rear elements of queue
    int *next; // Array of size n to store next entry in all queues
    // and free list
    int n, k;
    int free; // To store beginning index of free list
public:
    //constructor to create k queue in an array of size n
    kQueues(int k, int n);

    // A utility function to check if there is space available
    bool isFull() { return (free == -1); }

    // To enqueue an item in queue number 'qn' where qn is from 0 to k-1
    void enqueue(int item, int qn);

    // To dequeue an from queue number 'qn' where qn is from 0 to k-1
    int dequeue(int qn);

    // To check whether queue number 'qn' is empty or not
    bool isEmpty(int qn) { return (front[qn] == -1); }
};

// Constructor to create k queues in an array of size n
kQueues::kQueues(int k1, int n1)
{
    // Initialize n and k, and allocate memory for all arrays
    k = k1, n = n1;
    arr = new int[n];
    front = new int[k];
    rear = new int[k];
    next = new int[n];

    // Initialize all queues as empty
    for (int i = 0; i < k; i++)
        front[i] = -1;

    // Initialize all spaces as free
    free = 0;
    for (int i=0; i<n-1; i++)
        next[i] = i+1;
    next[n-1] = -1; // -1 is used to indicate end of free list
}

// To enqueue an item in queue number 'qn' where qn is from 0 to k-1
void kQueues::enqueue(int item, int qn)
{
    // Overflow check
    if (isFull())
    {
        cout << "nQueue Overflow\n";
        return;
    }

    int i = free; // Store index of first free slot

    // Update index of free slot to index of next slot in free list
    free = next[i];

    if (isEmpty(qn))
        front[qn] = i;
    else
        next[rear[qn]] = i;
```



```

next[i] = -1;

// Update next of rear and then rear for queue number 'qn'
rear[qn] = i;

// Put the item in array
arr[i] = item;
}

// To dequeue an from queue number 'qn' where qn is from 0 to k-1
int kQueues::dequeue(int qn)
{
    // Underflow check
    if (isEmpty(qn))
    {
        cout << "nQueue Underflow\n";
        return INT_MAX;
    }

    // Find index of front item in queue number 'qn'
    int i = front[qn];

    front[qn] = next[i]; // Change top to store next of previous top

    // Attach the previous front to the beginning of free list
    next[i] = free;
    free = i;

    // Return the previous front item
    return arr[i];
}

/* Driver program to test kStacks class */
int main()
{
    // Let us create 3 queue in an array of size 10
    int k = 3, n = 10;
    kQueues ks(k, n);

    // Let us put some items in queue number 2
    ks.enqueue(15, 2);
    ks.enqueue(45, 2);

    // Let us put some items in queue number 1
    ks.enqueue(17, 1);
    ks.enqueue(49, 1);
    ks.enqueue(39, 1);

    // Let us put some items in queue number 0
    ks.enqueue(11, 0);
    ks.enqueue(9, 0);
    ks.enqueue(7, 0);

    cout << "Dequeued element from queue 2 is " << ks.dequeue(2) << endl;
    cout << "Dequeued element from queue 1 is " << ks.dequeue(1) << endl;
    cout << "Dequeued element from queue 0 is " << ks.dequeue(0) << endl;

    return 0;
}

```

Output:

```

Dequeued element from queue 2 is 15
Dequeued element from queue 1 is 17
Dequeued element from queue 0 is 11

```

Time complexities of enqueue() and dequeue() is $O(1)$.

The best part of above implementation is, if there is a slot available in queue, then an item can be enqueued in any of the queues, i.e., no wastage of space. This method requires some extra space. Space may not be an issue because queue items are typically large, for example queues of employees, students, etc where every item is of hundreds of bytes. For such large queues, the extra space used is comparatively very less as we use three integer arrays as extra space.

This article is contributed by **Sachin**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Queue
Queue
Stack-Queue
StackAndQueue

Binary Tree | Set 1 (Introduction)

Trees: Unlike Arrays, Linked Lists, Stack and queues, which are linear data structures, trees are hierarchical data structures.

Tree Vocabulary: The topmost node is called root of the tree. The elements that are directly under an element are called its children. The element directly above something is called its parent. For example, a is a child of f and f is the parent of a. Finally, elements with no children are called leaves.

```

tree
----
j

```

Why Trees?

1. One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer:

```

file system
-----
/

```

2. Trees (with some ordering e.g., BST) provide moderate access/search (quicker than Linked List and slower than arrays).
3. Trees provide moderate insertion/deletion (quicker than Arrays and slower than Unordered Linked Lists).
4. Like Linked Lists and unlike Arrays, Trees don't have an upper limit on number of nodes as nodes are linked using pointers.

Main applications of trees include:

1. Manipulate hierarchical data.
2. Make information easy to search (see tree traversal).

3. Manipulate sorted lists of data.

4. As a workflow for compositing digital images for visual effects.

5. Router algorithms

6. Form of a multi-stage decision-making (see business chess).

Binary Tree: A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.

Binary Tree Representation in C: A tree is represented by a pointer to the topmost node in tree. If the tree is empty, then value of root is NULL.

A Tree node contains following parts.

1. Data

2. Pointer to left child

3. Pointer to right child

In C, we can represent a tree node using structures. Below is an example of a tree node with an integer data.

C

```
struct node
{
    int data;
    struct node *left;
    struct node *right;
};
```

Python

```
# A Python class that represents an individual node
# in a Binary Tree
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key
```

Java

```
/* Class containing left and right child of current
   node and key value*/
class Node
{
    int key;
    Node left, right;

    public Node(int item)
    {
        key = item;
        left = right = null;
    }
}
```

First Simple Tree in C

Let us create a simple tree with 4 nodes in C. The created tree would be as following.

```
tree
----
1
```

C

```
struct node
{
    int data;
    struct node *left;
    struct node *right;
};

/* newNode() allocates a new node with the given data and NULL left and
   right pointers. */
struct node* newNode(int data)
{
    // Allocate memory for new node
```

```

struct node* node = (struct node*)malloc(sizeof(struct node));

// Assign data to this node
node->data = data;

// Initialize left and right children as NULL
node->left = NULL;
node->right = NULL;
return(node);
}

int main()
{
    /*create root*/
    struct node *root = newNode(1);
    /* following is the tree after above statement

    1
   / \
  NULL NULL
  */

    root->left = newNode(2);
    root->right = newNode(3);
    /* 2 and 3 become left and right children of 1
    1
   / \
  2   3
 / \ / \
NULL NULL NULL NULL
  */

    root->left->left = newNode(4);
    /* 4 becomes left child of 2
    1
   / \
  2   3
 / \ / \
4  NULL NULL NULL
/ \
NULL NULL
  */

    getch();
    return 0;
}

```

Python

```

# Python program to introduce Binary Tree

# A class that represents an individual node in a
# Binary Tree
class Node:
    def __init__(self,key):
        self.left = None
        self.right = None
        self.val = key

# create root
root = Node(1)
""" following is the tree after above statement
    1
   / \
  None None
  """

root.left = Node(2);
root.right = Node(3);

""" 2 and 3 become left and right children of 1
    1
   / \
  2   3
 / \ / \
None None None None
  """

root.left.left = Node(4);
"""4 becomes left child of 2
    1
   / \
  2   3
 / \ / \
4  None None None
/ \
None None
  """

```

Java

```

/* Class containing left and right child of current
node and key value*/
class Node
{
    int key;
    Node left, right;
}

```

```

public Node(int item)
{
    key = item;
    left = right = null;
}
}

// A Java program to introduce Binary Tree
class BinaryTree
{
    // Root of Binary Tree
    Node root;

    // Constructors
    BinaryTree(int key)
    {
        root = new Node(key);
    }

    BinaryTree()
    {
        root = null;
    }

    public static void main(String[] args)
    {
        BinaryTree tree = new BinaryTree();

        /*create root*/
        tree.root = new Node(1);

        /* following is the tree after above statement

            1
           / \
          null null */

        tree.root.left = new Node(2);
        tree.root.right = new Node(3);

        /* 2 and 3 become left and right children of 1

            1
           / \
          2   3
         / \ / \
        null null null null */

        tree.root.left.left = new Node(4);
        /* 4 becomes left child of 2

            1
           / \
          2   3
         / \ / \
        4  null null null
       / \
      null null
     */
    }
}

```

Summary: Tree is a hierarchical data structure. Main uses of trees include maintaining hierarchical data, providing moderate access and insert/delete operations. Binary trees are special cases of tree where every node has at mc

Below are set 2 and set 3 of this post.

[Properties of Binary Tree](#)

[Types of Binary Tree](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

Binary Tree | Set 2 (Properties)

We have discussed [Introduction to Binary Tree](#) in set 1. In this post, properties of binary are discussed.

1) The maximum number of nodes at level 'l' of a binary tree is 2^{l-1} .

Here level is number of nodes on path from root to the node (including root and node). Level of root is 1.

This can be proved by induction.

For root, $l = 1$, number of nodes = $2^{1-1} = 1$

Assume that maximum number of nodes on level l is 2^{l-1}

Since in Binary tree every node has at most 2 children, next level would have twice nodes, i.e. $2 * 2^{l-1}$

2) Maximum number of nodes in a binary tree of height 'h' is $2^h - 1$.

Here height of a tree is maximum number of nodes on root to leaf path. Height of a leaf node is considered as 1.

This result can be derived from point 2 above. A tree has maximum nodes if all levels have maximum nodes. So maximum number of nodes in a binary tree of height h is $1 + 2 + 4 + \dots + 2^{h-1}$. This is a simple geometric series with h terms and sum of this series is $2^h - 1$.

In some books, height of a leaf is considered as 0. In this convention, the above formula becomes $2^{h+1} - 1$

3) In a Binary Tree with N nodes, minimum possible height or minimum number of levels is $\lceil \log_2(N+1) \rceil$

This can be directly derived from point 2 above. If we consider the convention where height of a leaf node is considered as 0, then above formula for minimum possible height becomes $\lceil \log_2(N+1) \rceil - 1$

4) A Binary Tree with L leaves has at least $\lceil \log_2 L \rceil + 1$ levels

A Binary tree has maximum number of leaves when all levels are fully filled. Let all leaves be at level l , then below is true for number of leaves L .

$L \leq 2^{l-1}$ [From Point 1]
 $\log_2 L = \lceil \log_2 L \rceil + 1$

5) In Binary tree, number of leaf nodes is always one more than nodes with two children.

$L = T + 1$
Where L = Number of leaf nodes
 T = Number of internal nodes with two children

See [Handshaking Lemma](#) and [Tree](#) for proof.

In the next article on tree series, we will be discussing [different types of Binary Trees and their properties](#).

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

Binary Tree | Set 3 (Types of Binary Tree)

We have discussed [Introduction to Binary Tree](#) in set 1 and [Properties of Binary Tree](#) in Set 2. In this post, common types of binary is discussed.

Following are common types of Binary Trees.

Full Binary Tree A Binary Tree is full if every node has 0 or 2 children. Following are examples of full binary tree.

```
      18
     /  \
    15   30
   /\   /\
  40 50 100 40
```

```
      18
     /  \
    15   20
   /\   /\
  40  50
 /\ 
30 50
```

```
      18
     /  \
    40   30
     /\ 
    100 40
```

In a Full Binary, number of leaf nodes is number of internal nodes plus 1

$L = I + 1$

Where L = Number of leaf nodes, I = Number of internal nodes

See [Handshaking Lemma](#) and [Tree](#) for proof.

Complete Binary Tree: A Binary Tree is complete Binary Tree if all levels are completely filled except possibly the last level and the last level has all keys as left as possible

Following are examples of Complete Binary Trees

```
      18
     /  \
    15   30
   /\   /\
  40 50 100 40
```

```
      18
     /  \
    15   30
   /\   /\
```

```

40  50  100  40
 /  \  /  \
8   7 9

```

Practical example of Complete Binary Tree is [Binary Heap](#).

Perfect Binary Tree A Binary tree is Perfect Binary Tree in which all internal nodes have two children and all leaves are at same level. Following are examples of Perfect Binary Trees.

```

      18
     /  \
    15   30
   /\  /\
  40 50 100 40

```

```

      18
     /  \
    15   30

```

A Perfect Binary Tree of height h (where height is number of nodes on path from root to leaf) has $2^h - 1$ node.

Example of Perfect binary tree is ancestors in family. Keep a person at root, parents as children, parents of parents as their children.

Balanced Binary Tree

A binary tree is balanced if height of the tree is $O(\log n)$ where n is number of nodes. For Example, AVL tree maintain $O(\log n)$ height by making sure that the difference between heights of left and right subtrees is 1. Red-Black trees maintain $O(\log n)$ height by making sure that the number of Black nodes on every root to leaf paths are same and there are no adjacent red nodes. Balanced Binary Search trees are performance wise good as they provide $O(\log n)$ time for search, insert and delete.

A degenerate (or pathological) tree A Tree where every internal node has one child. Such trees are performance-wise same as linked list.

```

10
 /
20
 \
30
 \
40

```

Source:

https://en.wikipedia.org/wiki/Binary_tree#Types_of_binary_trees

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: [Tree](#)

Handshaking Lemma and Interesting Tree Properties

What is Handshaking Lemma?

Handshaking lemma is about undirected graph. In every finite undirected graph number of vertices with odd degree is always even. The handshaking lemma is a consequence of the degree sum formula (also sometimes called the handshaking lemma)



How is Handshaking Lemma useful in Tree Data structure?

Following are some interesting facts that can be proved using Handshaking lemma.

1) In a k -ary tree where every node has either 0 or k children, following property is always true.

$$L = (k - 1) * I + 1$$

Where L = Number of leaf nodes
 I = Number of internal nodes

Proof:

Proof can be divided in two cases.

Case 1 (Root is Leaf): There is only one node in tree. The above formula is true for single node as $L = 1$, $I = 0$.

Case 2 (Root is Internal Node): For trees with more than 1 nodes, root is always internal node. The above formula can be proved using Handshaking Lemma for this case. A tree is an undirected acyclic graph.

Total number of edges in Tree is number of nodes minus 1, i.e., $|E| = L + I - 1$.

All internal nodes except root in the given type of tree have degree $k + 1$. Root has degree k . All leaves have degree 1. Applying the Handshaking lemma to such trees, we get following relation.

Sum of all degrees = $2 * (\text{Sum of Edges})$

Sum of degrees of leaves +
Sum of degrees for Internal Node except root +
Root's degree = $2 * (\text{No. of nodes} - 1)$

Putting values of above terms,
 $L + (I - 1) * (k + 1) + k = 2 * (L + I - 1)$
 $L + k * I - k + I - 1 + k = 2 * L + 2 * I - 2$
 $L + k * I + I - 1 = 2 * L + 2 * I - 2$
 $k * I + 1 - I = L$
 $(k - 1) * I + 1 = L$

So the above property is proved using Handshaking Lemma, let us discuss one more interesting property.

2) In Binary tree, number of leaf nodes is always one more than nodes with two children.

$$L = T + 1$$

Where L = Number of leaf nodes
 T = Number of internal nodes with two children

Proof:

Let number of nodes with 2 children be T. Proof can be divided in three cases.

Case 1: There is only one node, the relationship holds

as $T = 0$, $L = 1$.

Case 2: Root has two children, i.e., degree of root is 2.

Sum of degrees of nodes with two children except root +
Sum of degrees of nodes with one child +
Sum of degrees of leaves + Root's degree = $2 * (\text{No. of Nodes} - 1)$

Putting values of above terms,
 $(T-1)*3 + S*2 + L + 2 = (S + T + L - 1)*2$

Cancelling 2S from both sides.

$(T-1)*3 + L + 2 = (S + L - 1)*2$

$T - 1 = L - 2$

$T = L - 1$

Case 3: Root has one child, i.e., degree of root is 1.

Sum of degrees of nodes with two children +
Sum of degrees of nodes with one child except root +
Sum of degrees of leaves + Root's degree = $2 * (\text{No. of Nodes} - 1)$

Putting values of above terms,
 $T*3 + (S-1)*2 + L + 1 = (S + T + L - 1)*2$

Cancelling 2S from both sides.

$3*T + L - 1 = 2*T + 2*L - 2$

$T - 1 = L - 2$

$T = L - 1$

Therefore, in all three cases, we get $T = L - 1$.

We have discussed proof of two important properties of Trees using Handshaking Lemma. Many GATE questions have been asked on these properties, following are few links.

[GATE-CS-2015 \(Set 3\) | Question 35](#)

[GATE-CS-2015 \(Set 2\) | Question 20](#)

[GATE-CS-2005 | Question 36](#)

[GATE-CS-2002 | Question 34](#)

[GATE-CS-2007 | Question 43](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Trees

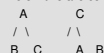
Enumeration of Binary Trees

A Binary Tree is labeled if every node is assigned a label and a Binary Tree is unlabeled if nodes are not assigned any label.

Below two are considered same unlabeled trees



Below two are considered different labeled trees



How many different Unlabeled Binary Trees can be there with n nodes?

For $n = 1$, there is only one tree



For $n = 2$, there are two trees



For $n = 3$, there are five trees



The idea is to consider all possible pair of counts for nodes in left and right subtrees and multiply the counts for a particular pair. Finally add results of all pairs.

For example, let $T(n)$ be count for n nodes.

$T(0) = 1$ [There is only 1 empty tree]

$T(1) = 1$

$T(2) = 2$

$T(3) = T(0)*T(2) + T(1)*T(1) + T(2)*T(0) = 1*2 + 1*1 + 2*1 = 5$

$T(4) = T(0)*T(3) + T(1)*T(2) + T(2)*T(1) + T(3)*T(0)$

$= 1*5 + 1*2 + 2*1 + 5*1$

$= 14$

The above pattern basically represents n^{th} Catalan Numbers. First few catalan numbers are 1 1 2 5 14 42 132 429 1430 4862,...

catalan



Here,

$T(i-1)$ represents number of nodes on the left-sub-tree

$T(n-i-1)$ represents number of nodes on the right-sub-tree

n^{th} Catalan Number can also be evaluated using direct formula.

$$T(n) = (2n)! / ((n+1)!n!)$$

Number of Binary Search Trees (BST) with n nodes is also same as number of unlabeled trees. The reason for this is simple, in BST also we can make any key as root. If root is i 'th key in sorted order, then $i-1$ keys can go on one side and $(n-i)$ keys can go on other side.

How many labeled Binary Trees can be there with n nodes?

To count labeled trees, we can use above count for unlabeled trees. The idea is simple, every unlabeled tree with n nodes can create $n!$ different labeled trees by assigning different permutations of labels to all nodes.

Therefore,

$$\begin{aligned}\text{Number of Labeled Trees} &= (\text{Number of unlabeled trees}) \times n! \\ &= [(2n)! / ((n+1)!n!)] \times n!\end{aligned}$$

For example for $n = 3$, there are $5 \times 3! = 5 \times 6 = 30$ different labeled trees

Reference:

<http://qa.geeksforgeeks.org/4343/many-labeled-unlabeled-binary-trees-are-possible-with-nodes/>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: [Tree](#)

Applications of tree data structure

Difficulty Level: Rookie

Why Tree?

Unlike Array and Linked List, which are linear data structures, tree is hierarchical (or non-linear) data structure.

1) One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer:

file system

- 1) If we organize keys in form of a tree (with some ordering e.g., BST), we can search for a given key in moderate time (quicker than Linked List and slower than arrays). [Self-balancing search trees](#) like AVL and [Red-Black trees](#) guarantee
- 2) We can insert/delete keys in moderate time (quicker than Arrays and slower than Unordered Linked Lists). [Self-balancing search trees](#) like AVL and [Red-Black trees](#) guarantee an upper bound of $O(\log n)$ for insertion/deletion.
- 3) Like Linked Lists and unlike Arrays, Pointer implementation of trees don't have an upper limit on number of nodes as nodes are linked using pointers.

[As per Wikipedia](#), following are the common uses of tree.

1. Manipulate hierarchical data.
2. Make information easy to search (see tree traversal).
3. Manipulate sorted lists of data.
4. As a workflow for compositing digital images for visual effects.
5. Router algorithms

References:

<http://www.cs.bu.edu/teaching/c/tree/binary/>

[http://en.wikipedia.org/wiki/Tree_\(data_structure\)#Common_uses](http://en.wikipedia.org/wiki/Tree_(data_structure)#Common_uses)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner

Company Wise Coding Practice

Tree Traversals (Inorder, Preorder and Postorder)

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.

Example Tree



Example Tree

Depth First Traversals:

- (a) Inorder (Left, Root, Right) : 4 2 5 1 3
- (b) Preorder (Root, Left, Right) : 1 2 4 5 3
- (c) Postorder (Left, Right, Root) : 4 5 2 3 1

Breadth First or Level Order Traversal : 1 2 3 4 5

Please see [this](#) post for Breadth First Traversal.

Inorder Traversal:

Algorithm Inorder(tree)

1. Traverse the left subtree, i.e., call Inorder(left-subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right-subtree)

Uses of Inorder

In case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal is reversed, can be used.

Example: Inorder traversal for the above given figure is 4 2 5 1 3.

Practice Inorder Traversal

Preorder Traversal:

Algorithm Preorder(tree)

1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)

Uses of Preorder

Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expression on of an expression tree. Please see http://en.wikipedia.org/wiki/Polish_notation to know why prefix expressions are useful.

Example: Preorder traversal for the above given figure is 1 2 4 5 3.

Practice Preorder Traversal

Postorder Traversal:

Algorithm Postorder(tree)

1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root.

Uses of Postorder

Postorder traversal is used to delete the tree. Please see the [question for deletion of tree](#) for details. Postorder traversal is also useful to get the postfix expression of an expression tree. Please see http://en.wikipedia.org/wiki/Reverse_Polish_notation to for the usage of postfix expression.

Practice Postorder Traversal

Example: Postorder traversal for the above given figure is 4 5 2 3 1.

C

```
// C program for different tree traversals
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Given a binary tree, print its nodes according to the
   "bottom-up" postorder traversal. */
void printPostorder(struct node* node)
{
    if (node == NULL)
        return;

    // first recur on left subtree
    printPostorder(node->left);

    // then recur on right subtree
    printPostorder(node->right);

    // now deal with the node
    printf("%d ", node->data);
}

/* Given a binary tree, print its nodes in inorder*/
```

```

void printInorder(struct node* node)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    printInorder(node->left);

    /* then print the data of node */
    printf("%d ", node->data);

    /* now recur on right child */
    printInorder(node->right);
}

/* Given a binary tree, print its nodes in preorder*/
void printPreorder(struct node* node)
{
    if (node == NULL)
        return;

    /* first print data of node */
    printf("%d ", node->data);

    /* then recur on left subtree */
    printPreorder(node->left);

    /* now recur on right subtree */
    printPreorder(node->right);
}

/* Driver program to test above functions*/
int main()
{
    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    printf("\nPreorder traversal of binary tree is \n");
    printPreorder(root);

    printf("\nInorder traversal of binary tree is \n");
    printInorder(root);

    printf("\nPostorder traversal of binary tree is \n");
    printPostorder(root);

    getchar();
    return 0;
}

```

Python

```

# Python program to for tree traversals

# A class that represents an individual node in a
# Binary Tree
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

# A function to do inorder tree traversal
def printInorder(root):

    if root:

        # First recur on left child
        printInorder(root.left)

        # then print the data of node
        print(root.val),

        # now recur on right child
        printInorder(root.right)

# A function to do postorder tree traversal
def printPostorder(root):

    if root:

        # First recur on left child
        printPostorder(root.left)

        # the recur on right child
        printPostorder(root.right)

        # now print the data of node
        print(root.val),

# A function to do postorder tree traversal
def printPreorder(root):

    if root:

        # First print the data of node
        print(root.val),

        # Then recur on left child
        printPreorder(root.left)

        # Finally recur on right child
        printPreorder(root.right)

# Driver code

```

```

root = Node(1)
root.left  = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
print "Preorder traversal of binary tree is"
printPreorder(root)

print "\nInorder traversal of binary tree is"
printInorder(root)

print "\nPostorder traversal of binary tree is"
printPostorder(root)

```

Java

```

// Java program for different tree traversals

/* Class containing left and right child of current
node and key value*/
class Node
{
    int key;
    Node left, right;

    public Node(int item)
    {
        key = item;
        left = right = null;
    }
}

class BinaryTree
{
    // Root of Binary Tree
    Node root;

    BinaryTree()
    {
        root = null;
    }

    /* Given a binary tree, print its nodes according to the
    "bottom-up" postorder traversal. */
    void printPostorder(Node node)
    {
        if (node == null)
            return;

        // first recur on left subtree
        printPostorder(node.left);

        // then recur on right subtree
        printPostorder(node.right);

        // now deal with the node
        System.out.print(node.key + " ");
    }

    /* Given a binary tree, print its nodes in inorder*/
    void printInorder(Node node)
    {
        if (node == null)
            return;

        /* first recur on left child */
        printInorder(node.left);

        /* then print the data of node */
        System.out.print(node.key + " ");

        /* now recur on right child */
        printInorder(node.right);
    }

    /* Given a binary tree, print its nodes in preorder*/
    void printPreorder(Node node)
    {
        if (node == null)
            return;

        /* first print data of node */
        System.out.print(node.key + " ");

        /* then recur on left subtree */
        printPreorder(node.left);

        /* now recur on right subtree */
        printPreorder(node.right);
    }

    // Wrappers over above recursive functions
    void printPostorder() { printPostorder(root); }
    void printInorder() { printInorder(root); }
    void printPreorder() { printPreorder(root); }

    // Driver method
    public static void main(String[] args)
    {
        BinaryTree tree = new BinaryTree();
        tree.root = new Node(1);
        tree.root.left = new Node(2);
        tree.root.right = new Node(3);
        tree.root.left.left = new Node(4);
        tree.root.left.right = new Node(5);

        System.out.println("Preorder traversal of binary tree is ");
        tree.printPreorder();

        System.out.println("\nInorder traversal of binary tree is ");
        tree.printInorder();

        System.out.println("\nPostorder traversal of binary tree is ");

```

```

    tree.printPostorder();
}
}

```

Output:

```

Preorder traversal of binary tree is
1 2 4 5 3
Inorder traversal of binary tree is
4 2 5 1 3
Postorder traversal of binary tree is
4 5 2 3 1

```

One more example:

tree-traversal



Image Source : https://www.cs.swarthmore.edu/~newhall/unixhelp/Java_bst.pdf

Time Complexity: $O(n)$

Let us see different corner cases.

Complexity function $T(n)$ — for all problem where tree traversal is involved — can be defined as:

$$T(n) = T(k) + T(n - k - 1) + c$$

Where k is the number of nodes on one side of root and $n-k-1$ on the other side.

Let's do analysis of boundary conditions

Case 1: Skewed tree (One of the subtrees is empty and other subtree is non-empty)

k is 0 in this case.

$$T(n) = T(0) + T(n-1) + c$$

$$T(n) = 2T(0) + T(n-2) + 2c$$

$$T(n) = 3T(0) + T(n-3) + 3c$$

$$T(n) = 4T(0) + T(n-4) + 4c$$

.....

$$T(n) = (n-1)T(0) + T(1) + (n-1)c$$

$$T(n) = nT(0) + (n)c$$

Value of $T(0)$ will be some constant say d . (traversing a empty tree will take some constants time)

$$T(n) = n(c+d)$$

$$T(n) = O(n) \text{ (Theta of } n)$$

Case 2: Both left and right subtrees have equal number of nodes.

$$T(n) = 2T(\lfloor n/2 \rfloor) + c$$

This recursive function is in the standard form $(T(n) = aT(n/b) + (-)(n))$ for master method http://en.wikipedia.org/wiki/Master_theorem. If we solve it by master method we get $(-)(n)$

Auxiliary Space : If we don't consider size of stack for function calls then $O(1)$ otherwise $O(n)$.

GATE CS Corner Company Wise Coding Practice

Trees
Inorder Traversal
PostOrder Traversal
Preorder Traversal
Tree Traversal
Trees
Tutorial

BFS vs DFS for Binary Tree

What are BFS and DFS for Binary Tree?

A Tree is typically traversed in two ways:

- **Breadth First Traversal** (Or Level Order Traversal)
- **Depth First Traversals**
 - Inorder Traversal (Left-Root-Right)
 - Preorder Traversal (Root-Left-Right)
 - Postorder Traversal (Left-Right-Root)

Example Tree



BFS and DFSs of above Tree

Breadth First Traversal : 1 2 3 4 5

Depth First Traversals:

Preorder Traversal : 1 2 4 5 3

Inorder Traversal : 4 2 5 1 3

Postorder Traversal : 4 5 2 3 1

Why do we care?

There are many tree questions that can be solved using any of the above four traversals. Examples of such questions are [size](#), [maximum](#), [minimum](#), [print left view](#), etc.

Is there any difference in terms of Time Complexity?

All four traversals require $O(n)$ time as they visit every node exactly once.

Is there any difference in terms of Extra Space?

There is difference in terms of extra space required.

1. Extra Space required for Level Order Traversal is $O(w)$ where w is maximum width of Binary Tree. In level order traversal, queue one by one stores nodes of different level.
2. Extra Space required for Depth First Traversals is $O(h)$ where h is maximum height of Binary Tree. In Depth First Traversals, stack (or function call stack) stores all ancestors of a node.

Maximum Width of a Binary Tree at depth (or height) h can be 2^h where h starts from 0. So the maximum number of nodes can be at the last level. And worst case occurs when Binary Tree is a perfect Binary Tree with numbers of nodes like 1, 3, 7, 15, ...etc. In worst case, value of 2^h is **Ceil($n/2$)**.

Height for a Balanced Binary Tree is $O(\log n)$. Worst case occurs for skewed tree and worst case height becomes $O(n)$.

So in worst case extra space required is $O(n)$ for both. But worst cases occur for different types of trees.

It is evident from above points that extra space required for Level order traversal is likely to be more when tree is more balanced and extra space for Depth First Traversal is likely to be more when tree is less balanced.

How to Pick One?

1. Extra Space can be one factor (Explained above)
2. Depth First Traversals are typically recursive and recursive code requires function call overheads.
3. The most important points is, BFS starts visiting nodes from root while DFS starts visiting nodes from leaves. So if our problem is to search something that is more likely to be closer to root, we would prefer BFS. And if the target node is close to a leaf, we would prefer DFS.

Exercise:

Which traversal should be used to print leaves of Binary Tree and why?

Which traversal should be used to print nodes at k 'th level where k is much less than total number of levels?

This article is contributed by **Dheeraj Gupta**. This Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Trees
BFS
DFS

Level Order Tree Traversal

Level order traversal of a tree is **breadth first traversal** for the tree.

Example Tree



Example Tree

Level order traversal of the above tree is 1 2 3 4 5

METHOD 1 (Use function to print a given level)

Algorithm:

There are basically two functions in this method. One is to print all nodes at a given level (`printGivenLevel`), and other is to print level order traversal of the tree (`printLevelorder`). `printLevelorder` makes use of `printGivenLevel` to print nodes at all levels one by one starting from root.

```
/*Function to print level order traversal of tree*/
printLevelorder(tree)
for d = 1 to height(tree)
    printGivenLevel(tree, d);

/*Function to print all nodes at a given level*/
printGivenLevel(tree, level)
if tree is NULL then return;
if level is 1, then
    print(tree->data);
else if level greater than 1, then
    printGivenLevel(tree->left, level-1);
    printGivenLevel(tree->right, level-1);
```

Implementation:

C

```
// Recursive C program for level order traversal of Binary Tree
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct node
{
    int data;
    struct node* left, *right;
};

/* Function prototypes */
void printGivenLevel(struct node* root, int level);
int height(struct node* node);
struct node* newNode(int data);

/* Function to print level order traversal a tree*/
void printLevelOrder(struct node* root)
{
    int h = height(root);
    int i;
    for (i=1; i<=h; i++)
        printGivenLevel(root, i);
}

/* Print nodes at a given level */
void printGivenLevel(struct node* root, int level)
{
    if (root == NULL)
        return;
    if (level == 1)
        printf("%d ", root->data);
    else if (level > 1)
    {
        printGivenLevel(root->left, level-1);
        printGivenLevel(root->right, level-1);
    }
}

/* Compute the "height" of a tree -- the number of
nodes along the longest path from the root node
down to the farthest leaf node.*/
int height(struct node* node)
{
    if (node==NULL)
        return 0;
    else
    {
        /* compute the height of each subtree */
        int lheight = height(node->left);
        int rheight = height(node->right);

        /* use the larger one */
        if (lheight > rheight)
            return(lheight+1);
        else return(rheight+1);
    }
}

/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Driver program to test above functions*/
int main()
{
    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    printf("Level Order traversal of binary tree is \n");
    printLevelOrder(root);

    return 0;
}
```

Java

```
// Recursive Java program for level order traversal of Binary Tree

/* Class containing left and right child of current
node and key value*/
class Node
{
    int data;
    Node left, right;
    public Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    // Root of the Binary Tree
    Node root;
```

```

public BinaryTree()
{
    root = null;
}

/* function to print level order traversal of tree*/
void printLevelOrder()
{
    int h = height(root);
    int i;
    for (i=1; i<=h; i++)
        printGivenLevel(root, i);
}

/* Compute the "height" of a tree -- the number of
nodes along the longest path from the root node
down to the farthest leaf node.*/
int height(Node root)
{
    if (root == null)
        return 0;
    else
    {
        /* compute height of each subtree */
        int lheight = height(root.left);
        int rheight = height(root.right);

        /* use the larger one */
        if (lheight > rheight)
            return (lheight+1);
        else return (rheight+1);
    }
}

/* Print nodes at the given level */
void printGivenLevel (Node root,int level)
{
    if (root == null)
        return;
    if (level == 1)
        System.out.print(root.data + " ");
    else if (level > 1)
    {
        printGivenLevel(root.left, level-1);
        printGivenLevel(root.right, level-1);
    }
}

/* Driver program to test above functions */
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();
    tree.root= new Node(1);
    tree.root.left= new Node(2);
    tree.root.right= new Node(3);
    tree.root.left.left= new Node(4);
    tree.root.left.right= new Node(5);

    System.out.println("Level order traversal of binary tree is ");
    tree.printLevelOrder();
}
}

```

Python

```

# Recursive Python program for level order traversal of Binary Tree

# A node structure
class Node:

    # A utility function to create a new node
    def __init__(self, key):
        self.data = key
        self.left = None
        self.right = None

# Function to print level order traversal of tree
def printLevelOrder(root):
    h = height(root)
    for i in range(1, h+1):
        printGivenLevel(root, i)

# Print nodes at a given level
def printGivenLevel(root , level):
    if root is None:
        return
    if level == 1:
        print "%d" %(root.data),
    elif level > 1 :
        printGivenLevel(root.left , level-1)
        printGivenLevel(root.right , level-1)

""" Compute the height of a tree--the number of nodes
along the longest path from the root node down to
the farthest leaf node
"""
def height(node):
    if node is None:
        return 0
    else :
        # Compute the height of each subtree
        lheight = height(node.left)
        rheight = height(node.right)

        #Use the larger one
        if lheight > rheight :
            return lheight+1
        else:
            return rheight+1

```

```
# Driver program to test above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)

print "Level order traversal of binary tree is -"
printLevelOrder(root)

#This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
Level order traversal of binary tree is -
1 2 3 4 5
```

Time Complexity: $O(n^2)$ in worst case. For a skewed tree, printGivenLevel() takes $O(n)$ time where n is the number of nodes in the skewed tree. So time complexity of printLevelOrder() is $O(n) + O(n-1) + O(n-2) + \dots + O(1)$ which is $O(n^2)$.

METHOD 2 (Use Queue)

Algorithm:

For each node, first the node is visited and then it's child nodes are put in a FIFO queue.

```
printLevelorder(tree)
1) Create an empty queue q
2) temp_node = root //start from root/
3) Loop while temp_node is not NULL
   a) print temp_node->data.
   b) Enqueue temp_node's children (first left then right children) to q
   c) Dequeue a node from q and assign it's value to temp_node
```

Implementation:

Here is a simple implementation of the above algorithm. Queue is implemented using an array with maximum size of 500. We can implement queue as linked list also.

C

```
// Iterative Queue based C program to do level order traversal
// of Binary Tree
#include <stdio.h>
#include <stdlib.h>
#define MAX_Q_SIZE 500

/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* function prototypes */
struct node** createQueue(int *, int *);
void enqueue(struct node **, int *, struct node *);
struct node *dequeue(struct node **, int *);

/* Given a binary tree, print its nodes in level order
using array for implementing queue */
void printLevelOrder(struct node* root)
{
    int rear, front;
    struct node **queue = createQueue(&front, &rear);
    struct node *temp_node = root;

    while (temp_node)
    {
        printf("%d ", temp_node->data);

        /* Enqueue left child */
        if (temp_node->left)
            enqueue(queue, &rear, temp_node->left);

        /* Enqueue right child */
        if (temp_node->right)
            enqueue(queue, &rear, temp_node->right);

        /* Dequeue node and make it temp_node */
        temp_node = dequeue(queue, &front);
    }
}

/* UTILITY FUNCTIONS */
struct node** createQueue(int *front, int *rear)
{
    struct node **queue =
        (struct node **)malloc(sizeof(struct node *)*MAX_Q_SIZE);

    *front = *rear = 0;
    return queue;
}

void enqueue(struct node **queue, int *rear, struct node *new_node)
{
    queue[*rear] = new_node;
    (*rear)++;
}

struct node *dequeue(struct node **queue, int *front)
{
    (*front)++;
    return queue[*front - 1];
}

/* Helper function that allocates a new node with the
```



```

given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Driver program to test above functions */
int main()
{
    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    printf("Level Order traversal of binary tree is \n");
    printLevelOrder(root);

    return 0;
}

```

C++

```

/* C++ program to print level order traversal using STL */
#include <iostream>
#include <queue>
using namespace std;

// A Binary Tree Node
struct Node
{
    int data;
    struct Node *left, *right;
};

// Iterative method to find height of Binary Tree
void printLevelOrder(Node *root)
{
    // Base Case
    if (root == NULL) return;

    // Create an empty queue for level order traversal
    queue<Node *> q;

    // Enqueue Root and initialize height
    q.push(root);

    while (q.empty() == false)
    {
        // Print front of queue and remove it from queue
        Node *node = q.front();
        cout << node->data << " ";
        q.pop();

        /* Enqueue left child */
        if (node->left != NULL)
            q.push(node->left);

        /* Enqueue right child */
        if (node->right != NULL)
            q.push(node->right);
    }
}

// Utility function to create a new tree node
Node* newNode(int data)
{
    Node *temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Driver program to test above functions
int main()
{
    // Let us create binary tree shown in above diagram
    Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    cout << "Level Order traversal of binary tree is \n";
    printLevelOrder(root);
    return 0;
}

```

Java

```

// Iterative Queue based Java program to do level order traversal
// of Binary Tree

/* importing the inbuilt java classes required for the program */
import java.util.Queue;
import java.util.LinkedList;

/* Class to represent Tree node */
class Node {
    int data;
    Node left, right;

    public Node(int item) {
        data = item;
        left = null;
    }
}

```

```

        right = null;
    }
}

/* Class to print Level Order Traversal */
class BinaryTree {

    Node root;

    /* Given a binary tree. Print its nodes in level order
    using array for implementing queue */
    void printLevelOrder()
    {
        Queue<Node> queue = new LinkedList<Node>();
        queue.add(root);
        while (!queue.isEmpty())
        {

            /* poll() removes the present head.
            For more information on poll() visit
            http://www.tutorialspoint.com/java/util/linkedlist_poll.htm */
            Node tempNode = queue.poll();
            System.out.print(tempNode.data + " ");

            /* Enqueue left child */
            if (tempNode.left != null) {
                queue.add(tempNode.left);
            }

            /* Enqueue right child */
            if (tempNode.right != null) {
                queue.add(tempNode.right);
            }
        }
    }

    public static void main(String args[])
    {
        /* creating a binary tree and entering
        the nodes */
        BinaryTree tree_level = new BinaryTree();
        tree_level.root = new Node(1);
        tree_level.root.left = new Node(2);
        tree_level.root.right = new Node(3);
        tree_level.root.left.left = new Node(4);
        tree_level.root.left.right = new Node(5);

        System.out.println("Level order traversal of binary tree is - ");
        tree_level.printLevelOrder();
    }
}

```

Python

```

# Python program to print level order traversal using Queue

# A node structure
class Node:
    # A utility function to create a new node
    def __init__(self, key):
        self.data = key
        self.left = None
        self.right = None

# Iterative Method to print the height of binary tree
def printLevelOrder(root):
    # Base Case
    if root is None:
        return

    # Create an empty queue for level order traversal
    queue = []

    # Enqueue Root and initialize height
    queue.append(root)

    while(len(queue) > 0):
        # Print front of queue and remove it from queue
        print queue[0].data,
        node = queue.pop(0)

        # Enqueue left child
        if node.left is not None:
            queue.append(node.left)

        # Enqueue right child
        if node.right is not None:
            queue.append(node.right)

#Driver Program to test above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)

print "Level Order Traversal of binary tree is -"
printLevelOrder(root)
#This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

```

Level order traversal of binary tree is -
1 2 3 4 5

```

Time Complexity: $O(n)$ where n is number of nodes in the binary tree

References:

http://en.wikipedia.org/wiki/Breadth-first_traversal

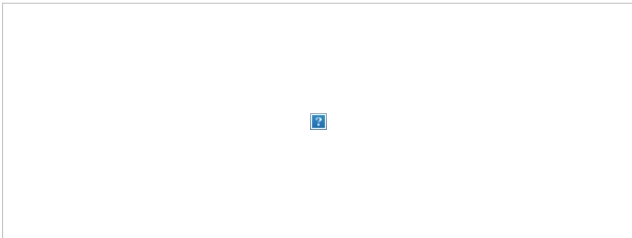
Please write comments if you find any bug in the above programs/algorithms or other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Queue
Trees

Diameter of a Binary Tree

The diameter of a tree (sometimes called the width) is the number of nodes on the longest path between two leaves in the tree. The diagram below shows two trees each with diameter nine, the leaves that form the ends of a longest path are shaded (note that there is more than one path in each tree of length nine, but no path longer than nine nodes).



The diameter of a tree T is the largest of the following quantities:

- * the diameter of T's left subtree
- * the diameter of T's right subtree
- * the longest path between leaves that goes through the root of T (this can be computed from the heights of the subtrees of T)

Implementation:

C

```
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct node
{
    int data;
    struct node* left, *right;
};

/* function to create a new node of tree and returns pointer */
struct node* newNode(int data);

/* returns max of two integers */
int max(int a, int b);

/* function to Compute height of a tree. */
int height(struct node* node);

/* Function to get diameter of a binary tree */
int diameter(struct node * tree)
{
    /* base case where tree is empty */
    if (tree == NULL)
        return 0;

    /* get the height of left and right sub-trees */
    int lheight = height(tree->left);
    int rheight = height(tree->right);

    /* get the diameter of left and right sub-trees */
    int ldiameter = diameter(tree->left);
    int rdiameter = diameter(tree->right);

    /* Return max of following three
    1) Diameter of left subtree
    2) Diameter of right subtree
    3) Height of left subtree + height of right subtree + 1 */
    return max(lheight + rheight + 1, max(ldiameter, rdiameter));
}

/* UTILITY FUNCTIONS TO TEST diameter() FUNCTION */

/* The function Compute the "height" of a tree. Height is the
number of nodes along the longest path from the root node
down to the farthest leaf node.*/
int height(struct node* node)
{
    /* base case tree is empty */
    if (node == NULL)
```

```

return 0;

/* If tree is not empty then height = 1 + max of left
height and right heights */
return 1 + max(height(node->left), height(node->right));
}

/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* returns maximum of two integers */
int max(int a, int b)
{
    return (a >= b)? a: b;
}

/* Driver program to test above functions*/
int main()
{
    /* Constructed binary tree is
        1
       / \
      2  3
     / \
    4  5
    */
    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    printf("Diameter of the given binary tree is %d\n", diameter(root));

    getchar();
    return 0;
}

```

Java

```

// Recursive optimized Java program to find the diameter of a
// Binary Tree

/* Class containing left and right child of current
node and key value*/
class Node
{
    int data;
    Node left, right;

    public Node(int item)
    {
        data = item;
        left = right = null;
    }
}

/* Class to print the Diameter */
class BinaryTree
{
    Node root;

    /* Method to calculate the diameter and return it to main */
    int diameter(Node root)
    {
        /* base case if tree is empty */
        if (root == null)
            return 0;

        /* get the height of left and right sub trees */
        int lheight = height(root.left);
        int rheight = height(root.right);

        /* get the diameter of left and right subtrees */
        int ldiameter = diameter(root.left);
        int rdiameter = diameter(root.right);

        /* Return max of following three
        1) Diameter of left subtree
        2) Diameter of right subtree
        3) Height of left subtree + height of right subtree + 1 */
        return Math.max(lheight + rheight + 1,
            Math.max(ldiameter, rdiameter));
    }

    /* A wrapper over diameter(Node root) */
    int diameter()
    {
        return diameter(root);
    }

    /*The function Compute the "height" of a tree. Height is the
    number f nodes along the longest path from the root node
    down to the farthest leaf node.*/
    static int height(Node node)
    {
        /* base case tree is empty */
        if (node == null)
            return 0;

        /* If tree is not empty then height = 1 + max of left

```

```

        height and right heights */
        return (1 + Math.max(height(node.left), height(node.right)));
    }

    public static void main(String args[])
    {
        /* creating a binary tree and entering the nodes */
        BinaryTree tree = new BinaryTree();
        tree.root = new Node(1);
        tree.root.left = new Node(2);
        tree.root.right = new Node(3);
        tree.root.left.left = new Node(4);
        tree.root.left.right = new Node(5);

        System.out.println("The diameter of given binary tree is : "
            + tree.diameter());
    }
}

```

Python

```

# Python program to find the diameter of binary tree

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

"""
The function Compute the "height" of a tree. Height is the
number f nodes along the longest path from the root node
down to the farthest leaf node.
"""
def height(node):

    # Base Case : Tree is empty
    if node is None:
        return 0 ;

    # If tree is not empty then height = 1 + max of left
    # height and right heights
    return 1 + max(height(node.left) , height(node.right))

# Function to get the diamtere of a binary tree
def diameter(root):

    # Base Case when tree is empty
    if root is None:
        return 0;

    # Get the height of left and right sub-trees
    lheight = height(root.left)
    rheight = height(root.right)

    # Get the diameter of left and igh sub-trees
    ldiameter = diameter(root.left)
    rdiameter = diameter(root.right)

    # Return max of the following tree:
    # 1) Diameter of left subtree
    # 2) Diameter of right subtree
    # 3) Height of left subtree + height of right subtree +1
    return max(lheight + rheight + 1, max(ldiameter, rdiameter))

# Driver program to test above functions

"""
Constructed binary tree is
      1
     / \
    2   3
   / \
  4   5
"""

root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
print "Diameter of given binary tree is %d" %(diameter(root))

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Time Complexity: $O(n^2)$

Output:

```
Diameter of the given binary tree is 4
```

Optimized implementation: The above implementation can be optimized by calculating the height in the same recursion rather than calling a height() separately. Thanks to Amar for suggesting this optimized version. This optimization reduces time complexity to $O(n)$.

C

```

/*The second parameter is to store the height of tree.
Initially, we need to pass a pointer to a location with value
as 0. So, function should be used as follows:

int height = 0;

```

```

struct node *root = SomeFunctionToMakeTree();
int diameter = diameterOpt(root, &height); */
int diameterOpt(struct node *root, int* height)
{
    /* lh --> Height of left subtree
       rh --> Height of right subtree */
    int lh = 0, rh = 0;

    /* ldiameter --> diameter of left subtree
       rdiameter --> Diameter of right subtree */
    int ldiameter = 0, rdiameter = 0;

    if(root == NULL)
    {
        *height = 0;
        return 0; /* diameter is also 0 */
    }

    /* Get the heights of left and right subtrees in lh and rh
       And store the returned values in ldiameter and ldiameter */
    ldiameter = diameterOpt(root->left, &lh);
    rdiameter = diameterOpt(root->right, &rh);

    /* Height of current node is max of heights of left and
       right subtrees plus 1 */
    *height = max(lh, rh) + 1;

    return max(lh + rh + 1, max(ldiameter, rdiameter));
}

```

Java

```

// Recursive Java program to find the diameter of a
// Binary Tree

/* Class containing left and right child of current
node and key value*/
class Node
{
    int data;
    Node left, right;

    public Node(int item)
    {
        data = item;
        left = right = null;
    }
}

// A utility class to pass height object
class Height
{
    int h;
}

/* Class to print the Diameter */
class BinaryTree
{
    Node root;

    /* define height =0 globally and call diameterOpt(root,height)
    from main */
    int diameterOpt(Node root, Height height)
    {
        /* lh --> Height of left subtree
           rh --> Height of right subtree */
        Height lh = new Height(), rh = new Height();

        if (root == null)
        {
            height.h = 0;
            return 0; /* diameter is also 0 */
        }

        /* ldiameter --> diameter of left subtree
           rdiameter --> Diameter of right subtree */
        /* Get the heights of left and right subtrees in lh and rh
        And store the returned values in ldiameter and ldiameter */
        lh.h++; rh.h++;
        int ldiameter = diameterOpt(root.left, lh);
        int rdiameter = diameterOpt(root.right, rh);

        /* Height of current node is max of heights of left and
        right subtrees plus 1 */
        height.h = Math.max(lh.h, rh.h) + 1;

        return Math.max(lh.h + rh.h + 1, Math.max(ldiameter, rdiameter));
    }

    /* A wrapper over diameter(Node root) */
    int diameter()
    {
        Height height = new Height();
        return diameterOpt(root, height);
    }

    /*The function Compute the "height" of a tree. Height is the
    number of nodes along the longest path from the root node
    down to the farthest leaf node.*/
    static int height(Node node)
    {
        /* base case tree is empty */
        if (node == null)
            return 0;

        /* If tree is not empty then height = 1 + max of left
        height and right heights */
        return (1 + Math.max(height(node.left), height(node.right)));
    }

    public static void main(String args[])
    {
        /* creating a binary tree and entering the nodes */

```

```

BinaryTree tree = new BinaryTree();
tree.root = new Node(1);
tree.root.left = new Node(2);
tree.root.right = new Node(3);
tree.root.left.left = new Node(4);
tree.root.left.right = new Node(5);

System.out.println("The diameter of given binary tree is : "
    + tree.diameter());
}
}

```

Time Complexity: O(n)

Output:

4

Diameter of an N-ary tree

References:

<http://www.cs.duke.edu/courses/spring00/cps100/assign/trees/diameter.html>

Please write comments if you find any of the above codes/algorithms incorrect, or find other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Trees

Inorder Tree Traversal without Recursion

Using [Stack](#) is the obvious way to traverse tree without recursion. Below is an algorithm for traversing binary tree using stack. See [this](#) for step wise step execution of the algorithm.

- 1) Create an empty stack S.
- 2) Initialize current node as root
- 3) Push the current node to S and set current = current->left until current is NULL
- 4) If current is NULL and stack is not empty then
 - a) Pop the top item from stack.
 - b) Print the popped item, set current = popped_item->right
 - c) Go to step 3.
- 5) If current is NULL and stack is empty then we are done.

Let us consider the below tree for example

```

      1
     / \
    2   3
   / \
  4   5

```

Step 1 Creates an empty stack: S = NULL

Step 2 sets current as address of root: current -> 1

Step 3 Pushes the current node and set current = current->left until current is NULL

```

current -> 1
push 1: Stack S -> 1
current -> 2
push 2: Stack S -> 2, 1
current -> 4
push 4: Stack S -> 4, 2, 1
current = NULL

```

Step 4 pops from S

- a) Pop 4: Stack S -> 2, 1
- b) print "4"
- c) current = NULL /*right of 4 */ and go to step 3

Since current is NULL step 3 doesn't do anything.

Step 4 pops again.

- a) Pop 2: Stack S -> 1
- b) print "2"
- c) current -> 5/*right of 2 */ and go to step 3

Step 3 pushes 5 to stack and makes current NULL

```

Stack S -> 5, 1
current = NULL

```

Step 4 pops from S

- a) Pop 5: Stack S -> 1
- b) print "5"
- c) current = NULL /*right of 5 */ and go to step 3

Since current is NULL step 3 doesn't do anything

Step 4 pops again.

- a) Pop 1: Stack S -> NULL
- b) print "1"
- c) current -> 3 /*right of 5 */

Step 3 pushes 3 to stack and makes current NULL

```

Stack S -> 3
current = NULL

```

Step 4 pops from S

- a) Pop 3: Stack S -> NULL
- b) print "3"
- c) current = NULL /*right of 3 */

Traversal is done now as stack S is empty and current is NULL.

Implementation:

C

```

#include<stdio.h>
#include<stdlib.h>
#define bool int

/* A binary tree tNode has data, pointer to left child
and a pointer to right child */

```

```

struct tNode
{
    int data;
    struct tNode* left;
    struct tNode* right;
};

/* Structure of a stack node. Linked List implementation is used for
stack. A stack node contains a pointer to tree node and a pointer to
next stack node */
struct sNode
{
    struct tNode *t;
    struct sNode *next;
};

/* Stack related functions */
void push(struct sNode** top_ref, struct tNode *t);
struct tNode *pop(struct sNode** top_ref);
bool isEmpty(struct sNode *top);

/* Iterative function for inorder tree traversal */
void inorder(struct tNode *root)
{
    /* set current to root of binary tree */
    struct tNode *current = root;
    struct sNode *s = NULL; /* Initialize stack s */
    bool done = 0;

    while (!done)
    {
        /* Reach the left most tNode of the current tNode */
        if(current != NULL)
        {
            /* place pointer to a tree node on the stack before traversing
            the node's left subtree */
            push(&s, current);
            current = current->left;
        }

        /* backtrack from the empty subtree and visit the tNode
        at the top of the stack; however, if the stack is empty,
        you are done */
        else
        {
            if (!isEmpty(s))
            {
                current = pop(&s);
                printf("%d ", current->data);

                /* we have visited the node and its left subtree.
                Now, it's right subtree's turn */
                current = current->right;
            }
            else
            {
                done = 1;
            }
        }
    } /* end of while */
}

/* UTILITY FUNCTIONS */
/* Function to push an item to sNode*/
void push(struct sNode** top_ref, struct tNode *t)
{
    /* allocate tNode */
    struct sNode* new_tNode =
        (struct sNode*) malloc(sizeof(struct sNode));

    if(new_tNode == NULL)
    {
        printf("Stack Overflow\n");
        getchar();
        exit(0);
    }

    /* put in the data */
    new_tNode->t = t;

    /* link the old list off the new tNode */
    new_tNode->next = (*top_ref);

    /* move the head to point to the new tNode */
    (*top_ref) = new_tNode;
}

/* The function returns true if stack is empty, otherwise false */
bool isEmpty(struct sNode *top)
{
    return (top == NULL)? 1 : 0;
}

/* Function to pop an item from stack*/
struct tNode *pop(struct sNode** top_ref)
{
    struct tNode *res;
    struct sNode *top;

    /* If sNode is empty then error */
    if(isEmpty(*top_ref))
    {
        printf("Stack Underflow\n");
        getchar();
        exit(0);
    }
    else
    {
        top = *top_ref;
        res = top->t;
        *top_ref = top->next;
        free(top);
        return res;
    }
}

/* Helper function that allocates a new tNode with the
given data and NULL left and right pointers. */

```



```

struct tNode* newtNode(int data)
{
    struct tNode* tNode = (struct tNode*)
        malloc(sizeof(struct tNode));

    tNode->data = data;
    tNode->left = NULL;
    tNode->right = NULL;

    return(tNode);
}

/* Driver program to test above functions*/
int main()
{

    /* Constructed binary tree is
      1
     / \
    2   3
   / \
  4   5
  */
    struct tNode *root = newtNode(1);
    root->left = newtNode(2);
    root->right = newtNode(3);
    root->left->left = newtNode(4);
    root->left->right = newtNode(5);

    inorder(root);

    getchar();
    return 0;
}

```

Java

```

// non-recursive java program for inorder traversal

/* importing the necessary class */
import java.util.Stack;

/* Class containing left and right child of current
node and key value*/
class Node {

    int data;
    Node left, right;

    public Node(int item) {
        data = item;
        left = right = null;
    }
}

/* Class to print the inorder traversal */
class BinaryTree {

    Node root;

    void inorder() {
        if (root == null) {
            return;
        }

        //keep the nodes in the path that are waiting to be visited
        Stack<Node> stack = new Stack<Node>();
        Node node = root;

        //first node to be visited will be the left one
        while (node != null) {
            stack.push(node);
            node = node.left;
        }

        // traverse the tree
        while (stack.size() > 0) {

            // visit the top node
            node = stack.pop();
            System.out.print(node.data + " ");
            if (node.right != null) {
                node = node.right;
            }

            // the next node to be visited is the leftmost
            while (node != null) {
                stack.push(node);
                node = node.left;
            }
        }
    }

    public static void main(String args[]) {

        /* creating a binary tree and entering
        the nodes */
        BinaryTree tree = new BinaryTree();
        tree.root = new Node(1);
        tree.root.left = new Node(2);
        tree.root.right = new Node(3);
        tree.root.left.left = new Node(4);
        tree.root.left.right = new Node(5);
        tree.inorder();
    }
}

```

Python

```

# Python program to do inorder traversal without recursion

```

```
# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

# Iterative function for inorder tree traversal
def inOrder(root):

    # Set current to root of binary tree
    current = root
    s = [] # initialize stack
    done = 0

    while(not done):

        # Reach the left most Node of the current Node
        if current is not None:

            # Place pointer to a tree node on the stack
            # before traversing the node's left subtree
            s.append(current)

            current = current.left

        # BackTrack from the empty subtree and visit the Node
        # at the top of the stack; however, if the stack is
        # empty you are done
        else:
            if(len(s) >0 ):
                current = s.pop()
                print current.data,

            # We have visited the node and its left
            # subtree. Now, it's right subtree's turn
            current = current.right

        else:
            done = 1

# Driver program to test above function

""" Constructed binary tree is
      1
     / \
    2   3
   / \
  4   5 """

root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)

inOrder(root)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Time Complexity: O(n)

Output:

```
4 2 5 1 3
```

References:

<http://web.cs.wpi.edu/~cs2005/common/iterative.inorder>

<http://neural.cs.nthu.edu.tw/jang/courses/cs2351/slide/animation/Iterative%20Inorder%20Traversal.pps>

See [this post](#) for another approach of Inorder Tree Traversal without recursion and without stack!

Please write comments if you find any bug in above code/algorithm, or want to share more information about stack based Inorder Tree Traversal.

GATE CS Corner **Company Wise Coding Practice**

Trees
Tree Traversal

Inorder Tree Traversal without recursion and without stack!

Using Morris Traversal, we can traverse the tree without using stack and recursion. The idea of Morris Traversal is based on [Threaded Binary Tree](#). In this traversal, we first create links to Inorder successor and print the data using these links, and finally revert the changes to restore original tree.

```
1. Initialize current as root
```

```

2. While current is not NULL
If current does not have left child
a) Print current's data
b) Go to the right, i.e., current = current->right
Else
a) Make current as right child of the rightmost
   node in current's left subtree
b) Go to this left child, i.e., current = current->left

```

Although the tree is modified through the traversal, it is reverted back to its original shape after the completion. Unlike [Stack based traversal](#), no extra space is required for this traversal.

C

```

#include<stdio.h>
#include<stdlib.h>

/* A binary tree tNode has data, pointer to left child
   and a pointer to right child */
struct tNode
{
    int data;
    struct tNode* left;
    struct tNode* right;
};

/* Function to traverse binary tree without recursion and
   without stack */
void MorrisTraversal(struct tNode *root)
{
    struct tNode *current,*pre;

    if(root == NULL)
        return;

    current = root;
    while(current != NULL)
    {
        if(current->left == NULL)
        {
            printf("%d ", current->data);
            current = current->right;
        }
        else
        {
            /* Find the inorder predecessor of current */
            pre = current->left;
            while(pre->right != NULL && pre->right != current)
                pre = pre->right;

            /* Make current as right child of its inorder predecessor */
            if(pre->right == NULL)
            {
                pre->right = current;
                current = current->left;
            }

            /* Revert the changes made in if part to restore the original
               tree i.e., fix the right child of predecessor */
            else
            {
                pre->right = NULL;
                printf("%d ",current->data);
                current = current->right;
            } /* End of if condition pre->right == NULL */
        } /* End of if condition current->left == NULL */
    } /* End of while */
}

/* UTILITY FUNCTIONS */
/* Helper function that allocates a new tNode with the
   given data and NULL left and right pointers. */
struct tNode* newNode(int data)
{
    struct tNode* tNode = (struct tNode*)
        malloc(sizeof(struct tNode));
    tNode->data = data;
    tNode->left = NULL;
    tNode->right = NULL;

    return(tNode);
}

/* Driver program to test above functions */
int main()
{
    /* Constructed binary tree is
      1
     / \
    2   3
   / \
  4   5
    */
    struct tNode *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    MorrisTraversal(root);

    getchar();
    return 0;
}

```

Java

```

// Java program to print inorder traversal without recursion and stack

/* A binary tree tNode has data, pointer to left child
   and a pointer to right child */

```

```

class tNode
{
    int data;
    tNode left, right;

    tNode(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    tNode root;

    /* Function to traverse binary tree without recursion and
    without stack */
    void MorrisTraversal(tNode root) {
        tNode current, pre;

        if (root == null)
            return;

        current = root;
        while (current != null)
        {
            if (current.left == null)
            {
                System.out.print(current.data + " ");
                current = current.right;
            }
            else
            {
                /* Find the inorder predecessor of current */
                pre = current.left;
                while (pre.right != null && pre.right != current)
                    pre = pre.right;

                /* Make current as right child of its inorder predecessor */
                if (pre.right == null)
                {
                    pre.right = current;
                    current = current.left;
                }

                /* Revert the changes made in if part to restore the
                original tree i.e., fix the right child of predecessor */
                else
                {
                    pre.right = null;
                    System.out.print(current.data + " ");
                    current = current.right;
                } /* End of if condition pre->right == NULL */
            } /* End of if condition current->left == NULL */
        } /* End of while */
    }

    public static void main(String args[])
    {
        /* Constructed binary tree is
            1
           / \
          2  3
         / \
        4  5
        */
        BinaryTree tree = new BinaryTree();
        tree.root = new tNode(1);
        tree.root.left = new tNode(2);
        tree.root.right = new tNode(3);
        tree.root.left.left = new tNode(4);
        tree.root.left.right = new tNode(5);

        tree.MorrisTraversal(tree.root);
    }
}

// This code has been contributed by Mayank Jaiswal(mayank_24)

```

Python

```

# Python program to do inorder traversal without recursion and
# without stack Morris inOrder Traversal

```

```

# A binary tree node
class Node:

```

```

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

```

```

# Iterative function for inorder tree traversal
def MorrisTraversal(root):

```

```

    # Set current to root of binary tree
    current = root

```

```

    while(current is not None):

```

```

        if current.left is None:
            print current.data ,
            current = current.right
        else:
            #Find the inorder predecessor of current
            pre = current.left
            while(pre.right is not None and pre.right != current):
                pre = pre.right

```

```
# Make current as right child of its inorder predecessor
if(pre.right is None):
    pre.right = current
    current = current.left
```

```
# Revert the changes made in if part to restore the
# original tree i.e., fix the right child of predecessor
else:
    pre.right = None
    print current.data ,
    current = current.right
```

```
# Driver program to test above function
```

```
=====
```

```
Constructed binary tree is
```

```
      1
     / \
    2   3
   / \
  4   5
```

```
=====
```

```
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
```

```
MorrisTraversal(root)
```

```
# This code is contributed by Naveen Aili
```

Output:

```
4 2 5 1 3
```

References:

www.liacs.nl/~deutz/DS/september28.pdf

<http://comsci.liu.edu/~murali/algo/Morris.htm>

www.scss.tcd.ie/disciplines/software_systems/.../HughGibbonsSlides.pdf

Please write comments if you find any bug in above code/algorithm, or want to share more information about stack Morris Inorder Tree Traversal.

GATE CS Corner Company Wise Coding Practice

Trees
Tree Traversal

Threaded Binary Tree

Inorder traversal of a **Binary tree** is either be done using recursion or **with the use of a auxiliary stack**. The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion. A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node (if it exists).

There are two types of threaded binary trees.

Single Threaded: Where a NULL right pointers is made to point to the inorder successor (if successor exists)

Double Threaded: Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.

The threads are also useful for fast accessing ancestors of a node.

Following diagram shows an example Single Threaded Binary Tree. The dotted lines represent threads.

threadedBT



C representation of a Threaded Node

Following is C representation of a single threaded node.

```
struct Node
{
    int data;
    Node *left, *right;
    bool rightThread;
}
```

Since right pointer is used for two purposes, the boolean variable rightThread is used to indicate whether right pointer points to right child or inorder successor. Similarly, we can add leftThread for a double threaded binary tree.

Inorder Taversal using Threads

Following is C code for inorder traversal in a threaded binary tree.

```
// Utility function to find leftmost node in a tree rooted with n
struct Node* leftMost(struct Node *n)
{
    if (n == NULL)
        return NULL;

    while (n->left != NULL)
        n = n->left;

    return n;
}

// C code to do inorder traversal in a threaded binary tree
void inOrder(struct Node *root)
{
    struct Node *cur = leftMost(root);
    while (cur != NULL)
    {
        printf("%d ", cur->data);

        // If this node is a thread node, then go to
        // inorder successor
        if (cur->rightThread)
```

```
cur = cur->right;
else // Else go to the leftmost child in right subtree
cur = leftmost(cur->right);
}
}
```

Following diagram demonstrates inorder order traversal using threads.

Threaded Traversal



We will soon be discussing insertion and deletion in threaded binary trees.

Sources:

http://en.wikipedia.org/wiki/Threaded_binary_tree
www.cs.berkeley.edu/~kamil/teaching/su02/080802.ppt

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: [Tree](#)

Write a Program to Find the Maximum Depth or Height of a Tree

Given a binary tree, find height of it. Height of empty tree is 0 and height of below tree is 3.

Example Tree



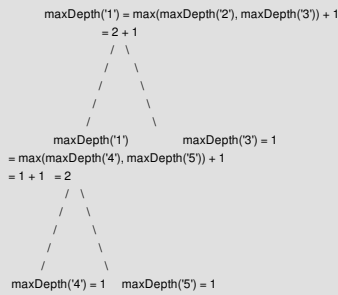
Example Tree

Recursively calculate height of left and right subtrees of a node and assign height to the node as max of the heights of two children plus 1. See below pseudo code and program for details.

Algorithm:

```
maxDepth()
1. If tree is empty then return 0
2. Else
  (a) Get the max depth of left subtree recursively i.e.,
    call maxDepth( tree->left-subtree)
  (a) Get the max depth of right subtree recursively i.e.,
    call maxDepth( tree->right-subtree)
  (c) Get the max of max depths of left and right
    subtrees and add 1 to it for the current node.
    max_depth = max(max dept of left subtree,
                    max depth of right subtree)
    + 1
  (d) Return max_depth
```

See the below diagram for more clarity about execution of the recursive function `maxDepth()` for above example tree.



Implementation:

C

```

#include<stdio.h>
#include<stdlib.h>

/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Compute the "maxDepth" of a tree -- the number of
nodes along the longest path from the root node
down to the farthest leaf node.*/
int maxDepth(struct node* node)
{
    if (node==NULL)
        return 0;
    else
    {
        /* compute the depth of each subtree */
        int lDepth = maxDepth(node->left);
        int rDepth = maxDepth(node->right);

        /* use the larger one */
        if (lDepth > rDepth)
            return(lDepth+1);
        else return(rDepth+1);
    }
}

/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

int main()
{
    struct node *root = newNode(1);

    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    printf("Hight of tree is %d", maxDepth(root));

    getchar();
    return 0;
}

```

Java

```

// Java program to find height of tree

/* A binary tree node
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    /* Compute the "maxDepth" of a tree -- the number of
nodes along the longest path from the root node
down to the farthest leaf node.*/
    int maxDepth(Node node)
    {
        if (node == null)

```

```

        return 0;
    else
    {
        /* compute the depth of each subtree */
        int lDepth = maxDepth(node.left);
        int rDepth = maxDepth(node.right);

        /* use the larger one */
        if (lDepth > rDepth)
            return (lDepth + 1);
        else
            return (rDepth + 1);
    }
}

/* Driver program to test above functions */
public static void main(String[] args)
{
    BinaryTree tree = new BinaryTree();

    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);

    System.out.println("Height of tree is : " +
        tree.maxDepth(tree.root));
}
}

// This code has been contributed by Mayank Jaiswal(mayank_24)

```

Python

```

# Python program to find the maximum depth of tree

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

# Compute the "maxDepth" of a tree -- the number of nodes
# along the longest path from the root node down to the
# farthest leaf node
def maxDepth(node):
    if node is None:
        return 0 ;

    else :

        # Compute the depth of each subtree
        lDepth = maxDepth(node.left)
        rDepth = maxDepth(node.right)

        # Use the larger one
        if (lDepth > rDepth):
            return lDepth+1
        else:
            return rDepth+1

# Driver program to test above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)

print "Height of tree is %d" %(maxDepth(root))

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Time Complexity: O(n) (Please see our post [Tree Traversal](#) for details)

References:

<http://cslibrary.stanford.edu/110/BinaryTrees.html>

GATE CS Corner Company Wise Coding Practice

Trees
Height of a Tree
Tree Traversal
Trees

If you are given two traversal sequences, can you construct the binary tree?

It depends on what traversals are given. If one of the traversal methods is Inorder then the tree can be constructed, otherwise not.

Mirror



Therefore, following combination can uniquely identify a tree.

Inorder and Preorder.

Inorder and Postorder.

Inorder and Level-order.

And following do not.

Postorder and Preorder.

Preorder and Level-order.

Postorder and Level-order.

For example, Preorder, Level-order and Postorder traversals are same for the trees given in above diagram.

Preorder Traversal = AB

Postorder Traversal = BA

Level-Order Traversal = AB

So, even if three of them (Pre, Post and Level) are given, the tree can not be constructed.

GATE CS Corner Company Wise Coding Practice

Trees
Binary Tree
Tree Traversal

Clone a Binary Tree with Random Pointers

Given a Binary Tree where every node has following structure.

```
struct node {
    int key;
    struct node *left,*right,*random;
}
```

The random pointer points to any random node of the binary tree and can even point to NULL, clone the given binary tree.

Method 1 (Use Hashing)

The idea is to store mapping from given tree nodes to clone tree node in hashtable. Following are detailed steps.

1) Recursively traverse the given Binary and copy key value, left pointer and right pointer to clone tree. While copying, store the mapping from given tree node to clone tree node in a hashtable. In the following pseudo code, 'cloneNode' is currently visited node of clone tree and 'treeNode' is currently visited node of given tree.

```
cloneNode->key = treeNode->key
cloneNode->left = treeNode->left
cloneNode->right = treeNode->right
map[treeNode] = cloneNode
```

2) Recursively traverse both trees and set random pointers using entries from hash table.

```
cloneNode->random = map[treeNode->random]
```

Following is C++ implementation of above idea. The following implementation uses `map` from C++ STL. Note that `map` doesn't implement hash table, it actually is based on self-balancing binary search tree.

```
// A hashmap based C++ program to clone a binary tree with random pointers
#include<iostream>
#include<map>
using namespace std;

/* A binary tree node has data, pointer to left child, a pointer to right
child and a pointer to random node*/
struct Node
{
    int key;
    struct Node* left, *right, *random;
};

/* Helper function that allocates a new Node with the
given data and NULL left, right and random pointers. */
Node* newNode(int key)
{
    Node* temp = new Node;
    temp->key = key;
    temp->random = temp->right = temp->left = NULL;
    return (temp);
}

/* Given a binary tree, print its Nodes in inorder*/
void printInorder(Node* node)
{
    if (node == NULL)
        return;

    /* First recur on left subtree */
    printInorder(node->left);

    /* then print data of Node and its random */
    cout << "[" << node->key << " ";
    if (node->random == NULL)
        cout << "NULL], ";
    else
        cout << node->random->key << "], ";

    /* now recur on right subtree */
    printInorder(node->right);
}

// This function creates clone by copying key and left and right pointers
// This function also stores mapping from given tree node to clone.
Node* copyLeftRightNode(Node* treeNode, map<Node *, Node *> *mymap)
{
    if (treeNode == NULL)
        return NULL;
    Node* cloneNode = newNode(treeNode->key);
    (*mymap)[treeNode] = cloneNode;
    cloneNode->left = copyLeftRightNode(treeNode->left, mymap);
    cloneNode->right = copyLeftRightNode(treeNode->right, mymap);
    return cloneNode;
}

// This function copies random node by using the hashmap built by
// copyLeftRightNode()
void copyRandom(Node* treeNode, Node* cloneNode, map<Node *, Node *> *mymap)
{
    if (cloneNode == NULL)
        return;
}
```

```

cloneNode->random = (*mymap)[treeNode->random];
copyRandom(treeNode->left, cloneNode->left, mymap);
copyRandom(treeNode->right, cloneNode->right, mymap);
}

// This function makes the clone of given tree. It mainly uses
// copyLeftRightNode() and copyRandom()
Node* cloneTree(Node* tree)
{
    if (tree == NULL)
        return NULL;
    map<Node*, Node*> *mymap = new map<Node*, Node*>;
    Node* newTree = copyLeftRightNode(tree, mymap);
    copyRandom(tree, newTree, mymap);
    return newTree;
}

/* Driver program to test above functions */
int main()
{
    //Test No 1
    Node *tree = newNode(1);
    tree->left = newNode(2);
    tree->right = newNode(3);
    tree->left->left = newNode(4);
    tree->left->right = newNode(5);
    tree->random = tree->left->right;
    tree->left->left->random = tree;
    tree->left->right->random = tree->right;

    // Test No 2
    // tree = NULL;

    // Test No 3
    // tree = newNode(1);

    // Test No 4
    /* tree = newNode(1);
    tree->left = newNode(2);
    tree->right = newNode(3);
    tree->random = tree->right;
    tree->left->random = tree;
    */

    cout << "Inorder traversal of original binary tree is:\n";
    printInorder(tree);

    Node *clone = cloneTree(tree);

    cout << "\n\nInorder traversal of cloned binary tree is:\n";
    printInorder(clone);

    return 0;
}

```

Output:

```

Inorder traversal of original binary tree is:
[4 1], [2 NULL], [5 3], [1 5], [3 NULL],

Inorder traversal of cloned binary tree is:
[4 1], [2 NULL], [5 3], [1 5], [3 NULL],

```

Method 2 (Temporarily Modify the Given Binary Tree)

1. Create new nodes in cloned tree and insert each new node in original tree between the left pointer edge of corresponding node in the original tree (See the below image).

i.e. if current node is A and it's left child is B (A —>> B), then new cloned node with key A will be created (say cA) and it will be put as A —>> cA —>> B (B can be a NULL or a non-NULL left child). Right child pointer will be set correctly i.e. if for current node A, right child is C in original tree (A —>> C) then corresponding cloned nodes cA and cC will like cA —>> cC



2. Set random pointer in cloned tree as per original tree

i.e. if node A's random pointer points to node B, then in cloned tree, cA will point to cB (cA and cB are new node in cloned tree corresponding to node A and B in original tree)

3. Restore left pointers correctly in both original and cloned tree

Following is C++ implementation of above algorithm.

```

#include <iostream>
using namespace std;

/* A binary tree node has data, pointer to left child, a pointer to right
child and a pointer to random node */
struct Node
{
    int key;
    struct Node* left, *right, *random;
};

/* Helper function that allocates a new Node with the
given data and NULL left, right and random pointers. */
Node* newNode(int key)
{
    Node* temp = new Node;
    temp->key = key;
    temp->random = temp->right = temp->left = NULL;
    return (temp);
}

/* Given a binary tree, print its Nodes in inorder */
void printInorder(Node* node)
{
    if (node == NULL)
        return;
}

```

```

/* First recur on left subtree */
printInorder(node->left);

/* then print data of Node and its random */
cout << " " << node->key << " ";
if (node->random == NULL)
    cout << "NULL", ";
else
    cout << node->random->key << " ", ";

/* now recur on right subtree */
printInorder(node->right);
}

// This function creates new nodes cloned tree and puts new cloned node
// in between current node and its left child
// i.e. if current node is A and its left child is B ( A --->> B ),
// then new cloned node with key A will be created (say cA) and
// it will be put as
// A --->> cA --->> B
// Here B can be a NULL or a non-NULL left child
// Right child pointer will be set correctly
// i.e. if for current node A, right child is C in original tree
// (A --->> C) then corresponding cloned nodes cA and cC will like
// cA --->> cC
Node* copyLeftRightNode(Node* treeNode)
{
    if (treeNode == NULL)
        return NULL;

    Node* left = treeNode->left;
    treeNode->left = newNode(treeNode->key);
    treeNode->left->left = left;
    if(left != NULL)
        left->left = copyLeftRightNode(left);

    treeNode->left->right = copyLeftRightNode(treeNode->right);
    return treeNode->left;
}

// This function sets random pointer in cloned tree as per original tree
// i.e. if node A's random pointer points to node B, then
// in cloned tree, cA will point to cB (cA and cB are new node in cloned
// tree corresponding to node A and B in original tree)
void copyRandomNode(Node* treeNode, Node* cloneNode)
{
    if (treeNode == NULL)
        return;
    if(treeNode->random != NULL)
        cloneNode->random = treeNode->random->left;
    else
        cloneNode->random = NULL;

    if(treeNode->left != NULL && cloneNode->left != NULL)
        copyRandomNode(treeNode->left->left, cloneNode->left->left);
    copyRandomNode(treeNode->right, cloneNode->right);
}

// This function will restore left pointers correctly in
// both original and cloned tree
void restoreTreeLeftNode(Node* treeNode, Node* cloneNode)
{
    if (treeNode == NULL)
        return;
    if (cloneNode->left != NULL)
    {
        Node* cloneLeft = cloneNode->left->left;
        treeNode->left = treeNode->left->left;
        cloneNode->left = cloneLeft;
    }
    else
        treeNode->left = NULL;

    restoreTreeLeftNode(treeNode->left, cloneNode->left);
    restoreTreeLeftNode(treeNode->right, cloneNode->right);
}

//This function makes the clone of given tree
Node* cloneTree(Node* treeNode)
{
    if (treeNode == NULL)
        return NULL;
    Node* cloneNode = copyLeftRightNode(treeNode);
    copyRandomNode(treeNode, cloneNode);
    restoreTreeLeftNode(treeNode, cloneNode);
    return cloneNode;
}

/* Driver program to test above functions*/
int main()
{
    /* Test No 1
    Node *tree = newNode(1);
    tree->left = newNode(2);
    tree->right = newNode(3);
    tree->left->left = newNode(4);
    tree->left->right = newNode(5);
    tree->random = tree->left->right;
    tree->left->left->random = tree;
    tree->left->right->random = tree->right;

    // Test No 2
    // Node *tree = NULL;

    /* Test No 3
    Node *tree = newNode(1);

    // Test No 4
    Node *tree = newNode(1);
    tree->left = newNode(2);
    tree->right = newNode(3);
    tree->random = tree->right;
    tree->left->random = tree;

    Test No 5

```

```

Node *tree = newNode(1);
tree->left = newNode(2);
tree->right = newNode(3);
tree->left->left = newNode(4);
tree->left->right = newNode(5);
tree->right->left = newNode(6);
tree->right->right = newNode(7);
tree->random = tree->left;
*/

// Test No 6
Node *tree = newNode(10);
Node *n2 = newNode(6);
Node *n3 = newNode(12);
Node *n4 = newNode(5);
Node *n5 = newNode(8);
Node *n6 = newNode(11);
Node *n7 = newNode(13);
Node *n8 = newNode(7);
Node *n9 = newNode(9);
tree->left = n2;
tree->right = n3;
tree->random = n2;
n2->left = n4;
n2->right = n5;
n2->random = n8;
n3->left = n6;
n3->right = n7;
n3->random = n5;
n4->random = n9;
n5->left = n8;
n5->right = n9;
n5->random = tree;
n6->random = n9;
n9->random = n8;

/* Test No 7
Node *tree = newNode(1);
tree->left = newNode(2);
tree->right = newNode(3);
tree->left->random = tree;
tree->right->random = tree->left;
*/

cout << "Inorder traversal of original binary tree is:\n";
printInorder(tree);

Node *clone = cloneTree(tree);

cout << "\n\nInorder traversal of cloned binary tree is:\n";
printInorder(clone);

return 0;
}

```

Output:

```

Inorder traversal of original binary tree is:
[5 9], [6 7], [7 NULL], [8 10], [9 7], [10 6], [11 9], [12 8], [13 NULL],

Inorder traversal of cloned binary tree is:
[5 9], [6 7], [7 NULL], [8 10], [9 7], [10 6], [11 9], [12 8], [13 NULL],

```

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Hash
Hashing

Construct Tree from given Inorder and Preorder traversals

Let us consider the below traversals:

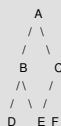
Inorder sequence: D B E A F C

Preorder sequence: A B D E C F

In a Preorder sequence, leftmost element is the root of the tree. So we know 'A' is root for given sequences. By searching 'A' in Inorder sequence, we can find out all elements on left side of 'A' are in left subtree and elements on right are in right subtree. So we know below structure now.



We recursively follow above steps and get the following tree.



Algorithm: buildTree()

- 1) Pick an element from Preorder. Increment a Preorder Index Variable (preIndex in below code) to pick next element in next recursive call.
- 2) Create a new tree node tNode with the data as picked element.
- 3) Find the picked element's index in Inorder. Let the index be inIndex.
- 4) Call buildTree for elements before inIndex and make the built tree as left subtree of tNode.
- 5) Call buildTree for elements after inIndex and make the built tree as right subtree of tNode.
- 6) return tNode.

Thanks to Rohini and Tushar for suggesting the code.

C

```

/* program to construct tree using inorder and preorder traversals */
#include<stdio.h>

```

```

#include<stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    char data;
    struct node* left;
    struct node* right;
};

/* Prototypes for utility functions */
int search(char arr[], int strt, int end, char value);
struct node* newNode(char data);

/* Recursive function to construct binary of size len from
   Inorder traversal in[] and Preorder traversal pre[]. Initial values
   of inStrt and inEnd should be 0 and len -1. The function doesn't
   do any error checking for cases where inorder and preorder
   do not form a tree */
struct node* buildTree(char in[], char pre[], int inStrt, int inEnd)
{
    static int preIndex = 0;

    if(inStrt > inEnd)
        return NULL;

    /* Pick current node from Preorder traversal using preIndex
       and increment preIndex */
    struct node *tNode = newNode(pre[preIndex++]);

    /* If this node has no children then return */
    if(inStrt == inEnd)
        return tNode;

    /* Else find the index of this node in Inorder traversal */
    int inIndex = search(in, inStrt, inEnd, tNode->data);

    /* Using index in Inorder traversal, construct left and
       right subtress */
    tNode->left = buildTree(in, pre, inStrt, inIndex-1);
    tNode->right = buildTree(in, pre, inIndex+1, inEnd);

    return tNode;
}

/* UTILITY FUNCTIONS */
/* Function to find index of value in arr[start...end]
   The function assumes that value is present in in[] */
int search(char arr[], int strt, int end, char value)
{
    int i;
    for(i = strt; i <= end; i++)
    {
        if(arr[i] == value)
            return i;
    }
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(char data)
{
    struct node* node = (struct node*) malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* This function is here just to test buildTree() */
void printInorder(struct node* node)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    printInorder(node->left);

    /* then print the data of node */
    printf("%c ", node->data);

    /* now recur on right child */
    printInorder(node->right);
}

/* Driver program to test above functions */
int main()
{
    char in[] = {'D', 'B', 'E', 'A', 'F', 'C'};
    char pre[] = {'A', 'B', 'D', 'E', 'C', 'F'};
    int len = sizeof(in)/sizeof(in[0]);
    struct node *root = buildTree(in, pre, 0, len - 1);

    /* Let us test the built tree by printing Inorder traversal */
    printf("Inorder traversal of the constructed tree is\n");
    printInorder(root);
    getchar();
}

```

Java

```

// Java program to construct a tree using inorder and preorder traversal

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
class Node
{
    char data;
    Node left, right;

    Node(char item)

```

```

{
    data = item;
    left = right = null;
}
}

class BinaryTree
{
    Node root;
    static int preIndex = 0;

    /* Recursive function to construct binary of size len from
    Inorder traversal in[] and Preorder traversal pre[].
    Initial values of inStrt and inEnd should be 0 and len -1.
    The function doesn't do any error checking for cases where
    inorder and preorder do not form a tree */
    Node buildTree(char in[], char pre[], int inStrt, int inEnd)
    {
        if (inStrt > inEnd)
            return null;

        /* Pick current node from Preorder traversal using preIndex
        and increment preIndex */
        Node tNode = new Node(pre[preIndex++]);

        /* If this node has no children then return */
        if (inStrt == inEnd)
            return tNode;

        /* Else find the index of this node in Inorder traversal */
        int inIndex = search(in, inStrt, inEnd, tNode.data);

        /* Using index in Inorder traversal, construct left and
        right subtress */
        tNode.left = buildTree(in, pre, inStrt, inIndex - 1);
        tNode.right = buildTree(in, pre, inIndex + 1, inEnd);

        return tNode;
    }

    /* UTILITY FUNCTIONS */

    /* Function to find index of value in arr[start...end]
    The function assumes that value is present in in[] */
    int search(char arr[], int strt, int end, char value)
    {
        int i;
        for (i = strt; i <= end; i++)
        {
            if (arr[i] == value)
                return i;
        }
        return i;
    }

    /* This function is here just to test buildTree() */
    void printInorder(Node node)
    {
        if (node == null)
            return;

        /* first recur on left child */
        printInorder(node.left);

        /* then print the data of node */
        System.out.print(node.data + " ");

        /* now recur on right child */
        printInorder(node.right);
    }

    // driver program to test above functions
    public static void main(String args[])
    {
        BinaryTree tree = new BinaryTree();
        char in[] = new char[]{'D', 'B', 'E', 'A', 'F', 'C'};
        char pre[] = new char[]{'A', 'B', 'D', 'E', 'C', 'F'};
        int len = in.length;
        Node root = tree.buildTree(in, pre, 0, len - 1);

        // building the tree by printing inorder traversal
        System.out.println("Inorder traversal of constructed tree is : ");
        tree.printInorder(root);
    }

    // This code has been contributed by Mayank Jaiswal

```

Python

```

# Python program to construct tree using inorder and
# preorder traversals

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    """Recursive function to construct binary of size len from
    Inorder traversal in[] and Preorder traversal pre[]. Initial values
    of inStrt and inEnd should be 0 and len -1. The function doesn't
    do any error checking for cases where inorder and preorder
    do not form a tree """
    def buildTree(inOrder, preOrder, inStrt, inEnd):

        if (inStrt > inEnd):
            return None

```

```

# Pich current node from Preorder traversal using
# preIndex and increment preIndex
tNode = Node(preOrder[buildTree.preIndex])
buildTree.preIndex += 1

# If this node has no children then return
if inSrt == inEnd :
    return tNode

# Else find the index of this node in Inorder traversal
inIndex = search(inOrder, inSrt, inEnd, tNode.data)

# Using index in Inorder Traversal, construct left
# and right subtrees
tNode.left = buildTree(inOrder, preOrder, inSrt, inIndex-1)
tNode.right = buildTree(inOrder, preOrder, inIndex+1, inEnd)

return tNode

# UTILITY FUNCTIONS
# Function to find index of vau in arr[start..end]
# The function assumes that value is rpresent in inOrder[]

def search(arr, start, end, value):
    for i in range(start, end+1):
        if arr[i] == value:
            return i

def printInorder(node):
    if node is None:
        return

    # first recur on left child
    printInorder(node.left)

    #then print the data of node
    print node.data,

    # now recur on right child
    printInorder(node.right)

# Driver program to test above function
inOrder = ['D','B','E','A','F','C']
preOrder = ['A','B','D','E','C','F']
# Static variable preIndex
buildTree.preIndex = 0
root = buildTree(inOrder, preOrder, 0, len(inOrder)-1)

# Let us test the build tree by priting Inorder traversal
print "Inorder traversal of the constructed tree is"
printInorder(root)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output :

```

Inorder traversal of constructed tree is :
D B E A F C

```

Time Complexity: $O(n^2)$. Worst case occurs when tree is left skewed. Example Preorder and Inorder traversals for worst case are {A, B, C, D} and {D, C, B, A}.

[Construct a Binary Tree from Postorder and Inorder](#)

Please write comments if you find any bug in above codes/algorithms, or find other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Trees
Inorder Traversal
Preorder Traversal
Tree Traversal

Maximum width of a binary tree

Given a binary tree, write a function to get the maximum width of the given tree. Width of a tree is maximum of widths of all levels.

Let us consider the below example tree.

```

      1
     /\
    2 3
   /\ /\
  4 5 8
   /\
  6 7

```

For the above tree,

width of level 1 is 1,

width of level 2 is 2,

width of level 3 is 3

width of level 4 is 2.

So the maximum width of the tree is 3.

We strongly recommend that you click here and practice it, before moving on to the solution.

Method 1 (Using Level Order Traversal)

This method mainly involves two functions. One is to count nodes at a given level (getWidth), and other is to get the maximum width of the tree(getMaxWidth). getMaxWidth() makes use of getWidth() to get the width of all levels starting from root.

```

/*Function to print level order traversal of tree*/
getMaxWidth(tree)
maxWidth = 0
for i = 1 to height(tree)
    width = getWidth(tree, i);
    if(width > maxWidth)
        maxWidth = width

```

```
return width
```

```
/*Function to get width of a given level */
getWidth(tree, level)
if tree is NULL then return 0;
if level is 1, then return 1;
else if level greater than 1, then
    return getWidth(tree->left, level-1) +
    getWidth(tree->right, level-1);
```

C

```
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/*Function prototypes*/
int getWidth(struct node* root, int level);
int height(struct node* node);
struct node* newNode(int data);

/* Function to get the maximum width of a binary tree*/
int getMaxWidth(struct node* root)
{
    int maxWidth = 0;
    int width;
    int h = height(root);
    int i;

    /* Get width of each level and compare
       the width with maximum width so far */
    for(i=1; i<=h; i++)
    {
        width = getWidth(root, i);
        if(width > maxWidth)
            maxWidth = width;
    }

    return maxWidth;
}

/* Get width of a given level */
int getWidth(struct node* root, int level)
{
    if(root == NULL)
        return 0;

    if(level == 1)
        return 1;

    else if (level > 1)
        return getWidth(root->left, level-1) +
            getWidth(root->right, level-1);
}

/* UTILITY FUNCTIONS */
/* Compute the "height" of a tree -- the number of
   nodes along the longest path from the root node
   down to the farthest leaf node.*/
int height(struct node* node)
{
    if (node==NULL)
        return 0;
    else
    {
        /* compute the height of each subtree */
        int lHeight = height(node->left);
        int rHeight = height(node->right);
        /* use the larger one */

        return ((lHeight > rHeight)? (lHeight+1): (rHeight+1));
    }
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return(node);
}

/* Driver program to test above functions*/
int main()
{
    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->right = newNode(8);
    root->right->right->left = newNode(6);
    root->right->right->right = newNode(7);

    /*
    Constructed bunary tree is:
        1
       /\
      2 3
     /\ \
    4 5 8
    */
```



```

    / \
   6  7
*/
printf("Maximum width is %d\n", getMaxWidth(root));
getchar();
return 0;
}

```

Java

```

// Java program to calculate width of binary tree

/* A binary tree node has data, pointer to left child
and a pointer to right child */
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    /* Function to get the maximum width of a binary tree */
    int getMaxWidth(Node node)
    {
        int maxWidth = 0;
        int width;
        int h = height(node);
        int i;

        /* Get width of each level and compare
        the width with maximum width so far */
        for (i = 1; i <= h; i++)
        {
            width = getWidth(node, i);
            if (width > maxWidth)
                maxWidth = width;
        }

        return maxWidth;
    }

    /* Get width of a given level */
    int getWidth(Node node, int level)
    {
        if (node == null)
            return 0;

        if (level == 1)
            return 1;
        else if (level > 1)
            return getWidth(node.left, level - 1)
                + getWidth(node.right, level - 1);
        return 0;
    }

    /* UTILITY FUNCTIONS */

    /* Compute the "height" of a tree -- the number of
    nodes along the longest path from the root node
    down to the farthest leaf node.*/
    int height(Node node)
    {
        if (node == null)
            return 0;
        else
        {
            /* compute the height of each subtree */
            int lHeight = height(node.left);
            int rHeight = height(node.right);

            /* use the larger one */
            return (lHeight > rHeight) ? (lHeight + 1) : (rHeight + 1);
        }
    }

    /* Driver program to test above functions */
    public static void main(String args[])
    {
        BinaryTree tree = new BinaryTree();

        /*
        Constructed binary tree is:
            1
           / \
          2  3
         / \  \
        4  5  8
         / \
        6  7
        */
        tree.root = new Node(1);
        tree.root.left = new Node(2);
        tree.root.right = new Node(3);
        tree.root.left.left = new Node(4);
        tree.root.left.right = new Node(5);
        tree.root.right.right = new Node(8);
        tree.root.right.right.left = new Node(6);
        tree.root.right.right.right = new Node(7);

        System.out.println("Maximum width is " + tree.getMaxWidth(tree.root));
    }
}

// This code has been contributed by Mayank Jaiswal

```

Python

```
# Python program to find the maximum width of binary tree using Level Order Traversal.

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

# Function to get the maximum width of a binary tree
def getMaxWidth(root):
    maxWidth = 0
    h = height(root)
    # Get width of each level and compare the width with maximum width so far
    for i in range(1, h+1):
        width = getWidth(root, i)
        if (width > maxWidth):
            maxWidth = width
    return maxWidth

# Get width of a given level
def getWidth(root, level):
    if root is None:
        return 0
    if level == 1:
        return 1
    elif level > 1:
        return (getWidth(root.left, level-1) + getWidth(root.right, level-1))

# UTILITY FUNCTIONS
# Compute the "height" of a tree -- the number of
# nodes along the longest path from the root node
# down to the farthest leaf node.
def height(node):
    if node is None:
        return 0
    else:
        # compute the height of each subtree
        lHeight = height(node.left)
        rHeight = height(node.right)
        # use the larger one
        return (lHeight+1) if (lHeight > rHeight) else (rHeight+1)

# Driver program to test above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.right = Node(8)
root.right.right.left = Node(6)
root.right.right.right = Node(7)

"""
Constructed binary tree is:
      1
     /\
    2 3
   /\ \
  4 5 8
   /\
  6 7
"""

print "Maximum width is %d" %(getMaxWidth(root))

# This code is contributed by Naveen Ali
```

Time Complexity: $O(n^2)$ in the worst case.

We can use Queue based level order traversal to optimize the time complexity of this method. The Queue based level order traversal will take $O(n)$ time in worst case. Thanks to [Nitish](#), [DivyaC](#) and [tech.login.id2](#) for suggesting this optimization. See their comments for implementation using queue based traversal.

Method 2 (Using Level Order Traversal with Queue)

In this method we store all the child nodes at the current level in the queue and then count the total number of nodes after the level order traversal for a particular level is completed. Since the queue now contains all the nodes of the next level, we can easily find out the total number of nodes in the next level by finding the size of queue. We then follow the same procedure for the successive levels. We store and update the maximum number of nodes found at each level.

C++

```
// A queue based C++ program to find maximum width
// of a Binary Tree
#include<bits/stdc++.h>
using namespace std ;

/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct Node
{
    int data ;
    struct Node * left ;
    struct Node * right ;
};

// Function to find the maximum width of the tree
// using level order traversal
int maxWidth(struct Node * root)
{
    // Base case
    if (root == NULL)
        return 0;

    // Initialize result
    int result = 0;

    // Do Level order traversal keeping track of number
```

```

// of nodes at every level.
queue<Node*> q;
q.push(root);
while (!q.empty())
{
    // Get the size of queue when the level order
    // traversal for one level finishes
    int count = q.size() ;

    // Update the maximum node count value
    result = max(count, result);

    // Iterate for all the nodes in the queue currently
    while (count-->0)
    {
        // Dequeue an node from queue
        Node *temp = q.front();
        q.pop();

        // Enqueue left and right children of
        // dequeued node
        if (temp->left != NULL)
            q.push(temp->left);
        if (temp->right != NULL)
            q.push(temp->right);
    }
}

return result;
}

/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
struct Node * newNode(int data)
{
    struct Node * node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

int main()
{
    struct Node *root = newNode(1);
    root->left  = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->right = newNode(8);
    root->right->right->left = newNode(6);
    root->right->right->right = newNode(7);

    /* Constructed Binary tree is:
      1
     / \
    2   3
   / \   \
  4  5   8
   / \   / \
  6  7  */*

    cout << "Maximum width is "
    << maxWidth(root) << endl;
    return 0;
}

// This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Python

Python program to find the maximum width of binary tree using Level Order Traversal with queue.

A binary tree node
class Node:

Constructor to create a new node
def __init__(self, data):
 self.data = data
 self.left = None
 self.right = None

Function to get the maximum width of a binary tree
def getMaxWidth(root):

base case
 if root is None:
 return 0

q = []
 maxWidth = 0

q.insert(0,root)

while (q != []):
 # Get the size of queue when the level order
 # traversal for one level finishes
 count = len(q)

Update the maximum node count value
 maxWidth = max(count,maxWidth)

while (count is not 0):
 count = count-1
 temp = q[0]
 q.pop() ;
 if temp.left is not None:
 q.insert(0,temp.left)

if temp.right is not None:
 q.insert(0,temp.right)

return maxWidth

Driver program to test above function
root = Node(1)
root.left = Node(2)

```

root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.right = Node(8)
root.right.right.left = Node(6)
root.right.right.right = Node(7)

===

Constructed binary tree is:
  1
 /\
2 3
 /\ \
4 5 8
 /\
6 7
===

print "Maximum width is %d" %(getMaxWidth(root))

# This code is contributed by Naveen Aili

```

Method 3 (Using Preorder Traversal)

In this method we create a temporary array count[] of size equal to the height of tree. We initialize all values in count as 0. We traverse the tree using preorder traversal and fill the entries in count so that the count array contains count of nodes at each level in Binary Tree.

C

```

#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

// A utility function to get height of a binary tree
int height(struct node* node);

// A utility function to allocate a new node with given data
struct node* newNode(int data);

// A utility function that returns maximum value in arr[] of size n
int getMax(int arr[], int n);

// A function that fills count array with count of nodes at every
// level of given binary tree
void getMaxWidthRecur(struct node *root, int count[], int level);

/* Function to get the maximum width of a binary tree */
int getMaxWidth(struct node* root)
{
    int width;
    int h = height(root);

    // Create an array that will store count of nodes at each level
    int *count = (int *)calloc(sizeof(int), h);

    int level = 0;

    // Fill the count array using preorder traversal
    getMaxWidthRecur(root, count, level);

    // Return the maximum value from count array
    return getMax(count, h);
}

// A function that fills count array with count of nodes at every
// level of given binary tree
void getMaxWidthRecur(struct node *root, int count[], int level)
{
    if(root)
    {
        count[level]++;
        getMaxWidthRecur(root->left, count, level+1);
        getMaxWidthRecur(root->right, count, level+1);
    }
}

/* UTILITY FUNCTIONS */
/* Compute the "height" of a tree -- the number of
nodes along the longest path from the root node
down to the farthest leaf node.*/
int height(struct node* node)
{
    if (node==NULL)
        return 0;
    else
    {
        /* compute the height of each subtree */
        int lHeight = height(node->left);
        int rHeight = height(node->right);
        /* use the larger one */

        return ((lHeight > rHeight)? (lHeight+1): (rHeight+1));
    }
}

/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
}

```

```

return(node);
}

// Return the maximum value from count array
int getMax(int arr[], int n)
{
    int max = arr[0];
    int i;
    for (i = 0; i < n; i++)
    {
        if (arr[i] > max)
            max = arr[i];
    }
    return max;
}

/* Driver program to test above functions */
int main()
{
    struct node *root = newNode(1);
    root->left    = newNode(2);
    root->right   = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->right = newNode(8);
    root->right->right->left = newNode(6);
    root->right->right->right = newNode(7);

    /*
    Constructed bunary tree is:
        1
       / \
      2  3
     / \  \
    4  5  8
     \ \
      6  7
    */
    printf("Maximum width is %d\n", getMaxWidth(root));
    getchar();
    return 0;
}

```

Java

```

// Java program to calculate width of binary tree

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    /* Function to get the maximum width of a binary tree */
    int getMaxWidth(Node node)
    {
        int width;
        int h = height(node);

        // Create an array that will store count of nodes at each level
        int count[] = new int[10];

        int level = 0;

        // Fill the count array using preorder traversal
        getMaxWidthRecur(node, count, level);

        // Return the maximum value from count array
        return getMax(count, h);
    }

    // A function that fills count array with count of nodes at every
    // level of given binary tree
    void getMaxWidthRecur(Node node, int count[], int level)
    {
        if (node != null)
        {
            count[level]++;
            getMaxWidthRecur(node.left, count, level + 1);
            getMaxWidthRecur(node.right, count, level + 1);
        }
    }

    /* UTILITY FUNCTIONS */

    /* Compute the "height" of a tree -- the number of
       nodes along the longest path from the root node
       down to the farthest leaf node.*/
    int height(Node node)
    {
        if (node == null)
            return 0;
        else
        {
            /* compute the height of each subtree */
            int lHeight = height(node.left);
            int rHeight = height(node.right);

            /* use the larger one */
            return (lHeight > rHeight) ? (lHeight + 1) : (rHeight + 1);
        }
    }
}

```

```

}

// Return the maximum value from count array
int getMax(int arr[], int n)
{
    int max = arr[0];
    int i;
    for (i = 0; i < n; i++)
    {
        if (arr[i] > max)
            max = arr[i];
    }
    return max;
}

/* Driver program to test above functions */
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();

    /*
    Constructed bunary tree is:
        1
       / \
      2  3
     /\  \
    4 5 8
     /\
    6 7 */
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);
    tree.root.right.right = new Node(8);
    tree.root.right.right.left = new Node(6);
    tree.root.right.right.right = new Node(7);

    System.out.println("Maximum width is " +
        tree.getMaxWidth(tree.root));
}
}

// This code has been contributed by Mayank Jaiswal

```

Python

Python program to find the maximum width of binary tree using Preorder Traversal.

A binary tree node
class Node:

```

# Constructor to create a new node
def __init__(self, data):
    self.data = data
    self.left = None
    self.right = None

```

```

# Function to get the maximum width of a binary tree
def getMaxWidth(root):
    h = height(root)
    # Create an array that will store count of nodes at each level
    count = [0] * h

```

```

    level = 0
    # Fill the count array using preorder traversal
    getMaxWidthRecur(root, count, level)

```

```

    # Return the maximum value from count array
    return getMax(count, h)

```

A function that fills count array with count of nodes at every
level of given binary tree

```

def getMaxWidthRecur(root, count, level):
    if root is not None:
        count[level] += 1
        getMaxWidthRecur(root.left, count, level+1)
        getMaxWidthRecur(root.right, count, level+1)

```

UTILITY FUNCTIONS

Compute the "height" of a tree -- the number of
nodes along the longest path from the root node
down to the farthest leaf node.

```

def height(node):
    if node is None:
        return 0
    else:
        # compute the height of each subtree
        lHeight = height(node.left)
        rHeight = height(node.right)
        # use the larger one
        return (lHeight+1) if (lHeight > rHeight) else (rHeight+1)

```

Return the maximum value from count array

```

def getMax(count, n):
    max = count[0]
    for i in range (1,n):
        if (count[i] > max):
            max = count[i]
    return max

```

Driver program to test above function

```

root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.right = Node(8)
root.right.right.left = Node(6)
root.right.right.right = Node(7)

```

====

Constructed bunary tree is:

```

1
/\
2 3
/\ \
4 5 8
/\
6 7
***

print "Maximum width is %d" %(getMaxWidth(root))

# This code is contributed by Naveen Aili

```

Thanks to [Raja](#) and [jagdish](#) for suggesting this method.
Time Complexity: O(n)

Please write comments if you find the above code/algorithm incorrect, or find better ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Trees

Print nodes at k distance from root

Given a root of a tree, and an integer k. Print all the nodes which are at k distance from root.

For example, in the below tree, 4, 5 & 8 are at distance 2 from root.

```

      1
     /\
    2  3
   /\ /
  4 5 8

```

The problem can be solved using recursion. Thanks to [eldho](#) for suggesting the solution.

C

```

#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

void printKDistant(struct node *root, int k)
{
    if(root == NULL)
        return;
    if( k == 0 )
    {
        printf( "%d ", root->data );
        return ;
    }
    else
    {
        printKDistant( root->left, k-1 );
        printKDistant( root->right, k-1 );
    }
}

/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Driver program to test above functions*/
int main()
{
    /* Constructed binary tree is
    1
   /\
  2  3
 /\ /
4 5 8
*/
    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(8);

    printKDistant(root, 2);

    getchar();
    return 0;
}

```

Java

```

// Java program to print nodes at k distance from root

```

```

/* A binary tree node has data, pointer to left child
and a pointer to right child */
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    void printKDistant(Node node, int k)
    {
        if (node == null)
            return;
        if (k == 0)
        {
            System.out.print(node.data + " ");
            return;
        }
        else
        {
            printKDistant(node.left, k - 1);
            printKDistant(node.right, k - 1);
        }
    }
}

/* Driver program to test above functions */
public static void main(String args[]) {
    BinaryTree tree = new BinaryTree();

    /* Constructed binary tree is
        1
       / \
      2  3
     / \ /
    4  5 8
    */
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);
    tree.root.right.left = new Node(8);

    tree.printKDistant(tree.root, 2);
}
}

// This code has been contributed by Mayank Jaiswal

```

Python

```

# Python program to find the nodes at k distance from root

# A Binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

def printKDistant(root, k):

    if root is None:
        return
    if k == 0:
        print root.data,
    else:
        printKDistant(root.left, k-1)
        printKDistant(root.right, k-1)

# Driver program to test above function
"""
Constructed binary tree is
    1
   / \
  2  3
 / \ /
4  5 8
"""
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.left = Node(8)

printKDistant(root, 2)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

The above program prints 4, 5 and 8.

Time Complexity: $O(n)$ where n is number of nodes in the given binary tree.

Please write comments if you find the above code/algorithm incorrect, or find better ways to solve the same problem.

Print Ancestors of a given node in Binary Tree

Given a Binary Tree and a key, write a function that prints all the ancestors of the key in the given binary tree.

For example, if the given tree is following Binary Tree and key is 7, then your function should print 4, 2 and 1.

```
      1
     / \
    2   3
   / \
  4   5
 /
7
```

Thanks to [Mike](#) , [Sambasiva](#) and [wgpshashank](#) for their contribution.

C++

```
#include<iostream>
#include<stdio.h>
#include<stdlib.h>

using namespace std;

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* If target is present in tree, then prints the ancestors
   and returns true, otherwise returns false. */
bool printAncestors(struct node *root, int target)
{
    /* base cases */
    if (root == NULL)
        return false;

    if (root->data == target)
        return true;

    /* If target is present in either left or right subtree of this node,
       then print this node */
    if ( printAncestors(root->left, target) ||
        printAncestors(root->right, target) )
    {
        cout << root->data << " ";
        return true;
    }

    /* Else return false */
    return false;
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newnode(int data)
{
    struct node* node = (struct node*)
        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Driver program to test above functions*/
int main()
{
    /* Construct the following binary tree
       1
      / \
     2   3
    / \
   4   5
  /
 7
    */
    struct node *root = newnode(1);
    root->left = newnode(2);
    root->right = newnode(3);
    root->left->left = newnode(4);
    root->left->right = newnode(5);
    root->left->left->left = newnode(7);

    printAncestors(root, 7);

    getchar();
    return 0;
}
```

Java

```
// Java program to print ancestors of given node

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
class Node
{
    int data;
    Node left, right, nextRight;
```

```

Node(int item)
{
    data = item;
    left = right = nextRight = null;
}
}

class BinaryTree
{
    Node root;

    /* If target is present in tree, then prints the ancestors
    and returns true, otherwise returns false. */
    boolean printAncestors(Node node, int target)
    {
        /* base cases */
        if (node == null)
            return false;

        if (node.data == target)
            return true;

        /* If target is present in either left or right subtree
        of this node, then print this node */
        if (printAncestors(node.left, target)
            || printAncestors(node.right, target))
        {
            System.out.print(node.data + " ");
            return true;
        }

        /* Else return false */
        return false;
    }

    /* Driver program to test above functions */
    public static void main(String args[])
    {
        BinaryTree tree = new BinaryTree();

        /* Construct the following binary tree
            1
           / \
          2  3
         / \
        4  5
       /
      7
        */
        tree.root = new Node(1);
        tree.root.left = new Node(2);
        tree.root.right = new Node(3);
        tree.root.left.left = new Node(4);
        tree.root.left.right = new Node(5);
        tree.root.left.left.left = new Node(7);

        tree.printAncestors(tree.root, 7);
    }
}

// This code has been contributed by Mayank Jaiswal

```

Python

```

# Python program to print ancestors of given node in
# binary tree

# A Binary Tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

# If target is present in tree, then prints the ancestors
# and returns true, otherwise returns false
def printAncestors(root, target):

    # Base case
    if root == None:
        return False

    if root.data == target:
        return True

    # If target is present in either left or right subtree
    # of this node, then print this node
    if (printAncestors(root.left, target) or
        printAncestors(root.right, target)):
        print root.data,
        return True

    # Else return False
    return False

# Driver program to test above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.left.left.left = Node(7)

printAncestors(root, 7)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

4 2 1

Time Complexity: $O(n)$ where n is the number of nodes in the given Binary Tree.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Trees

Check if a binary tree is subtree of another binary tree | Set 1

Given two binary trees, check if the first tree is subtree of the second one. A subtree of a tree T is a tree S consisting of a node in T and all of its descendants in T . The subtree corresponding to the root node is the entire tree; the subtree corresponding to any other node is called a proper subtree.

For example, in the following case, tree S is a subtree of tree T .

Tree 2

```
10
 / \
4   6
 \
 30
```

Tree 1

```
26
 / \
10  3
 / \ \
4   6 3
 \
 30
```

Solution: Traverse the tree T in preorder fashion. For every visited node in the traversal, see if the subtree rooted with this node is identical to S .

Following is the implementation for this.

C

```
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, left child and right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* A utility function to check whether trees with roots as root1 and
root2 are identical or not */
bool areIdentical(struct node * root1, struct node *root2)
{
    /* base cases */
    if (root1 == NULL && root2 == NULL)
        return true;

    if (root1 == NULL || root2 == NULL)
        return false;

    /* Check if the data of both roots is same and data of left and right
    subtrees are also same */
    return (root1->data == root2->data &&
            areIdentical(root1->left, root2->left) &&
            areIdentical(root1->right, root2->right) );
}

/* This function returns true if S is a subtree of T, otherwise false */
bool isSubtree(struct node *T, struct node *S)
{
    /* base cases */
    if (S == NULL)
        return true;

    if (T == NULL)
        return false;

    /* Check the tree with root as current node */
    if (areIdentical(T, S))
        return true;

    /* If the tree with root as current node doesn't match then
    try left and right subtrees one by one */
    return isSubtree(T->left, S) ||
           isSubtree(T->right, S);
}

/* Helper function that allocates a new node with the given data
and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node =
        (struct node*)malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return(node);
}

/* Driver program to test above function */
int main()
{
    // TREE 1
    /* Construct the following tree
```

```

    26
   / \
  10  3
 / \ \
4  6  3
 \
  30
*/
struct node *T = newNode(26);
T->right = newNode(3);
T->right->right = newNode(3);
T->left = newNode(10);
T->left->left = newNode(4);
T->left->left->right = newNode(30);
T->left->right = newNode(6);

// TREE 2
/* Construct the following tree
  10
 / \
4  6
 \
  30
*/
struct node *S = newNode(10);
S->right = newNode(6);
S->left = newNode(4);
S->left->right = newNode(30);

if (isSubtree(T, S))
    printf("Tree 2 is subtree of Tree 1");
else
    printf("Tree 2 is not a subtree of Tree 1");

getchar();
return 0;
}

```

Java

```

// Java program to check if binary tree is subtree of another binary tree

// A binary tree node
class Node
{
    int data;
    Node left, right, nextRight;

    Node(int item)
    {
        data = item;
        left = right = nextRight = null;
    }
}

class BinaryTree
{
    Node root1, root2;

    /* A utility function to check whether trees with roots as root1 and
    root2 are identical or not */
    boolean areIdentical(Node root1, Node root2)
    {
        /* base cases */
        if (root1 == null && root2 == null)
            return true;

        if (root1 == null || root2 == null)
            return false;

        /* Check if the data of both roots is same and data of left and right
        subtrees are also same */
        return (root1.data == root2.data
            && areIdentical(root1.left, root2.left)
            && areIdentical(root1.right, root2.right));
    }

    /* This function returns true if S is a subtree of T, otherwise false */
    boolean isSubtree(Node T, Node S)
    {
        /* base cases */
        if (S == null)
            return true;

        if (T == null)
            return false;

        /* Check the tree with root as current node */
        if (areIdentical(T, S))
            return true;

        /* If the tree with root as current node doesn't match then
        try left and right subtrees one by one */
        return isSubtree(T.left, S)
            || isSubtree(T.right, S);
    }

    public static void main(String args[])
    {
        BinaryTree tree = new BinaryTree();

        // TREE 1
        /* Construct the following tree
          26
         / \
        10  3
       / \ \
      4  6  3
       \
        30 */
    }
}

```

```

tree.root1 = new Node(26);
tree.root1.right = new Node(3);
tree.root1.right.right = new Node(3);
tree.root1.left = new Node(10);
tree.root1.left.left = new Node(4);
tree.root1.left.left.right = new Node(30);
tree.root1.left.right = new Node(6);

// TREE 2
/* Construct the following tree
  10
 /  \
4    6
 \
 30 */

tree.root2 = new Node(10);
tree.root2.right = new Node(6);
tree.root2.left = new Node(4);
tree.root2.left.right = new Node(30);

if (tree.isSubtree(tree.root1, tree.root2))
    System.out.println("Tree 2 is subtree of Tree 1 ");
else
    System.out.println("Tree 2 is not a subtree of Tree 1 ");
}
}

// This code has been contributed by Mayank Jaiswal

```

Python

```

# Python program to check binary tree is a subtree of
# another tree

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

# A utility function to check whether trees with roots
# as root 1 and root2 are identical or not
def areIdentical(root1, root2):

    # Base Case
    if root1 is None and root2 is None:
        return True
    if root1 is None or root2 is None:
        return False

    # Check if the data of both roots is same and data of
    # left and right subtrees are also same
    return (root1.data == root2.data and
            areIdentical(root1.left, root2.left) and
            areIdentical(root1.right, root2.right)
            )

# This function returns True if S is a subtree of T,
# otherwise False
def isSubtree(T, S):

    # Base Case
    if S is None:
        return True

    if T is None:
        return True

    # Check the tree with root as current node
    if (areIdentical(T, S)):
        return True

    # If the tree with root as current node doesn't match
    # then try left and right subtree one by one
    return isSubtree(T.left, S) or isSubtree(T.right, S)

# Driver program to test above function

""" TREE 1
Construct the following tree
  26
 /  \
10   3
/  \  \
4   6  3
 \
 30
"""

T = Node(26)
T.right = Node(3)
T.right.right = Node(3)
T.left = Node(10)
T.left.left = Node(4)
T.left.left.right = Node(30)
T.left.right = Node(6)

""" TREE 2
Construct the following tree
  10
 /  \
4    6
 \
 30
"""

S = Node(10)

```

```

S.right = Node(6)
S.left = Node(4)
S.left.right = Node(30)

if isSubtree(T, S):
    print "Tree 2 is subtree of Tree 1"
else :
    print "Tree 2 is not a subtree of Tree 1"

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

```
Tree 2 is subtree of Tree 1
```

Time Complexity: Time worst case complexity of above solution is $O(mn)$ where m and n are number of nodes in given two trees.

We can solve the above problem in $O(n)$ time. Please refer [Check if a binary tree is subtree of another binary tree | Set 2](#) for $O(n)$ solution.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Trees

Connect nodes at same level

Write a function to connect all the adjacent nodes at the same level in a binary tree. Structure of the given Binary Tree node is like following.

```

struct node{
    int data;
    struct node* left;
    struct node* right;
    struct node* nextRight;
}

```

Initially, all the nextRight pointers point to garbage values. Your function should set these pointers to point next right for each node.

Example

```

Input Tree
  A
 /\
B  C
 /\  \
D  E  F

Output Tree
A-->NULL
 /\
B->C->NULL
 /\  \
D->E->F->NULL

```

Method 1 (Extend Level Order Traversal or BFS)

Consider the method 2 of [Level Order Traversal](#). The method 2 can easily be extended to connect nodes of same level. We can augment queue entries to contain level of nodes also which is 0 for root, 1 for root's children and so on. So a queue node will now contain a pointer to a tree node and an integer level. When we enqueue a node, we make sure that correct level value for node is being set in queue. To set nextRight, for every node N , we dequeue the next node from queue, if the level number of next node is same, we set the nextRight of N as address of the dequeued node, otherwise we set nextRight of N as NULL.

Time Complexity: $O(n)$

Method 2 (Extend Pre Order Traversal)

This approach works only for [Complete Binary Trees](#). In this method we set nextRight in Pre Order fashion to make sure that the nextRight of parent is set before its children. When we are at node p , we set the nextRight of its left and right children. Since the tree is complete tree, nextRight of p 's left child ($p \rightarrow \text{left} \rightarrow \text{nextRight}$) will always be p 's right child, and nextRight of p 's right child ($p \rightarrow \text{right} \rightarrow \text{nextRight}$) will always be left child of p 's nextRight (if p is not the rightmost node at its level). If p is the rightmost node, then nextRight of p 's right child will be NULL.

C

```

#include <stdio.h>
#include <stdlib.h>

struct node
{
    int data;
    struct node *left;
    struct node *right;
    struct node *nextRight;
};

void connectRecur(struct node* p);

// Sets the nextRight of root and calls connectRecur() for other nodes
void connect (struct node *p)
{
    // Set the nextRight for root
    p->nextRight = NULL;

    // Set the next right for rest of the nodes (other than root)
    connectRecur(p);
}

/* Set next right of all descendents of p.
   Assumption: p is a complete binary tree */
void connectRecur(struct node* p)
{
    // Base case
    if (!p)
        return;

    // Set the nextRight pointer for p's left child
    if (p->left)
        p->left->nextRight = p->right;

    // Set the nextRight pointer for p's right child
    // p->nextRight will be NULL if p is the right most child at its level

```

```

if (p->right)
    p->right->nextRight = (p->nextRight)? p->nextRight->left: NULL;

// Set nextRight for other nodes in pre order fashion
connectRecur(p->left);
connectRecur(p->right);
}

/* UTILITY FUNCTIONS */
/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newnode(int data)
{
    struct node* node = (struct node*)
        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    node->nextRight = NULL;

    return(node);
}

/* Driver program to test above functions*/
int main()
{
    /* Constructed binary tree is
        10
       / \
      8   2
     /
    3
    */
    struct node *root = newnode(10);
    root->left = newnode(8);
    root->right = newnode(2);
    root->left->left = newnode(3);

    // Populates nextRight pointer in all nodes
    connect(root);

    // Let us check the values of nextRight pointers
    printf("Following are populated nextRight pointers in the tree "
        "(-1 is printed if there is no nextRight) \n");
    printf("nextRight of %d is %d \n", root->data,
        root->nextRight? root->nextRight->data: -1);
    printf("nextRight of %d is %d \n", root->left->data,
        root->left->nextRight? root->left->nextRight->data: -1);
    printf("nextRight of %d is %d \n", root->right->data,
        root->right->nextRight? root->right->nextRight->data: -1);
    printf("nextRight of %d is %d \n", root->left->left->data,
        root->left->left->nextRight? root->left->left->nextRight->data: -1);

    getchar();
    return 0;
}

```

Java

```

// Java program to connect nodes at same level using extended
// pre-order traversal

// A binary tree node
class Node
{
    int data;
    Node left, right, nextRight;

    Node(int item)
    {
        data = item;
        left = right = nextRight = null;
    }
}

class BinaryTree
{
    Node root;

    // Sets the nextRight of root and calls connectRecur() for other nodes
    void connect(Node p)
    {
        // Set the nextRight for root
        p.nextRight = null;

        // Set the next right for rest of the nodes (other than root)
        connectRecur(p);
    }

    /* Set next right of all descendents of p.
       Assumption: p is a complete binary tree */
    void connectRecur(Node p)
    {
        // Base case
        if (p == null)
            return;

        // Set the nextRight pointer for p's left child
        if (p.left != null)
            p.left.nextRight = p.right;

        // Set the nextRight pointer for p's right child
        // p->nextRight will be NULL if p is the right most child
        // at its level
        if (p.right != null)
            p.right.nextRight = (p.nextRight != null) ?
                p.nextRight.left : null;

        // Set nextRight for other nodes in pre order fashion
        connectRecur(p.left);
        connectRecur(p.right);
    }
}

```

```

}

// Driver program to test above functions
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();

    /* Constructed binary tree is
      10
     / \
    8   2
   /
  3
 */
    tree.root = new Node(10);
    tree.root.left = new Node(8);
    tree.root.right = new Node(2);
    tree.root.left.left = new Node(3);

    // Populates nextRight pointer in all nodes
    tree.connect(tree.root);

    // Let us check the values of nextRight pointers
    System.out.println("Following are populated nextRight pointers in "
        + "the tree" + " (-1 is printed if there is no nextRight)");
    int a = tree.root.nextRight != null ? tree.root.nextRight.data : -1;
    System.out.println("nextRight of " + tree.root.data + " is "
        + a);
    int b = tree.root.left.nextRight != null ?
        tree.root.left.nextRight.data : -1;
    System.out.println("nextRight of " + tree.root.left.data + " is "
        + b);
    int c = tree.root.right.nextRight != null ?
        tree.root.right.nextRight.data : -1;
    System.out.println("nextRight of " + tree.root.right.data + " is "
        + c);
    int d = tree.root.left.left.nextRight != null ?
        tree.root.left.left.nextRight.data : -1;
    System.out.println("nextRight of " + tree.root.left.left.data + " is "
        + d);

}
}

// This code has been contributed by Mayank Jaiswal

```

Thanks to Dhanya for suggesting this approach.

Time Complexity: O(n)

Why doesn't method 2 work for trees which are not Complete Binary Trees?

Let us consider following tree as an example. In Method 2, we set the nextRight pointer in pre order fashion. When we are at node 4, we set the nextRight of its children which are 8 and 9 (the nextRight of 4 is already set as node 5). nextRight of 8 will simply be set as 9, but nextRight of 9 will be set as NULL which is incorrect. We can't set the correct nextRight, because when we set nextRight of 9, we only have nextRight of node 4 and ancestors of node 4, we don't have nextRight of nodes in right subtree of root.

```

      1
     / \
    2   3
   /\  /\
  4 5 6 7
 /\  /\
8 9 10 11

```

See [Connect nodes at same level using constant extra space](#) for more solutions.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Trees

Binary Search Tree | Set 1 (Search and Insertion)

The following is definition of Binary Search Tree(BST) according to [Wikipedia](#)

Binary Search Tree, is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys less than the node's key.
 - The right subtree of a node contains only nodes with keys greater than the node's key.
 - The left and right subtree each must also be a binary search tree.
- There must be no duplicate nodes.

200px-Binary_search_tree.svg



The above properties of Binary Search Tree provide an ordering among keys so that the operations like search, minimum and maximum can be done fast. If there is no ordering, then we may have to compare every key to search a given key.

Searching a key

To search a given key in Binary Search Tree, we first compare it with root, if the key is present at root, we return root. If key is greater than root's key, we recur for right subtree of root node. Otherwise we recur for left subtree.

C/C++

```

// C function to search a given key in a given BST
struct node* search(struct node* root, int key)
{
    // Base Cases: root is null or key is present at root

```



```

if (root == NULL || root->key == key)
    return root;

// Key is greater than root's key
if (root->key < key)
    return search(root->right, key);

// Key is smaller than root's key
return search(root->left, key);
}

```

Python

```

# A utility function to search a given key in BST
def search(root,key):

    # Base Cases: root is null or key is present at root
    if root is None or root.val == key:
        return root

    # Key is greater than root's key
    if root.val < key:
        return search(root.right,key)

    # Key is smaller than root's key
    return search(root.left,key)

# This code is contributed by Bhavya Jain

```

Java

```

// A utility function to search a given key in BST
public Node search(Node root, int key)
{
    // Base Cases: root is null or key is present at root
    if (root==null || root.key==key)
        return root;

    // val is greater than root's key
    if (root.key > key)
        return search(root.left, key);

    // val is less than root's key
    return search(root.right, key);
}

```

Illustration:

bstsearch



Image is taken from [here](#).

Insertion of a key

A new key is always inserted at leaf. We start searching a key from root till we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node.

```

      100          100
     /  \      /  \
    /    \    /    \
   20    500 -----> 20    500
  /  \      /  \
 10  30    10  30
             \
             40

```

C/C++

```

// C program to demonstrate insert operation in binary search tree
#include<stdio.h>
#include<stdlib.h>

struct node
{
    int key;
    struct node *left, *right;
};

// A utility function to create a new BST node
struct node *newNode(int item)
{
    struct node *temp = (struct node *)malloc(sizeof(struct node));
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to do inorder traversal of BST
void inorder(struct node *root)

```

```

{
    if (root != NULL)
    {
        inorder(root->left);
        printf("%d\n", root->key);
        inorder(root->right);
    }
}

/* A utility function to insert a new node with given key in BST */
struct node* insert(struct node* node, int key)
{
    /* If the tree is empty, return a new node */
    if (node == NULL) return newNode(key);

    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);

    /* return the (unchanged) node pointer */
    return node;
}

// Driver Program to test above functions
int main()
{
    /* Let us create following BST
        50
       /  \
      30   70
     / \  / \
    20 40 60 80 */
    struct node *root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    // print inorder traversal of the BST
    inorder(root);

    return 0;
}

```

Python

Python program to demonstrate insert operation in binary search tree

A utility class that represents an individual node in a BST

```

class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

```

A utility function to insert a new node with the given key

```

def insert(root, node):
    if root is None:
        root = node
    else:
        if root.val < node.val:
            if root.right is None:
                root.right = node
            else:
                insert(root.right, node)
        else:
            if root.left is None:
                root.left = node
            else:
                insert(root.left, node)

```

A utility function to do inorder tree traversal

```

def inorder(root):
    if root:
        inorder(root.left)
        print(root.val)
        inorder(root.right)

```

Driver program to test the above functions

Let us create the following BST

```

# 50
# /  \
# 30   70
# /\  /\
# 20 40 60 80
r = Node(50)
insert(r, Node(30))
insert(r, Node(20))
insert(r, Node(40))
insert(r, Node(70))
insert(r, Node(60))
insert(r, Node(80))

```

```

# Print inorder traversal of the BST
inorder(r)

```

This code is contributed by Bhavya Jain

Java

```

// Java program to demonstrate insert operation in binary search tree
class BinarySearchTree {

```

```

    /* Class containing left and right child of current node and key value */
    class Node {

```

```

int key;
Node left, right;

public Node(int item) {
    key = item;
    left = right = null;
}
}

// Root of BST
Node root;

// Constructor
BinarySearchTree() {
    root = null;
}

// This method mainly calls insertRec()
void insert(int key) {
    root = insertRec(root, key);
}

/* A recursive function to insert a new key in BST */
Node insertRec(Node root, int key) {

    /* If the tree is empty, return a new node */
    if (root == null) {
        root = new Node(key);
        return root;
    }

    /* Otherwise, recur down the tree */
    if (key < root.key)
        root.left = insertRec(root.left, key);
    else if (key > root.key)
        root.right = insertRec(root.right, key);

    /* return the (unchanged) node pointer */
    return root;
}

// This method mainly calls inorderRec()
void inorder() {
    inorderRec(root);
}

// A utility function to do inorder traversal of BST
void inorderRec(Node root) {
    if (root != null) {
        inorderRec(root.left);
        System.out.println(root.key);
        inorderRec(root.right);
    }
}

// Driver Program to test above functions
public static void main(String[] args) {
    BinarySearchTree tree = new BinarySearchTree();

    /* Let us create following BST
        50
       /  \
      30   70
     / \  / \
    20 40 60 80 */
    tree.insert(50);
    tree.insert(30);
    tree.insert(20);
    tree.insert(40);
    tree.insert(70);
    tree.insert(60);
    tree.insert(80);

    // print inorder traversal of the BST
    tree.inorder();
}
// This code is contributed by Ankur Narain Verma

```

Output:

```

20
30
40
50
60
70
80

```

bstinsert



Image is taken from [here](#).

Time Complexity: The worst case time complexity of search and insert operations is $O(h)$ where h is height of Binary Search Tree. In worst case, we may have to travel from root to the deepest leaf node. The height of a

skewed tree may become n and the time complexity of search and insert operation may become $O(n)$.

Some Interesting Facts:

- Inorder traversal of BST always produces sorted output.
- We can construct a BST with only Preorder or Postorder or Level Order traversal. Note that we can always get inorder traversal by sorting the only given traversal.
- Number of unique BSTs with n distinct keys is Catalan Number

Related Links:

- [Binary Search Tree Delete Operation](#)
- [Quiz on Binary Search Tree](#)
- [Coding practice on BST](#)
- [All Articles on BST](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: [Tree](#)

Binary Search Tree | Set 2 (Delete)

We have discussed [BST search and insert operations](#). In this post, delete operation is discussed. When we delete a node, there possibilities arise.

1) **Node to be deleted is leaf:** Simply remove from the tree.

```
      50          50
     /  \      /  \
    30   70  delete(20) /  \
                   /  \
                  20   40
                 /  \
                40  60
               /  \
              60  80
```

2) **Node to be deleted has only one child:** Copy the child to the node and delete the child

```
      50          50
     /  \      /  \
    30   70  delete(30) /  \
                   /  \
                  40   70
                 /  \
                40  60
               /  \
              60  80
```

3) **Node to be deleted has two children:** Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.

```
      50          60
     /  \      /  \
    40   70  delete(50) /  \
                   /  \
                  40   70
                 /  \
                60   80
```

The important thing to note is, inorder successor is needed only when right child is not empty. In this particular case, inorder successor can be obtained by finding the minimum value in right child of the node.

C/C++

```
// C program to demonstrate delete operation in binary search tree
#include<stdio.h>
#include<stdlib.h>

struct node
{
    int key;
    struct node *left, *right;
};

// A utility function to create a new BST node
struct node *newNode(int item)
{
    struct node *temp = (struct node *)malloc(sizeof(struct node));
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to do inorder traversal of BST
void inorder(struct node *root)
{
    if (root != NULL)
    {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}

/* A utility function to insert a new node with given key in BST */
struct node *insert(struct node *node, int key)
{
    /* If the tree is empty, return a new node */
    if (node == NULL) return newNode(key);

    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left = insert(node->left, key);
    else
        node->right = insert(node->right, key);

    /* return the (unchanged) node pointer */
    return node;
}

/* Given a non-empty binary search tree, return the node with minimum
key value found in that tree. Note that the entire tree does not
need to be searched. */
```

```

struct node * minValueNode(struct node* node)
{
    struct node* current = node;

    /* loop down to find the leftmost leaf */
    while (current->left != NULL)
        current = current->left;

    return current;
}

/* Given a binary search tree and a key, this function deletes the key
and returns the new root */
struct node* deleteNode(struct node* root, int key)
{
    // base case
    if (root == NULL) return root;

    // If the key to be deleted is smaller than the root's key,
    // then it lies in left subtree
    if (key < root->key)
        root->left = deleteNode(root->left, key);

    // If the key to be deleted is greater than the root's key,
    // then it lies in right subtree
    else if (key > root->key)
        root->right = deleteNode(root->right, key);

    // if key is same as root's key, then This is the node
    // to be deleted
    else
    {
        // node with only one child or no child
        if (root->left == NULL)
        {
            struct node *temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL)
        {
            struct node *temp = root->left;
            free(root);
            return temp;
        }

        // node with two children: Get the inorder successor (smallest
        // in the right subtree)
        struct node* temp = minValueNode(root->right);

        // Copy the inorder successor's content to this node
        root->key = temp->key;

        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}

// Driver Program to test above functions
int main()
{
    /* Let us create following BST
        50
       /  \
      30   70
     / \  / \
    20 40 60 80 */
    struct node *root = NULL;
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 70);
    root = insert(root, 60);
    root = insert(root, 80);

    printf("Inorder traversal of the given tree \n");
    inorder(root);

    printf("\nDelete 20\n");
    root = deleteNode(root, 20);
    printf("Inorder traversal of the modified tree \n");
    inorder(root);

    printf("\nDelete 30\n");
    root = deleteNode(root, 30);
    printf("Inorder traversal of the modified tree \n");
    inorder(root);

    printf("\nDelete 50\n");
    root = deleteNode(root, 50);
    printf("Inorder traversal of the modified tree \n");
    inorder(root);

    return 0;
}

```

Java

```

// Java program to demonstrate delete operation in binary search tree
class BinarySearchTree
{
    /* Class containing left and right child of current node and key value */
    class Node
    {
        int key;
        Node left, right;

        public Node(int item)
        {
            key = item;
            left = right = null;
        }
    }
}

```

```

    }
}

// Root of BST
Node root;

// Constructor
BinarySearchTree()
{
    root = null;
}

// This method mainly calls deleteRec()
void deleteKey(int key)
{
    root = deleteRec(root, key);
}

/* A recursive function to insert a new key in BST */
Node deleteRec(Node root, int key)
{
    /* Base Case: If the tree is empty */
    if (root == null) return root;

    /* Otherwise, recur down the tree */
    if (key < root.key)
        root.left = deleteRec(root.left, key);
    else if (key > root.key)
        root.right = deleteRec(root.right, key);

    // If key is same as root's key, then This is the node
    // to be deleted
    else
    {
        // node with only one child or no child
        if (root.left == null)
            return root.right;
        else if (root.right == null)
            return root.left;

        // node with two children: Get the inorder successor (smallest
        // in the right subtree)
        root.key = minValue(root.right);

        // Delete the inorder successor
        root.right = deleteRec(root.right, root.key);
    }

    return root;
}

int minValue(Node root)
{
    int minv = root.key;
    while (root.left != null)
    {
        minv = root.left.key;
        root = root.left;
    }
    return minv;
}

// This method mainly calls insertRec()
void insert(int key)
{
    root = insertRec(root, key);
}

/* A recursive function to insert a new key in BST */
Node insertRec(Node root, int key)
{
    /* If the tree is empty, return a new node */
    if (root == null)
    {
        root = new Node(key);
        return root;
    }

    /* Otherwise, recur down the tree */
    if (key < root.key)
        root.left = insertRec(root.left, key);
    else if (key > root.key)
        root.right = insertRec(root.right, key);

    /* return the (unchanged) node pointer */
    return root;
}

// This method mainly calls inorderRec()
void inorder()
{
    inorderRec(root);
}

// A utility function to do inorder traversal of BST
void inorderRec(Node root)
{
    if (root != null)
    {
        inorderRec(root.left);
        System.out.print(root.key + " ");
        inorderRec(root.right);
    }
}

// Driver Program to test above functions
public static void main(String[] args)
{
    BinarySearchTree tree = new BinarySearchTree();

    /* Let us create following BST
    50
   / \
  30  70
 / \ / \
    */

```

```

20 40 60 80 */
tree.insert(50);
tree.insert(30);
tree.insert(20);
tree.insert(40);
tree.insert(70);
tree.insert(60);
tree.insert(80);

System.out.println("Inorder traversal of the given tree");
tree.inorder();

System.out.println("\nDelete 20");
tree.deleteKey(20);
System.out.println("Inorder traversal of the modified tree");
tree.inorder();

System.out.println("\nDelete 30");
tree.deleteKey(30);
System.out.println("Inorder traversal of the modified tree");
tree.inorder();

System.out.println("\nDelete 50");
tree.deleteKey(50);
System.out.println("Inorder traversal of the modified tree");
tree.inorder();
}
}

```

Python

```

# Python program to demonstrate delete operation
# in binary search tree

# A Binary Tree Node
class Node:

    # Constructor to create a new node
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

# A utility function to do inorder traversal of BST
def inorder(root):
    if root is not None:
        inorder(root.left)
        print root.key,
        inorder(root.right)

# A utility function to insert a new node with given key in BST
def insert( node, key):

    # If the tree is empty, return a new node
    if node is None:
        return Node(key)

    # Otherwise recur down the tree
    if key < node.key:
        node.left = insert(node.left, key)
    else:
        node.right = insert(node.right, key)

    # return the (unchanged) node pointer
    return node

# Given a non-empty binary search tree, return the node
# with minimum key value found in that tree. Note that the
# entire tree does not need to be searched
def minValueNode( node):
    current = node

    # loop down to find the leftmost leaf
    while(current.left is not None):
        current = current.left

    return current

# Given a binary search tree and a key, this function
# delete the key and returns the new root
def deleteNode(root, key):

    # Base Case
    if root is None:
        return root

    # If the key to be deleted is smaller than the root's
    # key then it lies in left subtree
    if key < root.key:
        root.left = deleteNode(root.left, key)

    # If the key to be deleted is greater than the root's key
    # then it lies in right subtree
    elif(key > root.key):
        root.right = deleteNode(root.right, key)

    # If key is same as root's key, then this is the node
    # to be deleted
    else:

        # Node with only one child or no child
        if root.left is None :
            temp = root.right
            root = None
            return temp

        elif root.right is None :
            temp = root.left
            root = None
            return temp

```

```

# Node with two children: Get the inorder successor
# (smallest in the right subtree)
temp = minValueNode(root.right)

# Copy the inorder successor's content to this node
root.key = temp.key

# Delete the inorder successor
root.right = deleteNode(root.right, temp.key)

return root

# Driver program to test above functions
"""Let us create following BST
          50
        /  \
       30   70
      / \   / \
     20 40 60 80 """

root = None
root = insert(root, 50)
root = insert(root, 30)
root = insert(root, 20)
root = insert(root, 40)
root = insert(root, 70)
root = insert(root, 60)
root = insert(root, 80)

print "Inorder traversal of the given tree"
inorder(root)

print "\nDelete 20"
root = deleteNode(root, 20)
print "Inorder traversal of the modified tree"
inorder(root)

print "\nDelete 30"
root = deleteNode(root, 30)
print "Inorder traversal of the modified tree"
inorder(root)

print "\nDelete 50"
root = deleteNode(root, 50)
print "Inorder traversal of the modified tree"
inorder(root)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

```

Inorder traversal of the given tree
20 30 40 50 60 70 80
Delete 20
Inorder traversal of the modified tree
30 40 50 60 70 80
Delete 30
Inorder traversal of the modified tree
40 50 60 70 80
Delete 50
Inorder traversal of the modified tree
40 60 70 80

```

Illustration:

bst-delete





Images source : <https://www.cs.purdue.edu/homes/ayg/CS251/slides/chap7b.pdf>

Time Complexity: The worst case time complexity of delete operation is $O(h)$ where h is height of Binary Search Tree. In worst case, we may have to travel from root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of delete operation may become $O(n)$

Related Links:

- [Binary Search Tree Introduction, Search and Insert/a>](#)
- [Quiz on Binary Search Tree](#)
- [Coding practice on BST](#)
- [All Articles on BST](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: Tree

Find the node with minimum value in a Binary Search Tree

This is quite simple. Just traverse the node from root to left recursively until left is NULL. The node whose left is NULL is the node with minimum value.

BST_LCA



For the above tree, we start with 20, then we move left 8, we keep on moving to left until we see NULL. Since left of 4 is NULL, 4 is the node with minimum value.

C

```
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Helper function that allocates a new node
with the given data and NULL left and right
pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Give a binary search tree and a number,
inserts a new node with the given number in
the correct place in the tree. Returns the new
root pointer which the caller should then use
(the standard trick to avoid using reference
parameters). */
struct node* insert(struct node* node, int data)
{
    /* 1. If the tree is empty, return a new,
       single node */
    if (node == NULL)
        return(newNode(data));
    else
    {
        /* 2. Otherwise, recur down the tree */
        if (data <= node->data)
```

```

    node->left = insert(node->left, data);
else
    node->right = insert(node->right, data);

/* return the (unchanged) node pointer */
return node;
}
}

/* Given a non-empty binary search tree,
return the minimum data value found in that
tree. Note that the entire tree does not need
to be searched. */
int minValue(struct node* node) {
    struct node* current = node;

    /* loop down to find the leftmost leaf */
    while (current->left != NULL) {
        current = current->left;
    }
    return(current->data);
}

/* Driver program to test sameTree function*/
int main()
{
    struct node* root = NULL;
    root = insert(root, 4);
    insert(root, 2);
    insert(root, 1);
    insert(root, 3);
    insert(root, 6);
    insert(root, 5);

    printf("\n Minimum value in BST is %d", minValue(root));
    getchar();
    return 0;
}

```

Java

```

// Java program to find minimum value node in Binary Search Tree

// A binary tree node
class Node {

    int data;
    Node left, right;

    Node(int d) {
        data = d;
        left = right = null;
    }
}

class BinaryTree {

    static Node head;

    /* Given a binary search tree and a number,
    inserts a new node with the given number in
    the correct place in the tree. Returns the new
    root pointer which the caller should then use
    (the standard trick to avoid using reference
    parameters). */
    Node insert(Node node, int data) {

        /* 1. If the tree is empty, return a new,
        single node */
        if (node == null) {
            return (new Node(data));
        } else {

            /* 2. Otherwise, recur down the tree */
            if (data <= node.data) {
                node.left = insert(node.left, data);
            } else {
                node.right = insert(node.right, data);
            }

            /* return the (unchanged) node pointer */
            return node;
        }
    }

    /* Given a non-empty binary search tree,
    return the minimum data value found in that
    tree. Note that the entire tree does not need
    to be searched. */
    int minValue(Node node) {
        Node current = node;

        /* loop down to find the leftmost leaf */
        while (current.left != null) {
            current = current.left;
        }
        return (current.data);
    }

    // Driver program to test above functions
    public static void main(String[] args) {
        BinaryTree tree = new BinaryTree();
        Node root = null;
        root = tree.insert(root, 4);
        tree.insert(root, 2);
        tree.insert(root, 1);
        tree.insert(root, 3);
        tree.insert(root, 6);
        tree.insert(root, 5);

        System.out.println("The minimum value of BST is " + tree.minValue(root));
    }
}

```

Python

```
# Python program to find the node with minimum value in bst

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, key):
        self.data = key
        self.left = None
        self.right = None

    """ Give a binary search tree and a number,
    inserts a new node with the given number in
    the correct place in the tree. Returns the new
    root pointer which the caller should then use
    (the standard trick to avoid using reference
    parameters). """
    def insert(node, data):

        # 1. If the tree is empty, return a new,
        # single node
        if node is None:
            return (Node(data))

        else:
            # 2. Otherwise, recur down the tree
            if data <= node.data:
                node.left = insert(node.left, data)
            else:
                node.right = insert(node.right, data)

        # Return the (unchanged) node pointer
        return node

    """ Given a non-empty binary search tree,
    return the minimum data value found in that
    tree. Note that the entire tree does not need
    to be searched. """
    def minValue(node):
        current = node

        # loop down to find the leftmost leaf
        while (current.left is not None):
            current = current.left

        return current.data

# Driver program
root = None
root = insert(root, 4)
insert(root, 2)
insert(root, 1)
insert(root, 3)
insert(root, 6)
insert(root, 5)

print "\nMinimum value in BST is %d" %(minValue(root))

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Time Complexity: O(n) Worst case happens for left skewed trees.

Similarly we can get the maximum value by recursively traversing the right node of a binary search tree.

References:

<http://cslibrary.stanford.edu/110/BinaryTrees.html>

GATE CS Corner Company Wise Coding Practice

Binary Search Tree

Inorder predecessor and successor for a given key in BST

I recently encountered with a question in an interview at e-commerce company. The interviewer asked the following question:

There is BST given with root node with key part as integer only. The structure of each node is as follows:

```
struct Node
{
    int key;
    struct Node *left, *right ;
};
```

You need to find the inorder successor and predecessor of a given key. In case the given key is not found in BST, then return the two values within which this key will lie.

Following is the algorithm to reach the desired result. Its a recursive method:

```
Input: root node, key
output: predecessor node, successor node

1. If root is NULL
   then return
2. if key is found then
   a. if its left subtree is not null
      Then predecessor will be the right most
      child of left subtree or left child itself.
   b. if its right subtree is not null
      The successor will be the left most child
      of right subtree or right child itself.
   return
3. If key is smaller then root node
   set the successor as root
```

```

    search recursively into left subtree
else
    set the predecessor as root
    search recursively into right subtree

```

Following is C++ implementation of the above algorithm:

C++

```

// C++ program to find predecessor and successor in a BST
#include <iostream>
using namespace std;

// BST Node
struct Node
{
    int key;
    struct Node *left, *right;
};

// This function finds predecessor and successor of key in BST.
// It sets pre and suc as predecessor and successor respectively
void findPreSuc(Node* root, Node*& pre, Node*& suc, int key)
{
    // Base case
    if (root == NULL) return ;

    // If key is present at root
    if (root->key == key)
    {
        // the maximum value in left subtree is predecessor
        if (root->left != NULL)
        {
            Node* tmp = root->left;
            while (tmp->right)
                tmp = tmp->right;
            pre = tmp ;
        }

        // the minimum value in right subtree is successor
        if (root->right != NULL)
        {
            Node* tmp = root->right ;
            while (tmp->left)
                tmp = tmp->left ;
            suc = tmp ;
        }
        return ;
    }

    // If key is smaller than root's key, go to left subtree
    if (root->key > key)
    {
        suc = root ;
        findPreSuc(root->left, pre, suc, key) ;
    }
    else // go to right subtree
    {
        pre = root ;
        findPreSuc(root->right, pre, suc, key) ;
    }
}

// A utility function to create a new BST node
Node *newNode(int item)
{
    Node *temp = new Node;
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

/* A utility function to insert a new node with given key in BST */
Node* insert(Node* node, int key)
{
    if (node == NULL) return newNode(key);
    if (key < node->key)
        node->left = insert(node->left, key);
    else
        node->right = insert(node->right, key);
    return node;
}

// Driver program to test above function
int main()
{
    int key = 65; //Key to be searched in BST

    /* Let us create following BST
        50
       / \
      30  70
     / \ / \
    20 40 60 80 */
    Node *root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    Node* pre = NULL, *suc = NULL;

    findPreSuc(root, pre, suc, key);
    if (pre != NULL)
        cout << "Predecessor is " << pre->key << endl;
    else
        cout << "No Predecessor";

    if (suc != NULL)
        cout << "Successor is " << suc->key;
}

```

```

else
    cout << "No Successor";
    return 0;
}

```

Python

```

# Python program to find predecessor and successor in a BST

```

```

# A BST node
class Node:

```

```

    # Constructor to create a new node
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

```

```

# This function finds predecessor and successor of key in BST
# It sets pre and suc as predecessor and successor respectively
def findPreSuc(root, key):

```

```

    # Base Case
    if root is None:
        return

    # If key is present at root
    if root.key == key:

        # the maximum value in left subtree is predecessor
        if root.left is not None:
            tmp = root.left
            while(tmp.right):
                tmp = tmp.right
            findPreSuc.pre = tmp

        # the minimum value in right subtree is successor
        if root.right is not None:
            tmp = root.right
            while(tmp.left):
                tmp = tmp.left
            findPreSuc.suc = tmp

    return

```

```

    # If key is smaller than root's key, go to left subtree
    if root.key > key :
        findPreSuc.suc = root
        findPreSuc(root.left, key)

```

```

    else: # go to right subtree
        findPreSuc.pre = root
        findPreSuc(root.right, key)

```

```

# A utility function to insert a new node in with given key in BST
def insert(node , key):
    if node is None:
        return Node(key)

```

```

    if key < node.key:
        node.left = insert(node.left, key)

```

```

    else:
        node.right = insert(node.right, key)

```

```

    return node

```

```

# Driver program to test above function
key = 65 #Key to be searched in BST

```

```

""" Let us create following BST
      50
     /  \
    30   70
   / \  / \
  20 40 60 80
"""

```

```

root = None
root = insert(root, 50)
insert(root, 30);
insert(root, 20);
insert(root, 40);
insert(root, 70);
insert(root, 60);
insert(root, 80);

```

```

# Static variables of the function findPreSuc
findPreSuc.pre = None
findPreSuc.suc = None

```

```

findPreSuc(root, key)

```

```

if findPreSuc.pre is not None:
    print "Predecessor is", findPreSuc.pre.key

```

```

else:
    print "No Predecessor"

```

```

if findPreSuc.suc is not None:
    print "Successor is", findPreSuc.suc.key
else:
    print "No Successor"

```

```

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

```

Predecessor is 60

```

This article is contributed by **algoLover**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Binary Search Tree

A program to check if a binary tree is BST or not

A binary search tree (BST) is a node based binary tree data structure which has the following properties.

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

From the above properties it naturally follows that:

- Each node (item in the tree) has a distinct key.

BST



METHOD 1 (Simple but Wrong)

Following is a simple program. For each node, check if left node of it is smaller than the node and right node of it is greater than the node.

```
int isBST(struct node* node)
{
    if (node == NULL)
        return 1;

    /* false if left is > than node */
    if (node->left != NULL && node->left->data > node->data)
        return 0;

    /* false if right is < than node */
    if (node->right != NULL && node->right->data < node->data)
        return 0;

    /* false if, recursively, the left or right is not a BST */
    if (!isBST(node->left) || !isBST(node->right))
        return 0;

    /* passing all that, it's a BST */
    return 1;
}
```

This approach is wrong as this will return true for below binary tree (and below tree is not a BST because 4 is in left subtree of 3)

tree_bst



METHOD 2 (Correct but not efficient)

For each node, check if max value in left subtree is smaller than the node and min value in right subtree greater than the node.

```
/* Returns true if a binary tree is a binary search tree */
int isBST(struct node* node)
{
    if (node == NULL)
        return(true);

    /* false if the max of the left is > than us */
    if (node->left != NULL && maxValue(node->left) > node->data)
        return(false);

    /* false if the min of the right is <= than us */
    if (node->right != NULL && minValue(node->right) < node->data)
        return(false);

    /* false if, recursively, the left or right is not a BST */
    if (!isBST(node->left) || !isBST(node->right))
        return(false);

    /* passing all that, it's a BST */
    return(true);
}
```

It is assumed that you have helper functions `minValue()` and `maxValue()` that return the min or max int value from a non-empty tree

METHOD 3 (Correct and Efficient)

Method 2 above runs slowly since it traverses over some parts of the tree many times. A better solution looks at each node only once. The trick is to write a utility helper function `isBSTUtil(struct node* node, int min, int max)` that traverses down the tree keeping track of the narrowing min and max allowed values as it goes, looking at each node only once. The initial values for min and max should be `INT_MIN` and `INT_MAX` — they narrow from there.

```
/* Returns true if the given tree is a binary search tree
(efficient version). */
int isBST(struct node* node)
{
    return(isBSTUtil(node, INT_MIN, INT_MAX));
}

/* Returns true if the given tree is a BST and its
values are >= min and
Implementation:
```

C

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

int isBSTUtil(struct node* node, int min, int max);

/* Returns true if the given tree is a binary search tree
(efficient version). */
int isBST(struct node* node)
{
    return(isBSTUtil(node, INT_MIN, INT_MAX));
}

/* Returns true if the given tree is a BST and its
values are >= min and <= max. */
int isBSTUtil(struct node* node, int min, int max)
{
    /* an empty tree is BST */
    if (node==NULL)
        return 1;

    /* false if this node violates the min/max constraint */
    if (node->data < min || node->data > max)
        return 0;

    /* otherwise check the subtrees recursively,
    tightening the min or max constraint */
    return
        isBSTUtil(node->left, min, node->data-1) && // Allow only distinct values
        isBSTUtil(node->right, node->data+1, max); // Allow only distinct values
}

/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Driver program to test above functions*/
int main()
{
    struct node *root = newNode(4);
    root->left = newNode(2);
    root->right = newNode(5);
    root->left->left = newNode(1);
    root->left->right = newNode(3);

    if(isBST(root))
        printf("Is BST");
    else
        printf("Not a BST");

    getchar();
    return 0;
}
```

Java

```
//Java implementation to check if given Binary tree
//is a BST or not

/* Class containing left and right child of current
node and key value*/
class Node
{
    int data;
    Node left, right;

    public Node(int item)
    {
        data = item;
        left = right = null;
    }
}

public class BinaryTree
{
    //Root of the Binary Tree
    Node root;

    /* can give min and max value according to your code or
```

can write a function to find min and max value of tree. */

```
/* returns true if given search tree is binary
search tree (efficient version) */
boolean isBST() {
    return isBSTUtil(root, Integer.MIN_VALUE,
        Integer.MAX_VALUE);
}

/* Returns true if the given tree is a BST and its
values are >= min and <= max. */
boolean isBSTUtil(Node node, int min, int max)
{
    /* an empty tree is BST */
    if (node == null)
        return true;

    /* false if this node violates the min/max constraints */
    if (node.data < min || node.data > max)
        return false;

    /* otherwise check the subtrees recursively
    tightening the min/max constraints */
    // Allow only distinct values
    return (isBSTUtil(node.left, min, node.data-1) &&
        isBSTUtil(node.right, node.data+1, max));
}

/* Driver program to test above functions */
public static void main(String args[])
{
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(4);
    tree.root.left = new Node(2);
    tree.root.right = new Node(5);
    tree.root.left.left = new Node(1);
    tree.root.left.right = new Node(3);

    if (tree.isBST())
        System.out.println("IS BST");
    else
        System.out.println("Not a BST");
}
}
```

Python

Python program to check if a binary tree is bst or not

```
INT_MAX = 4294967296
INT_MIN = -4294967296
```

A binary tree node
class Node:

```
    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
```

```
# Returns true if the given tree is a binary search tree
# (efficient version)
def isBST(node):
    return (isBSTUtil(node, INT_MIN, INT_MAX))
```

```
# Return true if the given tree is a BST and its values
# >= min and <= max
def isBSTUtil(node, mini, maxi):
```

```
    # An empty tree is BST
    if node is None:
        return True
```

```
    # False if this node violates min/max constraint
    if node.data < mini or node.data > maxi:
        return False
```

```
    # Otherwise check the subtrees recursively
    # tightening the min or max constraint
    return (isBSTUtil(node.left, mini, node.data - 1) and
        isBSTUtil(node.right, node.data + 1, maxi))
```

```
# Driver program to test above function
root = Node(4)
root.left = Node(2)
root.right = Node(5)
root.left.left = Node(1)
root.left.right = Node(3)
```

```
if (isBST(root)):
    print "Is BST"
else:
    print "Not a BST"
```

This code is contributed by Nikhil Kumar Singh(nickzuck_007)

Time Complexity: O(n)

Auxiliary Space : O(1) if Function Call Stack size is not considered, otherwise O(n)

METHOD 4(Using In-Order Traversal)

Thanks to [LJW489](#) for suggesting this method.

1) Do In-Order Traversal of the given tree and store the result in a temp array.

3) Check if the temp array is sorted in ascending order, if it is, then the tree is BST.

Time Complexity: $O(n)$

We can avoid the use of Auxiliary Array. While doing In-Order traversal, we can keep track of previously visited node. If the value of the currently visited node is less than the previous value, then tree is not BST. Thanks to [ygos](#) for this spa

C

```
bool isBST(struct node* root)
{
    static struct node *prev = NULL;

    // traverse the tree in inorder fashion and keep track of prev node
    if (root)
    {
        if (!isBST(root->left))
            return false;

        // Allows only distinct valued nodes
        if (prev != NULL && root->data <= prev->data)
            return false;

        prev = root;

        return isBST(root->right);
    }

    return true;
}
```

Java

```
// Java implementation to check if given Binary tree
// is a BST or not

/* Class containing left and right child of current
node and key value*/
class Node
{
    int data;
    Node left, right;

    public Node(int item)
    {
        data = item;
        left = right = null;
    }
}

public class BinaryTree
{
    // Root of the Binary Tree
    Node root;

    // To keep track of previous node in Inorder Traversal
    Node prev;

    boolean isBST() {
        prev = null;
        return isBST(root);
    }

    /* Returns true if given search tree is binary
    search tree (efficient version) */
    boolean isBST(Node node)
    {
        // traverse the tree in inorder fashion and
        // keep a track of previous node
        if (node != null)
        {
            if (!isBST(node.left))
                return false;

            // allows only distinct values node
            if (prev != null && node.data <= prev.data )
                return false;
            prev = node;
            return isBST(node.right);
        }
        return true;
    }

    /* Driver program to test above functions */
    public static void main(String args[])
    {
        BinaryTree tree = new BinaryTree();
        tree.root = new Node(4);
        tree.root.left = new Node(2);
        tree.root.right = new Node(5);
        tree.root.left.left = new Node(1);
        tree.root.left.right = new Node(3);

        if (tree.isBST())
```

```

        System.out.println("IS BST");
    else
        System.out.println("Not a BST");
    }
}

```

The use of static variable can also be avoided by using reference to prev node as a parameter (Similar to [this post](#)).

Sources:

http://en.wikipedia.org/wiki/Binary_search_tree

<http://cslibrary.stanford.edu/110/BinaryTrees.html>

Please write comments if you find any bug in the above programs/algorithms or other ways to solve the same problem.

GATE CS Corner
Company Wise Coding Practice

Binary Search Tree

Lowest Common Ancestor in a Binary Search Tree.

Given values of two nodes in a Binary Search Tree, write a c program to find the Lowest Common Ancestor (LCA). You may assume that both the values exist in the tree.

The function prototype should be as follows:

```

struct node *lca(struct node* root, int n1, int n2)
n1 and n2 are two given values in the tree with given root.

```

BST_LCA



For example, consider the BST in diagram, LCA of 10 and 14 is 12 and LCA of 8 and 14 is 8.

Following is definition of LCA from Wikipedia:

Let T be a rooted tree. The lowest common ancestor between two nodes n1 and n2 is defined as the lowest node in T that has both n1 and n2 as descendants (where we allow a node to be a descendant of itself).

The LCA of n1 and n2 in T is the shared ancestor of n1 and n2 that is located farthest from the root. Computation of lowest common ancestors may be useful, for instance, as part of a procedure for determining the distance between pairs of nodes in a tree: the distance from n1 to n2 can be computed as the distance from the root to n1, plus the distance from the root to n2, minus twice the distance from the root to their lowest common ancestor. (Source [Wiki](#))

Solutions:

If we are given a BST where every node has **parent pointer**, then LCA can be easily determined by traversing up using parent pointer and printing the first intersecting node.

We can solve this problem using BST properties. We can **recursively traverse** the BST from root. The main idea of the solution is, while traversing from top to bottom, the first node n we encounter with value between n1 and n2, i.e., $n1 < n < n2$ or same as one of the n1 or n2, is LCA of n1 and n2 (assuming that $n1 < n2$). So just recursively traverse the BST in, if node's value is greater than both n1 and n2 then our LCA lies in left side of the node, if it's smaller than both n1 and n2, then LCA lies on right side. Otherwise root is LCA (assuming that both n1 and n2 are present in BST)

C

```

// A recursive C program to find LCA of two nodes n1 and n2.

```

```

#include <stdio.h>
#include <stdlib.h>

struct node
{
    int data;
    struct node* left, *right;
};

/* Function to find LCA of n1 and n2. The function assumes that both
n1 and n2 are present in BST */
struct node *lca(struct node* root, int n1, int n2)
{
    if (root == NULL) return NULL;

    // If both n1 and n2 are smaller than root, then LCA lies in left
    if (root->data > n1 && root->data > n2)
        return lca(root->left, n1, n2);

    // If both n1 and n2 are greater than root, then LCA lies in right
    if (root->data < n1 && root->data < n2)
        return lca(root->right, n1, n2);

    return root;
}

/* Helper function that allocates a new node with the given data.*/
struct node* newNode(int data)
{
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->data = data;
    node->left = node->right = NULL;
    return(node);
}

/* Driver program to test lca() */
int main()
{
    // Let us construct the BST shown in the above figure
    struct node *root = newNode(20);
    root->left = newNode(8);
    root->right = newNode(22);
    root->left->left = newNode(4);
    root->left->right = newNode(12);
    root->left->right->left = newNode(10);
    root->left->right->right = newNode(14);

    int n1 = 10, n2 = 14;
    struct node *t = lca(root, n1, n2);
    printf("LCA of %d and %d is %d\n", n1, n2, t->data);

    n1 = 14, n2 = 8;
    t = lca(root, n1, n2);
    printf("LCA of %d and %d is %d\n", n1, n2, t->data);

    n1 = 10, n2 = 22;
    t = lca(root, n1, n2);
    printf("LCA of %d and %d is %d\n", n1, n2, t->data);

    getchar();
    return 0;
}

```

Java

```

// Recursive Java program to print Lca of two nodes

// A binary tree node
class Node
{
    int data;
    Node left, right;

    Node(int item)
    {
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    /* Function to find LCA of n1 and n2. The function assumes that both
n1 and n2 are present in BST */
    Node lca(Node node, int n1, int n2)
    {
        if (node == null)
            return null;

        // If both n1 and n2 are smaller than root, then LCA lies in left
        if (node.data > n1 && node.data > n2)
            return lca(node.left, n1, n2);

        // If both n1 and n2 are greater than root, then LCA lies in right
        if (node.data < n1 && node.data < n2)
            return lca(node.right, n1, n2);

        return node;
    }

    /* Driver program to test lca() */
    public static void main(String args[])
    {
        // Let us construct the BST shown in the above figure
        BinaryTree tree = new BinaryTree();
        tree.root = new Node(20);
        tree.root.left = new Node(8);
        tree.root.right = new Node(22);
        tree.root.left.left = new Node(4);
        tree.root.left.right = new Node(12);
        tree.root.left.right.left = new Node(10);
    }
}

```

```

tree.root.left.right.right = new Node(14);

int n1 = 10, n2 = 14;
Node t = tree.lca(tree.root, n1, n2);
System.out.println("LCA of " + n1 + " and " + n2 + " is " + t.data);

n1 = 14;
n2 = 8;
t = tree.lca(tree.root, n1, n2);
System.out.println("LCA of " + n1 + " and " + n2 + " is " + t.data);

n1 = 10;
n2 = 22;
t = tree.lca(tree.root, n1, n2);
System.out.println("LCA of " + n1 + " and " + n2 + " is " + t.data);
}
}

// This code has been contributed by Mayank Jaiswal

```

Python

```

# A recursive python program to find LCA of two nodes
# n1 and n2

# A Binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

# Function to find LCA of n1 and n2. The function assumes
# that both n1 and n2 are present in BST
def lca(root, n1, n2):

    # Base Case
    if root is None:
        return None

    # If both n1 and n2 are smaller than root, then LCA
    # lies in left
    if (root.data > n1 and root.data > n2):
        return lca(root.left, n1, n2)

    # If both n1 and n2 are greater than root, then LCA
    # lies in right
    if (root.data < n1 and root.data < n2):
        return lca(root.right, n1, n2)

    return root

# Driver program to test above function

# Let us construct the BST shown in the figure
root = Node(20)
root.left = Node(8)
root.right = Node(22)
root.left.left = Node(4)
root.left.right = Node(12)
root.left.right.left = Node(10)
root.left.right.right = Node(14)

n1 = 10 ; n2 = 14
t = lca(root, n1, n2)
print "LCA of %d and %d is %d" %(n1, n2, t.data)

n1 = 14 ; n2 = 8
t = lca(root, n1, n2)
print "LCA of %d and %d is %d" %(n1, n2, t.data)

n1 = 10 ; n2 = 22
t = lca(root, n1, n2)
print "LCA of %d and %d is %d" %(n1, n2, t.data)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

```

LCA of 10 and 14 is 12
LCA of 14 and 8 is 8
LCA of 10 and 22 is 20

```

Time complexity of above solution is $O(h)$ where h is height of tree. Also, the above solution requires $O(h)$ extra space in function call stack for recursive function calls. We can avoid extra space using **iterative solution**.

```

/* Function to find LCA of n1 and n2. The function assumes that both
n1 and n2 are present in BST */
struct node *lca(struct node* root, int n1, int n2)
{
    while (root != NULL)
    {
        // If both n1 and n2 are smaller than root, then LCA lies in left
        if (root->data > n1 && root->data > n2)
            root = root->left;

        // If both n1 and n2 are greater than root, then LCA lies in right
        else if (root->data < n1 && root->data < n2)
            root = root->right;

        else break;
    }
    return root;
}

```

See [this](#) for complete program.

You may like to see below articles as well :

[Lowest Common Ancestor in a Binary Tree](#)

[LCA using Parent Pointer](#)

[Find LCA in Binary Tree using RMQ](#)

Exercise

The above functions assume that n_1 and n_2 both are in BST. If n_1 and n_2 are not present, then they may return incorrect result. Extend the above solutions to return NULL if n_1 or n_2 or both not present in BST.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

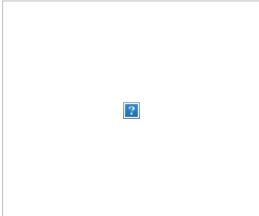
GATE CS Corner Company Wise Coding Practice

[Trees](#)
[LCA](#)

Inorder Successor in Binary Search Tree

In Binary Tree, Inorder successor of a node is the next node in Inorder traversal of the Binary Tree. Inorder Successor is NULL for the last node in Inorder traversal.

In Binary Search Tree, Inorder Successor of an input node can also be defined as the node with the smallest key greater than the key of input node. So, it is sometimes important to find next node in sorted order.



In the above diagram, inorder successor of **8** is **10**, inorder successor of **10** is **12** and inorder successor of **14** is **20**.

Method 1 (Uses Parent Pointer)

In this method, we assume that every node has parent pointer.

The Algorithm is divided into two cases on the basis of right subtree of the input node being empty or not.

Input: *node*, *root* // *node* is the node whose Inorder successor is needed.

output: *succ* // *succ* is Inorder successor of *node*.

1) If right subtree of *node* is not NULL, then *succ* lies in right subtree. Do following.

Go to right subtree and return the node with minimum key value in right subtree.

2) If right subtree of *node* is NULL, then *succ* is one of the ancestors. Do following.

Travel up using the parent pointer until you see a node which is left child of its parent. The parent of such a node is the *succ*.

Implementation

Note that the function to find InOrder Successor is highlighted (with gray background) in below code.

C

```
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
    struct node* parent;
};

struct node * minValue(struct node* node);

struct node * inOrderSuccessor(struct node *root, struct node *n)
{
    // step 1 of the above algorithm
    if (n->right != NULL )
        return minValue(n->right);

    // step 2 of the above algorithm
    struct node *p = n->parent;
    while(p != NULL && n == p->right)
    {
        n = p;
        p = p->parent;
    }
    return p;
}

/* Given a non-empty binary search tree, return the minimum data
value found in that tree. Note that the entire tree does not need
to be searched. */
struct node * minValue(struct node* node) {
    struct node* current = node;

    /* loop down to find the leftmost leaf */
    while (current->left != NULL) {
        current = current->left;
    }
    return current;
}

/* Helper function that allocates a new node with the given data and
NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    node->parent = NULL;

    return(node);
}
```

```

/* Give a binary search tree and a number, inserts a new node with
the given number in the correct place in the tree. Returns the new
root pointer which the caller should then use (the standard trick to
avoid using reference parameters). */
struct node* insert(struct node* node, int data)
{
    /* 1. If the tree is empty, return a new,
    single node */
    if (node == NULL)
        return(newNode(data));
    else
    {
        struct node *temp;

        /* 2. Otherwise, recur down the tree */
        if (data <= node->data)
        {
            temp = insert(node->left, data);
            node->left = temp;
            temp->parent= node;
        }
        else
        {
            temp = insert(node->right, data);
            node->right = temp;
            temp->parent = node;
        }
    }

    /* return the (unchanged) node pointer */
    return node;
}

/* Driver program to test above functions*/
int main()
{
    struct node* root = NULL, *temp, *succ, *min;

    //creating the tree given in the above diagram
    root = insert(root, 20);
    root = insert(root, 8);
    root = insert(root, 22);
    root = insert(root, 4);
    root = insert(root, 12);
    root = insert(root, 10);
    root = insert(root, 14);
    temp = root->left->right->right;

    succ = inOrderSuccessor(root, temp);
    if(succ != NULL)
        printf("In order Successor of %d is %d ", temp->data, succ->data);
    else
        printf("In order Successor doesn't exist");

    getchar();
    return 0;
}

```

Java

```

// Java program to find minimum value node in Binary Search Tree

// A binary tree node
class Node {

    int data;
    Node left, right, parent;

    Node(int d) {
        data = d;
        left = right = parent = null;
    }
}

class BinaryTree {

    static Node head;

    /* Given a binary search tree and a number,
    inserts a new node with the given number in
    the correct place in the tree. Returns the new
    root pointer which the caller should then use
    (the standard trick to avoid using reference
    parameters). */
    Node insert(Node node, int data) {

        /* 1. If the tree is empty, return a new,
        single node */
        if (node == null) {
            return (new Node(data));
        } else {

            Node temp = null;

            /* 2. Otherwise, recur down the tree */
            if (data <= node.data) {
                temp = insert(node.left, data);
                node.left = temp;
                temp.parent = node;
            } else {
                temp = insert(node.right, data);
                node.right = temp;
                temp.parent = node;
            }

            /* return the (unchanged) node pointer */
            return node;
        }
    }

    Node inOrderSuccessor(Node root, Node n) {

```

```

// step 1 of the above algorithm
if (n.right != null) {
    return minValue(n.right);
}

// step 2 of the above algorithm
Node p = n.parent;
while (p != null && n == p.right) {
    n = p;
    p = p.parent;
}
return p;
}

/* Given a non-empty binary search tree, return the minimum data
value found in that tree. Note that the entire tree does not need
to be searched. */
Node minValue(Node node) {
    Node current = node;

    /* loop down to find the leftmost leaf */
    while (current.left != null) {
        current = current.left;
    }
    return current;
}

// Driver program to test above functions
public static void main(String[] args) {
    BinaryTree tree = new BinaryTree();
    Node root = null, temp = null, suc = null, min = null;
    root = tree.insert(root, 20);
    root = tree.insert(root, 8);
    root = tree.insert(root, 22);
    root = tree.insert(root, 4);
    root = tree.insert(root, 12);
    root = tree.insert(root, 10);
    root = tree.insert(root, 14);
    temp = root.left.right.right;
    suc = tree.inOrderSuccessor(root, temp);
    if (suc != null) {
        System.out.println("Inorder successor of " + temp.data +
                           " is " + suc.data);
    } else {
        System.out.println("Inorder successor does not exist");
    }
}

// This code has been contributed by Mayank Jaiswal

```

Python

```

# Python program to find the inorder successor in a BST

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, key):
        self.data = key
        self.left = None
        self.right = None

def inOrderSuccessor(root, n):

    # Step 1 of the above algorithm
    if n.right is not None:
        return minValue(n.right)

    # Step 2 of the above algorithm
    p = n.parent
    while( p is not None):
        if n != p.right :
            break
        n = p
        p = p.parent
    return p

# Given a non-empty binary search tree, return the
# minimum data value found in that tree. Note that the
# entire tree doesn't need to be searched
def minValue(node):
    current = node

    # loop down to find the leftmost leaf
    while(current is not None):
        if current.left is None:
            break
        current = current.data

    return current

# Given a binary search tree and a number, inserts a
# new node with the given number in the correct place
# in the tree. Returns the new root pointer which the
# caller should then use( the standard trick to avoid
# using reference parameters)
def insert( node, data):

    # 1) If tree is empty then return a new singly node
    if node is None:
        return Node(data)
    else:

        # 2) Otherwise, recur down the tree
        if data <= node.data:
            temp = insert(node.left, data)
            node.left = temp
            temp.parent = node

```

```

else:
    temp = insert(node.right, data)
    node.right = temp
    temp.parent = node

# return the unchanged node pointer
return node

# Driver program to test above function

root = None

# Creating the tree given in the above diagram
root = insert(root, 20)
root = insert(root, 8);
root = insert(root, 22);
root = insert(root, 4);
root = insert(root, 12);
root = insert(root, 10);
root = insert(root, 14);
temp = root.left.right.right

succ = inOrderSuccessor( root, temp)
if succ is not None:
    print "\nInorder Successor of %d is %d \"
        %(temp.data , succ.data)
else:
    print "\nInorder Successor doesn't exist"

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output of the above program:
Inorder Successor of 14 is 20
Time Complexity: $O(h)$ where h is height of tree.

Method 2 (Search from root)

Parent pointer is NOT needed in this algorithm. The Algorithm is divided into two cases on the basis of right subtree of the input node being empty or not.

Input: *node*, *root* // *node* is the node whose Inorder successor is needed.

output: *succ* // *succ* is Inorder successor of *node*.

- 1) If right subtree of *node* is not *NULL*, then *succ* lies in right subtree. Do following.
Go to right subtree and return the node with minimum key value in right subtree.
- 2) If right subtree of *node* is *NULL*, then start from root and use search like technique. Do following.
Travel down the tree, if a node's data is greater than root's data then go right side, otherwise go to left side.

```

struct node * inOrderSuccessor(struct node *root, struct node *n)
{
    // step 1 of the above algorithm
    if( n->right != NULL )
        return minValue(n->right);

    struct node *succ = NULL;

    // Start from root and search for successor down the tree
    while (root != NULL)
    {
        if (n->data < root->data)
        {
            succ = root;
            root = root->left;
        }
        else if (n->data > root->data)
            root = root->right;
        else
            break;
    }

    return succ;
}

```

Thanks to [R.Srinivasan](#) for suggesting this method.

Time Complexity: $O(h)$ where h is height of tree.

References:

<http://net.pku.edu.cn/~course/cs101/2007/resource/Intro2Algorithm/book6/chap13.htm>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

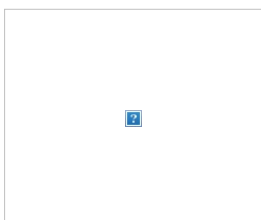
GATE CS Corner Company Wise Coding Practice

Binary Search Tree

Find k-th smallest element in BST (Order Statistics in BST)

Given root of binary search tree and K as input, find K -th smallest element in BST.

For example, in the following BST, if $k = 3$, then output should be 10, and if $k = 5$, then output should be 14.



Method 1: Using Inorder Traversal.

Inorder traversal of BST retrieves elements of tree in the sorted order. The inorder traversal uses stack to store to be explored nodes of tree (threaded tree avoids stack and recursion for traversal, see [this post](#)). The idea is to keep track of popped elements which participate in the order statistics. Hypothetical algorithm is provided below,

Time complexity: $O(n)$ where n is total nodes in tree..

Algorithm:

```
/* initialization */
pCrawl = root
set initial stack element as NULL (sentinal)

/* traverse upto left extreme */
while(pCrawl is valid )
    stack.push(pCrawl)
    pCrawl = pCrawl.left

/* process other nodes */
while( pCrawl = stack.pop() is valid )
    stop if sufficient number of elements are popped.
    if( pCrawl.right is valid )
        pCrawl = pCrawl.right
        while( pCrawl is valid )
            stack.push(pCrawl)
        pCrawl = pCrawl.left
```

Implementation:

```
#include <stdio.h>
#include <stdlib.h>

#define ARRAY_SIZE(arr) sizeof(arr)/sizeof(arr[0])

/* just add elements to test */
/* NOTE: A sorted array results in skewed tree */
int ele[] = { 20, 8, 22, 4, 12, 10, 14 };

/* same alias */
typedef struct node_t node_t;

/* Binary tree node */
struct node_t
{
    int data;

    node_t* left;
    node_t* right;
};

/* simple stack that stores node addresses */
typedef struct stack_t stack_t;

/* initial element always NULL, uses as sentinel */
struct stack_t
{
    node_t* base[ARRAY_SIZE(ele) + 1];
    int stackIndex;
};

/* pop operation of stack */
node_t* pop(stack_t* st)
{
    node_t* ret = NULL;

    if( st && st->stackIndex > 0 )
    {
        ret = st->base[st->stackIndex];
        st->stackIndex--;
    }

    return ret;
}

/* push operation of stack */
void push(stack_t* st, node_t* node)
{
    if( st )
    {
        st->stackIndex++;
        st->base[st->stackIndex] = node;
    }
}

/* Iterative insertion
Recursion is least preferred unless we gain something
*/
node_t* insert_node(node_t* root, node_t* node)
{
    /* A crawling pointer */
    node_t* pTraverse = root;
    node_t* currentParent = root;

    // Traverse till appropriate node
    while(pTraverse)
    {
        currentParent = pTraverse;

        if( node->data < pTraverse->data )
        {
            /* left subtree */
            pTraverse = pTraverse->left;
        }
        else
        {
            /* right subtree */
            pTraverse = pTraverse->right;
        }
    }

    /* If the tree is empty, make it as root node */
    if( !root )
    {
        root = node;
    }
    else if( node->data < currentParent->data )
    {
        /* Insert on left side */
        currentParent->left = node;
    }
}
```

```

else
{
    /* Insert on right side */
    currentParent->right = node;
}

return root;
}

/* Elements are in an array. The function builds binary tree */
node_t* binary_search_tree(node_t*root, int keys[], int const size)
{
    int iterator;
    node_t*new_node = NULL;

    for(iterator = 0; iterator < size; iterator++)
    {
        new_node = (node_t*)malloc( sizeof(node_t) );

        /* initialize */
        new_node->data = keys[iterator];
        new_node->left = NULL;
        new_node->right = NULL;

        /* insert into BST */
        root = insert_node(root, new_node);
    }

    return root;
}

node_t* k_smallest_element_inorder(stack_t*stack, node_t*root, int k)
{
    stack_t*st = stack;
    node_t*pCrawl = root;

    /* move to left extremen (minimum) */
    while( pCrawl )
    {
        push(st, pCrawl);
        pCrawl = pCrawl->left;
    }

    /* pop off stack and process each node */
    while( pCrawl = pop(st) )
    {
        /* each pop operation emits one element
        in the order
        */
        if( !k )
        {
            /* loop testing */
            st->stackIndex = 0;
            break;
        }

        /* there is right subtree */
        if( pCrawl->right )
        {
            /* push the left subtree of right subtree */
            pCrawl = pCrawl->right;
            while( pCrawl )
            {
                push(st, pCrawl);
                pCrawl = pCrawl->left;
            }
        }

        /* pop off stack and repeat */
    }
}

/* node having k-th element or NULL node */
return pCrawl;
}

/* Driver program to test above functions */
int main(void)
{
    node_t* root = NULL;
    stack_t stack = { {0}, 0 };
    node_t*kNode = NULL;

    int k = 5;

    /* Creating the tree given in the above diagram */
    root = binary_search_tree(root, ele, ARRAY_SIZE(ele));

    kNode = k_smallest_element_inorder(&stack, root, k);

    if( kNode )
    {
        printf("kth smallest elment for k = %d is %d", k, kNode->data);
    }
    else
    {
        printf("There is no such element");
    }

    getchar();
    return 0;
}

```

Method 2: Augmented Tree Data Structure.

The idea is to maintain rank of each node. We can keep track of elements in a subtree of any node while building the tree. Since we need K-th smallest element, we can maintain number of elements of left subtree in every node.

Assume that the root is having N nodes in its left subtree. If $K = N + 1$, root is K-th node. If $K < N$, we will continue our search (recursion) for the Kth smallest element in the left subtree of root. If $K > N + 1$, we continue our search in the right subtree for the $(K - N - 1)$ -th smallest element. Note that we need the count of elements in left subtree only.

Time complexity: $O(h)$ where h is height of tree.

Algorithm:

```

start:
if K = rootLeftElement + 1

```

```

root node is the K th node.
goto stop
else if K > root.leftElements
K = K - (root.leftElements + 1)
root = root.right
goto start
else
root = root.left
goto srart

```

stop:

Implementation:

```

#include <stdio.h>
#include <stdlib.h>

#define ARRAY_SIZE(arr) sizeof(arr)/sizeof(arr[0])

typedef struct node_t node_t;

/* Binary tree node */
struct node_t
{
    int data;
    int lCount;

    node_t * left;
    node_t * right;
};

/* Iterative insertion
Recursion is least preferred unless we gain something
*/
node_t *insert_node(node_t *root, node_t * node)
{
    /* A crawling pointer */
    node_t *pTraverse = root;
    node_t *currentParent = root;

    // Traverse till appropriate node
    while(pTraverse)
    {
        currentParent = pTraverse;

        if( node->data < pTraverse->data )
        {
            /* We are branching to left subtree
            increment node count */
            pTraverse->lCount++;
            /* left subtree */
            pTraverse = pTraverse->left;
        }
        else
        {
            /* right subtree */
            pTraverse = pTraverse->right;
        }
    }

    /* If the tree is empty, make it as root node */
    if( !root )
    {
        root = node;
    }
    else if( node->data < currentParent->data )
    {
        /* Insert on left side */
        currentParent->left = node;
    }
    else
    {
        /* Insert on right side */
        currentParent->right = node;
    }

    return root;
}

/* Elements are in an array. The function builds binary tree */
node_t * binary_search_tree(node_t *root, int keys[], int const size)
{
    int iterator;
    node_t *new_node = NULL;

    for(iterator = 0; iterator < size; iterator++)
    {
        new_node = (node_t *)malloc( sizeof(node_t) );

        /* initialize */
        new_node->data = keys[iterator];
        new_node->lCount = 0;
        new_node->left = NULL;
        new_node->right = NULL;

        /* insert into BST */
        root = insert_node(root, new_node);
    }

    return root;
}

int k_smallest_element(node_t *root, int k)
{
    int ret = -1;

    if( root )
    {
        /* A crawling pointer */
        node_t *pTraverse = root;

        /* Go to k-th smallest */
        while(pTraverse)
        {
            if( (pTraverse->lCount + 1) == k )

```

```

    {
        ret = pTraverse->data;
        break;
    }
    else if( k > pTraverse->lCount )
    {
        /* There are less nodes on left subtree
        Go to right subtree */
        k = k - (pTraverse->lCount + 1);
        pTraverse = pTraverse->right;
    }
    else
    {
        /* The node is on left subtree */
        pTraverse = pTraverse->left;
    }
}
}

return ret;
}

/* Driver program to test above functions */
int main(void)
{
    /* just add elements to test */
    /* NOTE: A sorted array results in skewed tree */
    int ele[] = { 20, 8, 22, 4, 12, 10, 14 };
    int i;
    node_t* root = NULL;

    /* Creating the tree given in the above diagram */
    root = binary_search_tree(root, ele, ARRAY_SIZE(ele));

    /* It should print the sorted array */
    for(i = 1; i <= ARRAY_SIZE(ele); i++)
    {
        printf("\n kth smallest element for k = %d is %d",
            i, k_smallest_element(root, i));
    }

    getchar();
    return 0;
}

```

Thanks to **Venki** for providing post. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Binary Search Tree
Order-Statistics
About Venki
Software Engineer
[View all posts by Venki](#) →

Merge two BSTs with limited extra space

Given two Binary Search Trees(BST), print the elements of both BSTs in sorted form. The expected time complexity is $O(m+n)$ where m is the number of nodes in first tree and n is the number of nodes in second tree. Maximum allowed auxiliary space is $O(\text{height of the first tree} + \text{height of the second tree})$.

Examples:

```

First BST
  3
 / \
1   5
Second BST
  4
 / \
2   6
Output: 1 2 3 4 5 6

```

```

First BST
  8
 / \
2  10
 /
1
Second BST
  5
 /
3
 /
0
Output: 0 1 2 3 5 8 10

```

Source: [Google interview question](#)

A similar question has been discussed earlier. Let us first discuss already discussed methods of the [previous post](#) which was for Balanced BSTs. The method 1 can be applied here also, but the time complexity will be $O(n^2)$ in worst case. The method 2 can also be applied here, but the extra space required will be $O(n)$ which violates the constraint given in this question. Method 3 can be applied here but the step 3 of method 3 can't be done in $O(n)$ for an unbalanced BST.

Thanks to **Kumar** for suggesting the following solution.

The idea is to use [iterative inorder traversal](#). We use two auxiliary stacks for two BSTs. Since we need to print the elements in sorted form, whenever we get a smaller element from any of the trees, we print it. If the element is greater, then we push it back to stack for the next iteration.

```

#include<stdio.h>
#include<stdlib.h>

// Structure of a BST Node
struct node
{
    int data;
    struct node *left;
    struct node *right;
};

//..... START OF STACK RELATED STUFF.....
// A stack node
struct snode
{

```

```

    struct node *t;
    struct snode *next;
};

// Function to add an elem k to stack
void push(struct snode **s, struct node *k)
{
    struct snode *tmp = (struct snode *) malloc(sizeof(struct snode));

    //perform memory check here
    tmp->t = k;
    tmp->next = *s;
    (*s) = tmp;
}

// Function to pop an element t from stack
struct node *pop(struct snode **s)
{
    struct node *t;
    struct snode *st;
    st=*s;
    (*s) = (*s)->next;
    t = st->t;
    free(st);
    return t;
}

// Fuction to check whether the stack is empty or not
int isEmpty(struct snode *s)
{
    if (s == NULL )
        return 1;

    return 0;
}
// ..... END OF STACK RELATED STUFF.....

/* Utility function to create a new Binary Tree node */
struct node* newNode( int data)
{
    struct node *temp = new struct node;
    temp->data = data;
    temp->left = NULL;
    temp->right = NULL;
    return temp;
}

/* A utility function to print Inorder traversal of a Binary Tree */
void inorder(struct node *root)
{
    if (root != NULL)
    {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

// The function to print data of two BSTs in sorted order
void merge(struct node *root1, struct node *root2)
{
    // s1 is stack to hold nodes of first BST
    struct snode *s1 = NULL;

    // Current node of first BST
    struct node *current1 = root1;

    // s2 is stack to hold nodes of second BST
    struct snode *s2 = NULL;

    // Current node of second BST
    struct node *current2 = root2;

    // If first BST is empty, then output is inorder
    // traversal of second BST
    if (root1 == NULL)
    {
        inorder(root2);
        return;
    }

    // If second BST is empty, then output is inorder
    // traversal of first BST
    if (root2 == NULL)
    {
        inorder(root1);
        return ;
    }

    // Run the loop while there are nodes not yet printed.
    // The nodes may be in stack(explored, but not printed)
    // or may be not yet explored
    while (current1 != NULL || !isEmpty(s1) ||
           current2 != NULL || !isEmpty(s2))
    {
        // Following steps follow iterative Inorder Traversal
        if (current1 != NULL || current2 != NULL )
        {
            // Reach the leftmost node of both BSTs and push ancestors of
            // leftmost nodes to stack s1 and s2 respectively
            if (current1 != NULL)
            {
                push(&s1, current1);
                current1 = current1->left;
            }
            if (current2 != NULL)
            {
                push(&s2, current2);
                current2 = current2->left;
            }
        }

        }
    else
    {
        // If we reach a NULL node and either of the stacks is empty,
        // then one tree is exhausted, print the other tree
    }
}

```

```

if (isEmpty(s1))
{
    while (!isEmpty(s2))
    {
        current2 = pop (&s2);
        current2->left = NULL;
        inorder(current2);
    }
    return ;
}
if (isEmpty(s2))
{
    while (!isEmpty(s1))
    {
        current1 = pop (&s1);
        current1->left = NULL;
        inorder(current1);
    }
    return ;
}

// Pop an element from both stacks and compare the
// popped elements
current1 = pop(&s1);
current2 = pop(&s2);

// If element of first tree is smaller, then print it
// and push the right subtree. If the element is larger,
// then we push it back to the corresponding stack.
if (current1->data < current2->data)
{
    printf("%d ", current1->data);
    current1 = current1->right;
    push(&s2, current2);
    current2 = NULL;
}
else
{
    printf("%d ", current2->data);
    current2 = current2->right;
    push(&s1, current1);
    current1 = NULL;
}
}
}

/* Driver program to test above functions */
int main()
{
    struct node *root1 = NULL, *root2 = NULL;

    /* Let us create the following tree as first tree
    3
   / \
  1  5
  */
    root1 = newNode(3);
    root1->left = newNode(1);
    root1->right = newNode(5);

    /* Let us create the following tree as second tree
    4
   / \
  2  6
  */
    root2 = newNode(4);
    root2->left = newNode(2);
    root2->right = newNode(6);

    // Print sorted nodes of both trees
    merge(root1, root2);

    return 0;
}

```

Time Complexity: $O(m+n)$

Auxiliary Space: $O(\text{height of the first tree} + \text{height of the second tree})$

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Binary Search Tree
Merge Sort

Two nodes of a BST are swapped, correct the BST

Two of the nodes of a Binary Search Tree (BST) are swapped. Fix (or correct) the BST.

```

Input Tree:
  10
 / \
5   8
/\
2  20

```

In the above tree, nodes 20 and 8 must be swapped to fix the tree.

Following is the output tree

```

  10
 / \
5   20
/\
2   8

```

The inorder traversal of a BST produces a sorted array. So a **simple method** is to store inorder traversal of the input tree in an auxiliary array. Sort the auxiliary array. Finally, insert the auxiliary array elements back to the BST, keeping the structure of the BST same. Time complexity of this method is $O(n \log n)$ and auxiliary space needed is $O(n)$.

We can solve this in $O(n)$ time and with a single traversal of the given BST. Since inorder traversal of BST is always a sorted array, the problem can be reduced to a problem where two elements of a sorted array are swapped. There are two cases that we need to handle:

1. The swapped nodes are not adjacent in the inorder traversal of the BST.

For example, Nodes 5 and 25 are swapped in {3 5 7 8 10 15 20 25}.
The inorder traversal of the given tree is 3 25 7 8 10 15 20 5

If we observe carefully, during inorder traversal, we find node 7 is smaller than the previous visited node 25. Here save the context of node 25 (previous node). Again, we find that node 5 is smaller than the previous node 20. This time, we save the context of node 5 (current node). Finally swap the two node's values.

2. The swapped nodes are adjacent in the inorder traversal of BST.

For example, Nodes 7 and 8 are swapped in {3 5 7 8 10 15 20 25}.
The inorder traversal of the given tree is 3 5 8 7 10 15 20 25

Unlike case #1, here only one point exists where a node value is smaller than previous node value. e.g. node 7 is smaller than node 8.

How to Solve? We will maintain three pointers, first, middle and last. When we find the first point where current node value is smaller than previous node value, we update the first with the previous node & middle with the current node. When we find the second point where current node value is smaller than previous node value, we update the last with the current node. In case #2, we will never find the second point. So, last pointer will not be updated. After processing, if the last node value is null, then two swapped nodes of BST are adjacent.

Following is C implementation of the given code.

```
// Two nodes in the BST's swapped, correct the BST.
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct node
{
    int data;
    struct node *left, *right;
};

// A utility function to swap two integers
void swap( int* a, int* b )
{
    int t = *a;
    *a = *b;
    *b = t;
}

/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node *)malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return(node);
}

// This function does inorder traversal to find out the two swapped nodes.
// It sets three pointers, first, middle and last. If the swapped nodes are
// adjacent to each other, then first and middle contain the resultant nodes
// Else, first and last contain the resultant nodes
void correctBSTUtil( struct node* root, struct node** first,
                    struct node** middle, struct node** last,
                    struct node** prev )
{
    if( root )
    {
        // Recur for the left subtree
        correctBSTUtil( root->left, first, middle, last, prev );

        // If this node is smaller than the previous node, it's violating
        // the BST rule.
        if (*prev && root->data < (*prev)->data)
        {
            // If this is first violation, mark these two nodes as
            // 'first' and 'middle'
            if ( !*first )
            {
                *first = *prev;
                *middle = root;
            }

            // If this is second violation, mark this node as last
            else
                *last = root;
        }

        // Mark this node as previous
        *prev = root;

        // Recur for the right subtree
        correctBSTUtil( root->right, first, middle, last, prev );
    }
}

// A function to fix a given BST where two nodes are swapped. This
// function uses correctBSTUtil() to find out two nodes and swaps the
// nodes to fix the BST
void correctBST( struct node* root )
{
    // Initialize pointers needed for correctBSTUtil()
    struct node *first, *middle, *last, *prev;
    first = middle = last = prev = NULL;

    // Set the pointers to find out two nodes
    correctBSTUtil( root, &first, &middle, &last, &prev );

    // Fix (or correct) the tree
    if( first && last )
        swap( &(first->data), &(last->data) );
    else if( first && middle ) // Adjacent nodes swapped
        swap( &(first->data), &(middle->data) );

    // else nodes have not been swapped, passed tree is really BST.
}

/* A utility function to print Inorder traversal */
void printInorder(struct node* node)
{
    if (node == NULL)
        return;
}
```

```

    printlnorder(node->left);
    printf("%d ", node->data);
    printlnorder(node->right);
}

/* Driver program to test above functions */
int main()
{
    /* 6
       / \
      10 2
     /\  /\
    1 3 7 12
    10 and 2 are swapped
    */

    struct node *root = newNode(6);
    root->left = newNode(10);
    root->right = newNode(2);
    root->left->left = newNode(1);
    root->left->right = newNode(3);
    root->right->right = newNode(12);
    root->right->left = newNode(7);

    printf("Inorder Traversal of the original tree \n");
    printlnorder(root);

    correctBST(root);

    printf("\nInorder Traversal of the fixed tree \n");
    printlnorder(root);

    return 0;
}

```

Output:

```

Inorder Traversal of the original tree
1 10 3 6 7 2 12
Inorder Traversal of the fixed tree
1 2 3 6 7 10 12

```

Time Complexity: $O(n)$

See [this](#) for different test cases of the above code.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

[Binary Search Tree](#)

Floor and Ceil from a BST

There are numerous applications we need to find floor (cell) value of a key in a binary search tree or sorted array. For example, consider designing memory management system in which free nodes are arranged in BST. Find best fit for the input request.

Ceil Value Node: Node with smallest data larger than or equal to key value.

Imagine we are moving down the tree, and assume we are root node. The comparison yields three possibilities,

- A)** Root data is equal to key. We are done, root data is ceil value.
- B)** Root data < key value, certainly the ceil value can't be in left subtree. Proceed to search on right subtree as reduced problem instance.
- C)** Root data > key value, the ceil value *may* be in left subtree. We may find a node with is larger data than key value in left subtree, if not the root itself will be ceil node.

Here is the code for ceil value.

C

```

// Program to find ceil of a given value in BST
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has key, left child and right child */
struct node
{
    int key;
    struct node* left;
    struct node* right;
};

/* Helper function that allocates a new node with the given key and
   NULL left and right pointers.*/
struct node* newNode(int key)
{
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    return(node);
}

// Function to find ceil of a given input in BST. If input is more
// than the max key in BST, return -1
int Ceil(struct node* root, int input)
{
    // Base case
    if( root == NULL )
        return -1;

    // We found equal key
    if( root->key == input )
        return root->key;

    // If root's key is smaller, ceil must be in right subtree
    if( root->key < input )
        return Ceil(root->right, input);

    // Else, either left subtree or root has the ceil value
    int ceil = Ceil(root->left, input);
}

```



```

    return (ceil >= input) ? ceil : root->key;
}

// Driver program to test above function
int main()
{
    node *root = newNode(8);

    root->left = newNode(4);
    root->right = newNode(12);

    root->left->left = newNode(2);
    root->left->right = newNode(6);

    root->right->left = newNode(10);
    root->right->right = newNode(14);

    for(int i = 0; i < 16; i++)
        printf("%d %d\n", i, Ceil(root, i));

    return 0;
}

```

Java

```

// Java program to find ceil of a given value in BST

class Node {

    int data;
    Node left, right;

    Node(int d) {
        data = d;
        left = right = null;
    }
}

class BinaryTree {

    static Node root;

    // Function to find ceil of a given input in BST. If input is more
    // than the max key in BST, return -1
    int Ceil(Node node, int input) {

        // Base case
        if (node == null) {
            return -1;
        }

        // We found equal key
        if (node.data == input) {
            return node.data;
        }

        // If root's key is smaller, ceil must be in right subtree
        if (node.data < input) {
            return Ceil(node.right, input);
        }

        // Else, either left subtree or root has the ceil value
        int ceil = Ceil(node.left, input);
        return (ceil >= input) ? ceil : node.data;
    }

    // Driver program to test the above functions
    public static void main(String[] args) {
        BinaryTree tree = new BinaryTree();
        tree.root = new Node(8);
        tree.root.left = new Node(4);
        tree.root.right = new Node(12);
        tree.root.left.left = new Node(2);
        tree.root.left.right = new Node(6);
        tree.root.right.left = new Node(10);
        tree.root.right.right = new Node(14);
        for (int i = 0; i < 16; i++) {
            System.out.println(i + " " + tree.Ceil(root, i));
        }
    }
}

// This code has been contributed by Mayank Jaiswal

```

Python

```

# Python program to find ceil of a given value in BST

# A Binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.key = data
        self.left = None
        self.right = None

# Function to find ceil of a given input in BST. If input
# is more than the max key in BST, return -1
def ceil(root, inp):

    # Base Case
    if root == None:
        return -1

    # We found equal key
    if root.key == inp :
        return root.key

    # If root's key is smaller, ceil must be in right subtree

```

```

if root.key < inp:
    return ceil(root.right, inp)

# Else, either left subtree or root has the ceil value
val = ceil(root.left, inp)
return val if val >= inp else root.key

# Driver program to test above function
root = Node(8)

root.left = Node(4)
root.right = Node(12)

root.left.left = Node(2)
root.left.right = Node(6)

root.right.left = Node(10)
root.right.right = Node(14)

for i in range(16):
    print "%d %d" %(i, ceil(root, i))

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

```

0 2
1 2
2 2
3 4
4 4
5 6
6 6
7 8
8 8
9 10
10 10
11 12
12 12
13 14
14 14
15 -1

```

Exercise:

1. Modify above code to find floor value of input key in a binary search tree.
 2. Write neat algorithm to find floor and ceil values in a sorted array. Ensure to handle all possible boundary conditions.
- **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Binary Search Tree
About Venki
 Software Engineer
[View all posts by Venki](#) →

In-place conversion of Sorted DLL to Balanced BST

Given a Doubly Linked List which has data members sorted in ascending order. Construct a [Balanced Binary Search Tree](#) which has same data members as the given Doubly Linked List. The tree must be constructed in-place (No new node should be allocated for tree conversion)

Examples:

```

Input: Doubly Linked List 1 2 3
Output: A Balanced BST
  2
 / \
1   3

Input: Doubly Linked List 1 2 3 4 5 6 7
Output: A Balanced BST
  4
 / \
2   6
/ \ / \
1 3 4 7

Input: Doubly Linked List 1 2 3 4
Output: A Balanced BST
  3
 / \
2   4
/
1

Input: Doubly Linked List 1 2 3 4 5 6
Output: A Balanced BST
  4
 / \
2   6
/ \ /
1 3 5

```

The Doubly Linked List conversion is very much similar to [this Singly Linked List problem](#) and the method 1 is exactly same as the method 1 of [previous post](#). Method 2 is also almost same. The only difference in method 2 is, instead of allocating new nodes for BST, we reuse same DLL nodes. We use prev pointer as left and next pointer as right.

Method 1 (Simple)

Following is a simple algorithm where we first find the middle node of list and make it root of the tree to be constructed.

- 1) Get the Middle of the linked list and make it root.
- 2) Recursively do same for left half and right half.
 - a) Get the middle of left half and make it left child of the root created in step 1.
 - b) Get the middle of right half and make it right child of the root created in step 1.

Time complexity: $O(n \log n)$ where n is the number of nodes in Linked List.

Method 2 (Tricky)

The method 1 constructs the tree from root to leaves. In this method, we construct from leaves to root. The idea is to insert nodes in BST in the same order as they appear in Doubly Linked List, so that the tree can be constructed in $O(n)$ time complexity. We first count the number of nodes in the given Linked List. Let the count be n . After counting nodes, we take left $n/2$ nodes and recursively construct the left subtree. After left subtree is constructed, we assign middle node to root and link the left subtree with root. Finally, we recursively construct the right subtree and link it with root.

While constructing the BST, we also keep moving the list head pointer to next so that we have the appropriate pointer in each recursive call.

Following is C implementation of method 2. The main code which creates Balanced BST is highlighted.

C++

```
#include<stdio.h>
#include<stdlib.h>

/* A Doubly Linked List node that will also be used as a tree node */
struct Node
{
    int data;

    // For tree, next pointer can be used as right subtree pointer
    struct Node* next;

    // For tree, prev pointer can be used as left subtree pointer
    struct Node* prev;
};

// A utility function to count nodes in a Linked List
int countNodes(struct Node *head);

struct Node* sortedListToBSTRecur(struct Node **head_ref, int n);

/* This function counts the number of nodes in Linked List and then calls
sortedListToBSTRecur() to construct BST */
struct Node* sortedListToBST(struct Node *head)
{
    /* Count the number of nodes in Linked List */
    int n = countNodes(head);

    /* Construct BST */
    return sortedListToBSTRecur(&head, n);
}

/* The main function that constructs balanced BST and returns root of it.
head_ref -> Pointer to pointer to head node of Doubly linked list
n -> No. of nodes in the Doubly Linked List */
struct Node* sortedListToBSTRecur(struct Node **head_ref, int n)
{
    /* Base Case */
    if (n <= 0)
        return NULL;

    /* Recursively construct the left subtree */
    struct Node *left = sortedListToBSTRecur(head_ref, n/2);

    /* head_ref now refers to middle node, make middle node as root of BST */
    struct Node *root = *head_ref;

    // Set pointer to left subtree
    root->prev = left;

    /* Change head pointer of Linked List for parent recursive calls */
    *head_ref = (*head_ref)->next;

    /* Recursively construct the right subtree and link it with root
    The number of nodes in right subtree is total nodes - nodes in
    left subtree - 1 (for root) */
    root->next = sortedListToBSTRecur(head_ref, n-n/2-1);

    return root;
}

/* UTILITY FUNCTIONS */
/* A utility function that returns count of nodes in a given Linked List */
int countNodes(struct Node *head)
{
    int count = 0;
    struct Node *temp = head;
    while(temp)
    {
        temp = temp->next;
        count++;
    }
    return count;
}

/* Function to insert a node at the beginning of the Doubly Linked List */
void push(struct Node ** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));

    /* put in the data */
    new_node->data = new_data;

    /* since we are adding at the beginning,
    prev is always NULL */
    new_node->prev = NULL;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* change prev of head node to new node */
    if((*head_ref) != NULL)
        (*head_ref)->prev = new_node;

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Function to print nodes in a given linked list */
void printList(struct Node *node)
{
    while (node!=NULL)
    {

```

```

        printf("%d ", node->data);
        node = node->next;
    }
}

/* A utility function to print preorder traversal of BST */
void preOrder(struct Node* node)
{
    if (node == NULL)
        return;
    printf("%d ", node->data);
    preOrder(node->prev);
    preOrder(node->next);
}

/* Driver program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    /* Let us create a sorted linked list to test the functions
    Created linked list will be 7->6->5->4->3->2->1 */
    push(&head, 7);
    push(&head, 6);
    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 1);

    printf("Given Linked List\n");
    printList(head);

    /* Convert List to BST */
    struct Node *root = sortedListToBST(head);
    printf("\n PreOrder Traversal of constructed BST \n");
    preOrder(root);

    return 0;
}

```

Java

```

class Node
{
    int data;
    Node next, prev;

    Node(int d)
    {
        data = d;
        next = prev = null;
    }
}

class LinkedList
{
    Node head;

    /* This function counts the number of nodes in Linked List
    and then calls sortedListToBSTRecur() to construct BST */
    Node sortedListToBST()
    {
        /*Count the number of nodes in Linked List */
        int n = countNodes(head);

        /* Construct BST */
        return sortedListToBSTRecur(n);
    }

    /* The main function that constructs balanced BST and
    returns root of it.
    n --> No. of nodes in the Doubly Linked List */
    Node sortedListToBSTRecur(int n)
    {
        /* Base Case */
        if (n <= 0)
            return null;

        /* Recursively construct the left subtree */
        Node left = sortedListToBSTRecur(n / 2);

        /* head_ref now refers to middle node,
        make middle node as root of BST */
        Node root = head;

        // Set pointer to left subtree
        root.prev = left;

        /* Change head pointer of Linked List for parent
        recursive calls */
        head = head.next;

        /* Recursively construct the right subtree and link it
        with root. The number of nodes in right subtree is
        total nodes - nodes in left subtree - 1 (for root) */
        root.next = sortedListToBSTRecur(n - n / 2 - 1);

        return root;
    }

    /* UTILITY FUNCTIONS */
    /* A utility function that returns count of nodes in a
    given Linked List */
    int countNodes(Node head)
    {
        int count = 0;
        Node temp = head;
        while (temp != null)
        {
            temp = temp.next;
            count++;
        }
    }
}

```

```

    }
    return count;
}

/* Function to insert a node at the beginging of
the Doubly Linked List */
void push(int new_data)
{
    /* allocate node */
    Node new_node = new Node(new_data);

    /* since we are adding at the beginning,
    prev is always NULL */
    new_node.prev = null;

    /* link the old list off the new node */
    new_node.next = head;

    /* change prev of head node to new node */
    if (head != null)
        head.prev = new_node;

    /* move the head to point to the new node */
    head = new_node;
}

/* Function to print nodes in a given linked list */
void printList()
{
    Node node = head;
    while (node != null)
    {
        System.out.print(node.data + " ");
        node = node.next;
    }
}

/* A utility function to print preorder traversal of BST */
void preOrder(Node node)
{
    if (node == null)
        return;
    System.out.print(node.data + " ");
    preOrder(node.prev);
    preOrder(node.next);
}

/* Drier program to test above functions */
public static void main(String[] args)
{
    LinkedList llist = new LinkedList();

    /* Let us create a sorted linked list to test the functions
    Created linked list will be 7->6->5->4->3->2->1 */
    llist.push(7);
    llist.push(6);
    llist.push(5);
    llist.push(4);
    llist.push(3);
    llist.push(2);
    llist.push(1);

    System.out.println("Given Linked List ");
    llist.printList();

    /* Convert List to BST */
    Node root = llist.sortedListToBST();
    System.out.println("");
    System.out.println("Pre-Order Traversal of constructed BST ");
    llist.preOrder(root);
}
}
// This code has been contributed by Mayank Jaiswal(mayank_24)

```

Output:

```

Given Linked List
1 2 3 4 5 6 7
Pre-Order Traversal of constructed BST
4 2 1 3 6 5 7

```

Time Complexity: $O(n)$

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Linked Lists

Find a pair with given sum in a Balanced BST

Given a Balanced Binary Search Tree and a target sum, write a function that returns true if there is a pair with sum equals to target sum, otherwise return false. Expected time complexity is $O(n)$ and only $O(\log n)$ extra space can be used. Any modification to Binary Search Tree is not allowed. Note that height of a Balanced BST is always $O(\log n)$.



This problem is mainly extension of the [previous post](#). Here we are not allowed to modify the BST.

The **Brute Force Solution** is to consider each pair in BST and check whether the sum equals to X. The time complexity of this solution will be $O(n^2)$.

A **Better Solution** is to create an auxiliary array and store Inorder traversal of BST in the array. The array will be sorted as Inorder traversal of BST always produces sorted data. Once we have the Inorder traversal, we can pair in $O(n)$ time (See [this](#) for details). This solution works in $O(n)$ time, but requires $O(n)$ auxiliary space.

A **space optimized solution** is discussed in [previous post](#). The idea was to first in-place convert BST to Doubly Linked List (DLL), then find pair in sorted DLL in $O(n)$ time. This solution takes $O(n)$ time and $O(\log n)$ extra space, but it modifies the given BST.

The **solution discussed below takes $O(n)$ time, $O(\log n)$ space and doesn't modify BST**. The idea is same as finding the pair in sorted array (See method 1 of [this](#) for details). We traverse BST in Normal Inorder and Reverse Inorder simultaneously. In reverse inorder, we start from the rightmost node which is the maximum value node. In normal inorder, we start from the left most node which is minimum value node. We add sum of current nodes in both traversals and compare this sum with given target sum. If the sum is same as target sum, we return true. If the sum is more than target sum, we move to next node in reverse inorder traversal, otherwise we move to next node in normal inorder traversal. If any of the traversals is finished without finding a pair, we return false. Following is C++ implementation of this approach.

```
/* In a balanced binary search tree isPairPresent two element which sums to
a given value time  $O(n)$  space  $O(\log n)$  */
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 100

// A BST node
struct node
{
    int val;
    struct node *left, *right;
};

// Stack type
struct Stack
{
    int size;
    int top;
    struct node* *array;
};

// A utility function to create a stack of given size
struct Stack* createStack(int size)
{
    struct Stack* stack =
        (struct Stack*) malloc(sizeof(struct Stack));
    stack->size = size;
    stack->top = -1;
    stack->array =
        (struct node**) malloc(stack->size * sizeof(struct node*));
    return stack;
}

// BASIC OPERATIONS OF STACK
int isFull(struct Stack* stack)
{ return stack->top - 1 == stack->size; }

int isEmpty(struct Stack* stack)
{ return stack->top == -1; }

void push(struct Stack* stack, struct node* node)
{
    if (isFull(stack))
        return;
    stack->array[++stack->top] = node;
}

struct node* pop(struct Stack* stack)
{
    if (isEmpty(stack))
        return NULL;
    return stack->array[stack->top--];
}

// Returns true if a pair with target sum exists in BST, otherwise false
bool isPairPresent(struct node *root, int target)
{
    // Create two stacks. s1 is used for normal inorder traversal
    // and s2 is used for reverse inorder traversal
    struct Stack* s1 = createStack(MAX_SIZE);
    struct Stack* s2 = createStack(MAX_SIZE);

    // Note the sizes of stacks is MAX_SIZE, we can find the tree size and
    // fix stack size as  $O(\log n)$  for balanced trees like AVL and Red Black
    // tree. We have used MAX_SIZE to keep the code simple

    // done1, val1 and curr1 are used for normal inorder traversal using s1
    // done2, val2 and curr2 are used for reverse inorder traversal using s2
    bool done1 = false, done2 = false;
    int val1 = 0, val2 = 0;
    struct node *curr1 = root, *curr2 = root;

    // The loop will break when we either find a pair or one of the two
    // traversals is complete
    while (1)
    {
        // Find next node in normal Inorder traversal. See following post
        // http://www.geeksforgeeks.org/inorder-tree-traversal-without-recursion/
        while (done1 == false)
        {
            if (curr1 != NULL)
            {
                push(s1, curr1);
                curr1 = curr1->left;
            }
            else
            {
                if (isEmpty(s1))
                    done1 = 1;
                else
                {
                    curr1 = pop(s1);
                    val1 = curr1->val;
                    curr1 = curr1->right;
                    done1 = 1;
                }
            }
        }

        // Find next node in REVERSE Inorder traversal. The only
        // difference between above and below loop is, in below loop
        // right subtree is traversed before left subtree
        while (done2 == false)
        {
            if (curr2 != NULL)
            {
                push(s2, curr2);
                curr2 = curr2->right;
            }
            else
            {
                if (isEmpty(s2))
                    done2 = 1;
                else
                {
                    curr2 = pop(s2);
                    val2 = curr2->val;
                    curr2 = curr2->left;
                    done2 = 1;
                }
            }
        }

        // If sum of current nodes is equal to target sum, return true
        if (val1 + val2 == target)
            return true;

        // If sum of current nodes is less than target sum, move to next node
        // in normal inorder traversal
        if (val1 + val2 < target)
            curr1 = curr1->right;

        // If sum of current nodes is more than target sum, move to next node
        // in reverse inorder traversal
        if (val1 + val2 > target)
            curr2 = curr2->left;
    }

    return false;
}
```

```

        curr2 = curr2->right;
    }
    else
    {
        if (isEmpty(s2))
            done2 = 1;
        else
        {
            curr2 = pop(s2);
            val2 = curr2->val;
            curr2 = curr2->left;
            done2 = 1;
        }
    }
}

// If we find a pair, then print the pair and return. The first
// condition makes sure that two same values are not added
if ((val1 != val2) && (val1 + val2) == target)
{
    printf("\n Pair Found: %d + %d = %d\n", val1, val2, target);
    return true;
}

// If sum of current values is smaller, then move to next node in
// normal inorder traversal
else if ((val1 + val2) < target)
    done1 = false;

// If sum of current values is greater, then move to next node in
// reverse inorder traversal
else if ((val1 + val2) > target)
    done2 = false;

// If any of the inorder traversals is over, then there is no pair
// so return false
if (val1 >= val2)
    return false;
}
}

// A utility function to create BST node
struct node * NewNode(int val)
{
    struct node *tmp = (struct node *)malloc(sizeof(struct node));
    tmp->val = val;
    tmp->right = tmp->left = NULL;
    return tmp;
}

// Driver program to test above functions
int main()
{
    /*
        15
       /  \
      10   20
     /\   /\
    8 12 16 25 */
    struct node *root = NewNode(15);
    root->left = NewNode(10);
    root->right = NewNode(20);
    root->left->left = NewNode(8);
    root->left->right = NewNode(12);
    root->right->left = NewNode(16);
    root->right->right = NewNode(25);

    int target = 33;
    if (isPairPresent(root, target) == false)
        printf("\n No such values are found\n");

    getch();
    return 0;
}

```

Output:

```
Pair Found: 8 + 25 = 33
```

This article is compiled by [Kumar](#) and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Binary Search Tree

Total number of possible Binary Search Trees with n keys

Total number of possible Binary Search Trees with n different keys = **Catalan number** $C_n = (2n)! / ((n+1)! * n!)$

For n = 0, 1, 2, 3, ... values of Catalan numbers are 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, So are numbers of Binary Search Trees.

Below is code for n'th Catalan number taken from [here](#).

```

// See http://www.geeksforgeeks.org/program-nth-catalan-number/
// for reference of below code.

unsigned long int binomialCoeff(unsigned int n, unsigned int k)
{
    unsigned long int res = 1;

    // Since C(n, k) = C(n, n-k)
    if (k > n - k)
        k = n - k;

    // Calculate value of [n*(n-1)*...*(n-k+1)] / [k*(k-1)*...*1]
    for (int i = 0; i < k; ++i)
    {
        res *= (n - i);
        res /= (i + 1);
    }

    return res;
}

```

```

}

// A Binomial coefficient based function to find nth catalan
// number in O(n) time
unsigned long int catalan(unsigned int n)
{
    // Calculate value of 2nCn
    unsigned long int c = binomialCoeff(2*n, n);

    // return 2nCn/(n+1)
    return c/(n+1);
}

```

Here is a systematic way to enumerate these BSTs. Consider all possible binary search trees with each element at the root. If there are n nodes, then for each choice of root node, there are $n - 1$ non-root nodes and these non-root nodes must be partitioned into those that are less than a chosen root and those that are greater than the chosen root.

Let's say node i is chosen to be the root. Then there are $i - 1$ nodes smaller than i and $n - i$ nodes bigger than i . For each of these two sets of nodes, there is a certain number of possible subtrees.

Let $t(n)$ be the total number of BSTs with n nodes. The total number of BSTs with i at the root is $t(i - 1) t(n - i)$. The two terms are multiplied together because the arrangements in the left and right subtrees are independent. That is, for each arrangement in the left tree and for each arrangement in the right tree, you get one BST with i at the root.

Summing over i gives the total number of binary search trees with n nodes.



The base case is $t(0) = 1$ and $t(1) = 1$, i.e. there is one empty BST and there is one BST with one node.



This article is contributed by Shubham Agarwal. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Binary Search Tree
catalan

Merge Two Balanced Binary Search Trees

You are given two balanced binary search trees e.g., AVL or Red Black Tree. Write a function that merges the two given balanced BSTs into a balanced binary search tree. Let there be m elements in first tree and n elements in the other tree. Your merge function should take $O(m+n)$ time.

In the following solutions, it is assumed that sizes of trees are also given as input. If the size is not given, then we can get the size by traversing the tree (See [this](#)).

Method 1 (Insert elements of first tree to second)

Take all elements of first BST one by one, and insert them into the second BST. Inserting an element to a self balancing BST takes $\text{Log}n$ time (See [this](#)) where n is size of the BST. So time complexity of this method is $\text{Log}(n) + \text{Log}(n+1) \dots \text{Log}(m+n-1)$. The value of this expression will be between $m\text{Log}n$ and $m\text{Log}(m+n-1)$. As an optimization, we can pick the smaller tree as first tree.

Method 2 (Merge Inorder Traversals)

- 1) Do inorder traversal of first tree and store the traversal in one temp array `arr1[]`. This step takes $O(m)$ time.
- 2) Do inorder traversal of second tree and store the traversal in another temp array `arr2[]`. This step takes $O(n)$ time.
- 3) The arrays created in step 1 and 2 are sorted arrays. Merge the two sorted arrays into one array of size $m + n$. This step takes $O(m+n)$ time.
- 4) Construct a balanced tree from the merged array using the technique discussed in [this](#) post. This step takes $O(m+n)$ time.

Time complexity of this method is $O(m+n)$ which is better than method 1. This method takes $O(m+n)$ time even if the input BSTs are not balanced.

Following is C++ implementation of this method.

```

#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

// A utility function to merge two sorted arrays into one
int *merge(int arr1[], int arr2[], int m, int n);

// A helper function that stores inorder traversal of a tree in inorder array
void storeInorder(struct node* node, int inorder[], int *index_ptr);

/* A function that constructs Balanced Binary Search Tree from a sorted array
   See http://www.geeksforgeeks.org/archives/17138 */
struct node* sortedArrayToBST(int arr[], int start, int end);

/* This function merges two balanced BSTs with roots as root1 and root2.
   m and n are the sizes of the trees respectively */
struct node* mergeTrees(struct node *root1, struct node *root2, int m, int n)
{
    // Store inorder traversal of first tree in an array arr1[]
    int *arr1 = new int[m];
    int i = 0;
    storeInorder(root1, arr1, &i);

    // Store inorder traversal of second tree in another array arr2[]
    int *arr2 = new int[n];
    int j = 0;
    storeInorder(root2, arr2, &j);

    // Merge the two sorted array into one
    int *mergedArr = merge(arr1, arr2, m, n);

    // Construct a tree from the merged array and return root of the tree
    return sortedArrayToBST(mergedArr, 0, m+n-1);
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */

```



```

struct node* newNode(int data)
{
    struct node* node = (struct node*)
        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

// A utility function to print inorder traversal of a given binary tree
void printInorder(struct node* node)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    printInorder(node->left);

    printf("%d ", node->data);

    /* now recur on right child */
    printInorder(node->right);
}

// A utility uncton to merge two sorted arrays into one
int *merge(int arr1[], int arr2[], int m, int n)
{
    // mergedArr[] is going to contain result
    int *mergedArr = new int[m + n];
    int i = 0, j = 0, k = 0;

    // Traverse through both arrays
    while (i < m && j < n)
    {
        // Pick the smaler element and put it in mergedArr
        if (arr1[i] < arr2[j])
        {
            mergedArr[k] = arr1[i];
            i++;
        }
        else
        {
            mergedArr[k] = arr2[j];
            j++;
        }
        k++;
    }

    // If there are more elements in first array
    while (i < m)
    {
        mergedArr[k] = arr1[i];
        i++; k++;
    }

    // If there are more elements in second array
    while (j < n)
    {
        mergedArr[k] = arr2[j];
        j++; k++;
    }

    return mergedArr;
}

// A helper function that stores inorder traversal of a tree rooted with node
void storeInorder(struct node* node, int inorder[], int *index_ptr)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    storeInorder(node->left, inorder, index_ptr);

    inorder[*index_ptr] = node->data;
    (*index_ptr)++; // increase index for next entry

    /* now recur on right child */
    storeInorder(node->right, inorder, index_ptr);
}

/* A function that constructs Balanced Binary Search Tree from a sorted array
See http://www.geeksforgeeks.org/archives/17138 */
struct node* sortedArrayToBST(int arr[], int start, int end)
{
    /* Base Case */
    if (start > end)
        return NULL;

    /* Get the middle element and make it root */
    int mid = (start + end)/2;
    struct node *root = newNode(arr[mid]);

    /* Recursively construct the left subtree and make it
    left child of root */
    root->left = sortedArrayToBST(arr, start, mid-1);

    /* Recursively construct the right subtree and make it
    right child of root */
    root->right = sortedArrayToBST(arr, mid+1, end);

    return root;
}

/* Driver program to test above functions*/
int main()
{
    /* Create following tree as first balanced BST
    100
    / \
    50 300
    /\
    20 70
    */

```

```

struct node *root1 = newNode(100);
root1->left = newNode(50);
root1->right = newNode(300);
root1->left->left = newNode(20);
root1->left->right = newNode(70);

/* Create following tree as second balanced BST
      80
     /\
    40 120
*/
struct node *root2 = newNode(80);
root2->left = newNode(40);
root2->right = newNode(120);

struct node *mergedTree = mergeTrees(root1, root2, 5, 3);

printf ("Following is Inorder traversal of the merged tree \n");
printInorder(mergedTree);

getchar();
return 0;
}

```

Output:

```

Following is Inorder traversal of the merged tree
20 40 50 70 80 100 120 300

```

Method 3 (In-Place Merge using DLL)

We can use a Doubly Linked List to merge trees in place. Following are the steps.

- 1) Convert the given two Binary Search Trees into doubly linked list in place (Refer [this post](#) for this step).
- 2) Merge the two sorted Linked Lists (Refer [this post](#) for this step).
- 3) Build a Balanced Binary Search Tree from the merged list created in step 2. (Refer [this post](#) for this step)

Time complexity of this method is also $O(m+n)$ and this method does conversion in place.

Thanks to Dheeraj and Ronzii for suggesting this method.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Binary Search Tree
Self-Balancing BST

Binary Tree to Binary Search Tree Conversion

Given a Binary Tree, convert it to a Binary Search Tree. The conversion must be done in such a way that keeps the original structure of Binary Tree.

Examples.

Example 1

```

Input:
      10
     /\
    2  7
   /\
  8  4
Output:
      8
     /\
    4 10
   /\
  2  7

```

Example 2

```

Input:
      10
     /\
    30 15
   /\
  20  5
Output:
      15
     /\
    10 20
   /\
  5   30

```

Solution

Following is a 3 step solution for converting Binary tree to Binary Search Tree.

- 1) Create a temp array arr[] that stores inorder traversal of the tree. This step takes $O(n)$ time.
- 2) Sort the temp array arr[]. Time complexity of this step depends upon the sorting algorithm. In the following implementation, Quick Sort is used which takes (n^2) time. This can be done in $O(n \log n)$ time using Heap Sort or Merge Sort.
- 3) Again do inorder traversal of tree and copy array elements to tree nodes one by one. This step takes $O(n)$ time.

Following is C implementation of the above approach. The main function to convert is highlighted in the following code.

C

```

/* A program to convert Binary Tree to Binary Search Tree */
#include<stdio.h>
#include<stdlib.h>

/* A binary tree node structure */
struct node
{
    int data;
    struct node *left;
    struct node *right;
};

/* A helper function that stores inorder traversal of a tree rooted
with node */
void storeInorder (struct node* node, int inorder[], int *index_ptr)

```

```

{
    // Base Case
    if (node == NULL)
        return;

    /* first store the left subtree */
    storeInorder (node->left, inorder, index_ptr);

    /* Copy the root's data */
    inorder[*index_ptr] = node->data;
    (*index_ptr)++; // increase index for next entry

    /* finally store the right subtree */
    storeInorder (node->right, inorder, index_ptr);
}

/* A helper function to count nodes in a Binary Tree */
int countNodes (struct node* root)
{
    if (root == NULL)
        return 0;
    return countNodes (root->left) +
        countNodes (root->right) + 1;
}

// Following function is needed for library function qsort()
int compare (const void * a, const void * b)
{
    return ( *(int*)a - *(int*)b );
}

/* A helper function that copies contents of arr[] to Binary Tree.
This function basically does Inorder traversal of Binary Tree and
one by one copy arr[] elements to Binary Tree nodes */
void arrayToBST (int *arr, struct node* root, int *index_ptr)
{
    // Base Case
    if (root == NULL)
        return;

    /* first update the left subtree */
    arrayToBST (arr, root->left, index_ptr);

    /* Now update root's data and increment index */
    root->data = arr[*index_ptr];
    (*index_ptr)++;

    /* finally update the right subtree */
    arrayToBST (arr, root->right, index_ptr);
}

// This function converts a given Binary Tree to BST
void binaryTreeToBST (struct node *root)
{
    // base case: tree is empty
    if (root == NULL)
        return;

    /* Count the number of nodes in Binary Tree so that
    we know the size of temporary array to be created */
    int n = countNodes (root);

    // Create a temp array arr[] and store inorder traversal of tree in arr[]
    int *arr = new int[n];
    int i = 0;
    storeInorder (root, arr, &i);

    // Sort the array using library function for quick sort
    qsort (arr, n, sizeof(arr[0]), compare);

    // Copy array elements back to Binary Tree
    i = 0;
    arrayToBST (arr, root, &i);

    // delete dynamically allocated memory to avoid memory leak
    delete [] arr;
}

/* Utility function to create a new Binary Tree node */
struct node* newNode (int data)
{
    struct node *temp = new struct node;
    temp->data = data;
    temp->left = NULL;
    temp->right = NULL;
    return temp;
}

/* Utility function to print inorder traversal of Binary Tree */
void printInorder (struct node* node)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    printInorder (node->left);

    /* then print the data of node */
    printf("%d ", node->data);

    /* now recur on right child */
    printInorder (node->right);
}

/* Driver function to test above functions */
int main()
{
    struct node *root = NULL;

    /* Constructing tree given in the above figure
    10
    / \
    30 15
    / \
    20 5 */
    root = newNode(10);

```

```

root->left = newNode(30);
root->right = newNode(15);
root->left->left = newNode(20);
root->right->right = newNode(5);

// convert Binary Tree to BST
binaryTreeToBST (root);

printf("Following is Inorder Traversal of the converted BST:\n");
printInorder (root);

return 0;
}

```

Python

```

# Program to convert binary tree to BST

# A binary tree node
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

# Helper function to store the inorder traversal of a tree
def storeInorder(root, inorder):

    # Base Case
    if root is None:
        return

    # First store the left subtree
    storeInorder(root.left, inorder)

    # Copy the root's data
    inorder.append(root.data)

    # Finally store the right subtree
    storeInorder(root.right, inorder)

# A helper function to count nodes in a binary tree
def countNodes(root):
    if root is None:
        return 0

    return countNodes(root.left) + countNodes(root.right) + 1

# Helper function that copies contents of sorted array
# to Binary tree
def arrayToBST(arr, root):

    # Base Case
    if root is None:
        return

    # First update the left subtree
    arrayToBST(arr, root.left)

    # now update root's data delete the value from array
    root.data = arr[0]
    arr.pop(0)

    # Finally update the right subtree
    arrayToBST(arr, root.right)

# This function converts a given binary tree to BST
def binaryTreeToBST(root):

    # Base Case: Tree is empty
    if root is None:
        return

    # Count the number of nodes in Binary Tree so that
    # we know the size of temporary array to be created
    n = countNodes(root)

    # Create the temp array and store the inorder traversal
    # of tree
    arr = []
    storeInorder(root, arr)

    # Sort the array
    arr.sort()

    # copy array elements back to binary tree
    arrayToBST(arr, root)

# Print the inorder traversal of the tree
def printInorder(root):
    if root is None:
        return
    printInorder(root.left)
    print root.data
    printInorder(root.right)

# Driver program to test above function
root = Node(10)
root.left = Node(30)
root.right = Node(15)
root.left.left = Node(20)
root.right.right = Node(5)

# Convert binary tree to BST
binaryTreeToBST(root)

print "Following is the inorder traversal of the converted BST"
printInorder(root)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

Following is Inorder Traversal of the converted BST:
5 10 15 20 30

We will be covering another method for this problem which converts the tree using $O(\text{height of tree})$ extra space.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Binary Search Tree

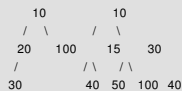
Binary Heap

A Binary Heap is a Binary Tree with following properties.

1) It's a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array.

2) A Binary Heap is either Min Heap or Max Heap. In a Min Binary Heap, the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree. Max Binary Heap is similar to Min Heap.

Examples of Min Heap:



How is Binary Heap represented?

A Binary Heap is a Complete Binary Tree. A binary heap is typically represented as array. Please refer below article for details.

[Array Representation Of Binary Heap](#)

Applications of Heaps:

1) **Heap Sort:** Heap Sort uses Binary Heap to sort an array in $O(n \log n)$ time.

2) **Priority Queue:** Priority queues can be efficiently implemented using Binary Heap because it supports insert(), delete() and extractmax(), decreaseKey() operations in $O(\log n)$ time. Binomial Heap and Fibonacci Heap are variations of Binary Heap. These variations perform union also efficiently.

3) **Graph Algorithms:** The priority queues are especially used in Graph Algorithms like [Dijkstra's Shortest Path](#) and [Prim's Minimum Spanning Tree](#).

4) Many problems can be efficiently solved using Heaps. See following for example.

a) [K'th Largest Element in an array.](#)

b) [Sort an almost sorted array/](#)

c) [Merge K Sorted Arrays.](#)

Operations on Min Heap:

1) **getMini():** It returns the root element of Min Heap. Time Complexity of this operation is $O(1)$.

2) **extractMin():** Removes the minimum element from Min Heap. Time Complexity of this Operation is $O(\log n)$ as this operation needs to maintain the heap property (by calling heapify()) after removing root.

3) **decreaseKey():** Decreases value of key. Time complexity of this operation is $O(\log n)$. If the decreases key value of a node is greater than parent of the node, then we don't need to do anything. Otherwise, we need to traverse up to fix the violated heap property.

4) **insert():** Inserting a new key takes $O(\log n)$ time. We add a new key at the end of the tree. IF new key is greater than its parent, then we don't need to do anything. Otherwise, we need to traverse up to fix the violated heap property.

5) **delete():** Deleting a key also takes $O(\log n)$ time. We replace the key to be deleted with minus infinite by calling decreaseKey(). After decreaseKey(), the minus infinite value must reach root, so we call extractMin() to remove key.

Following is C++ implementation of basic heap operations.

C++

```
// A C++ program to demonstrate common Binary Heap Operations
#include<iostream>
#include<climits>
using namespace std;

// Prototype of a utility function to swap two integers
void swap(int *x, int *y);

// A class for Min Heap
class MinHeap
{
    int *harr; // pointer to array of elements in heap
    int capacity; // maximum possible size of min heap
    int heap_size; // Current number of elements in min heap
public:
    // Constructor
    MinHeap(int capacity);

    // to heapify a subtree with root at given index
    void MinHeapify(int i);

    int parent(int i) { return (i-1)/2; }

    // to get index of left child of node at index i
    int left(int i) { return (2*i + 1); }

    // to get index of right child of node at index i
    int right(int i) { return (2*i + 2); }

    // to extract the root which is the minimum element
    int extractMin();

    // Decreases key value of key at index i to new_val
    void decreaseKey(int i, int new_val);

    // Returns the minimum key (key at root) from min heap
    int getMin() { return harr[0]; }

    // Deletes a key stored at index i
    void deleteKey(int i);

    // Inserts a new key 'k'
    void insertKey(int k);
```

```

};

// Constructor: Builds a heap from a given array a[] of given size
MinHeap::MinHeap(int cap)
{
    heap_size = 0;
    capacity = cap;
    harr = new int[cap];
}

// Inserts a new key 'k'
void MinHeap::insertKey(int k)
{
    if (heap_size == capacity)
    {
        cout << "nOverflow: Could not insertKey\n";
        return;
    }

    // First insert the new key at the end
    heap_size++;
    int i = heap_size - 1;
    harr[i] = k;

    // Fix the min heap property if it is violated
    while (i != 0 && harr[parent(i)] > harr[i])
    {
        swap(&harr[i], &harr[parent(i)]);
        i = parent(i);
    }
}

// Decreases value of key at index 'i' to new_val. It is assumed that
// new_val is smaller than harr[i].
void MinHeap::decreaseKey(int i, int new_val)
{
    harr[i] = new_val;
    while (i != 0 && harr[parent(i)] > harr[i])
    {
        swap(&harr[i], &harr[parent(i)]);
        i = parent(i);
    }
}

// Method to remove minimum element (or root) from min heap
int MinHeap::extractMin()
{
    if (heap_size <= 0)
        return INT_MAX;
    if (heap_size == 1)
    {
        heap_size--;
        return harr[0];
    }

    // Store the minimum value, and remove it from heap
    int root = harr[0];
    harr[0] = harr[heap_size-1];
    heap_size--;
    MinHeapify(0);

    return root;
}

// This function deletes key at index i. It first reduced value to minus
// infinite, then calls extractMin()
void MinHeap::deleteKey(int i)
{
    decreaseKey(i, INT_MIN);
    extractMin();
}

// A recursive method to heapify a subtree with root at given index
// This method assumes that the subtrees are already heapified
void MinHeap::MinHeapify(int i)
{
    int l = left(i);
    int r = right(i);
    int smallest = i;
    if (l < heap_size && harr[l] < harr[i])
        smallest = l;
    if (r < heap_size && harr[r] < harr[smallest])
        smallest = r;
    if (smallest != i)
    {
        swap(&harr[i], &harr[smallest]);
        MinHeapify(smallest);
    }
}

// A utility function to swap two elements
void swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

// Driver program to test above functions
int main()
{
    MinHeap h(11);
    h.insertKey(3);
    h.insertKey(2);
    h.deleteKey(1);
    h.insertKey(15);
    h.insertKey(5);
    h.insertKey(4);
    h.insertKey(45);
    cout << h.extractMin() << " ";
    cout << h.getMin() << " ";
    h.decreaseKey(2, 1);
    cout << h.getMin();
    return 0;
}

```

Python

```
# A Python program to demonstrate common binary heap operations

# Import the heap functions from python library
from heapq import heappush, heappop, heapify

# heappop - pop and return the smallest element from heap
# heappush - push the value item onto the heap, maintaining
#           heap invariant
# heapify - transform list into heap, in place, in linear time

# A class for Min Heap
class MinHeap:

    # Constructor to initialize a heap
    def __init__(self):
        self.heap = []

    def parent(self, i):
        return (i-1)/2

    # Inserts a new key 'k'
    def insertKey(self, k):
        heappush(self.heap, k)

    # Decrease value of key at index 'i' to new_val
    # It is assumed that new_val is smaller than heap[i]
    def decreaseKey(self, i, new_val):
        self.heap[i] = new_val
        while(i != 0 and self.heap[self.parent(i)] > self.heap[i]):
            # Swap heap[i] with heap[parent(i)]
            self.heap[i], self.heap[self.parent(i)] = (
                self.heap[self.parent(i)], self.heap[i])

    # Method to remove minimum element from min heap
    def extractMin(self):
        return heappop(self.heap)

    # This function deletes key at index i. It first reduces
    # value to minus infinite and then calls extractMin()
    def deleteKey(self, i):
        self.decreaseKey(i, float("-inf"))
        self.extractMin()

    # Get the minimum element from the heap
    def getMin(self):
        return self.heap[0]

# Driver program to test above function
heapObj = MinHeap()
heapObj.insertKey(3)
heapObj.insertKey(2)
heapObj.deleteKey(1)
heapObj.insertKey(15)
heapObj.insertKey(5)
heapObj.insertKey(4)
heapObj.insertKey(45)

print heapObj.extractMin(),
print heapObj.getMin(),
heapObj.decreaseKey(2, 1)
print heapObj.getMin()

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

```
2 4 1
```

[Coding Practice on Heap](#)

[All Articles on Heap](#)

[Quiz on Heap](#)

[PriorityQueue : Binary Heap Implementation in Java Library](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: Tree

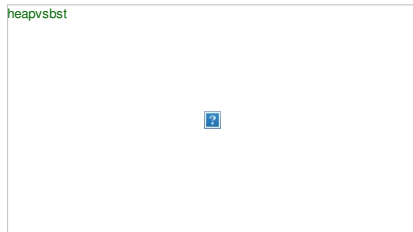
Why is Binary Heap Preferred over BST for Priority Queue?

A typical **Priority Queue** requires following operations to be efficient.

1. Get Top Priority Element (Get minimum or maximum)
2. Insert an element
3. Remove top priority element
4. Decrease Key

A **Binary Heap** supports above operations with following time complexities:

1. $O(1)$
2. $O(\log n)$
3. $O(\log n)$
4. $O(\log n)$



A Self Balancing Binary Search Tree like **AVL Tree**, **Red-Black Tree**, etc can also support above operations with same time complexities.

1. Finding minimum and maximum are not naturally $O(1)$, but can be easily implemented in $O(1)$ by keeping an extra pointer to minimum or maximum and updating the pointer with insertion and deletion if required. With deletion we can update by finding inorder predecessor or successor.
2. Inserting an element is naturally $O(\log n)$
3. Removing maximum or minimum are also $O(\log n)$
4. Decrease key can be done in $O(\log n)$ by doing a deletion followed by insertion. See [this](#) for details.

So why is Binary Heap Preferred for Priority Queue?

- Since Binary Heap is implemented using arrays, there is always better locality of reference and operations are more cache friendly.
- Although operations are of same time complexity, constants in Binary Search Tree are higher.
- We can build a Binary Heap in $O(n)$ time. Self Balancing BSTs require $O(n \log n)$ time to construct.
- Binary Heap doesn't require extra space for pointers.
- Binary Heap is easier to implement.
- There are variations of Binary Heap like Fibonacci Heap that can support insert and decrease-key in $O(1)$ time

Is Binary Heap always better?

Although Binary Heap is for Priority Queue, BSTs have their own advantages and the list of advantages is in-fact bigger compared to binary heap.

- Searching an element in self-balancing BST is $O(\log n)$ which is $O(n)$ in Binary Heap.
- We can print all elements of BST in sorted order in $O(n)$ time, but Binary Heap requires $O(n \log n)$ time.
- **Floor and ceil** can be found in $O(\log n)$ time.
- **K'th largest/smallest element** be found in $O(\log n)$ time by augmenting tree with an additional field.

This article is contributed by **Vivek Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Heap
Self-Balancing-BST

Binomial Heap

The main application of **Binary Heap** is as implement priority queue. Binomial Heap is to extension of **Binary Heap** that provides faster union or merge operation together with other operations provided by Binary Heap.

A **Binomial Heap** is a collection of **Binomial Trees**

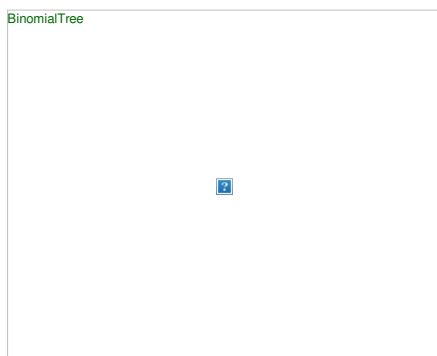
What is a Binomial Tree?

A Binomial Tree of order 0 has 1 node. A Binomial Tree of order k can be constructed by taking two binomial trees of order k-1, and making one as leftmost child of other.

A Binomial Tree of order k has following properties.

- a) It has exactly 2^k nodes.
- b) It has depth as k.
- c) There are exactly $\binom{k}{i}$ nodes at depth i for $i = 0, 1, \dots, k$.
- d) The root has degree k and children of root are themselves Binomial Trees with order k-1, k-2, ... 0 from left to right.

The following diagram is taken from 2nd Edition of **CLRS book**.

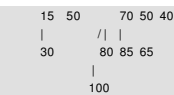


Binomial Heap:

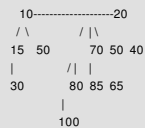
A Binomial Heap is a set of Binomial Trees where each Binomial Tree follows Min Heap property. And there can be at-most one Binomial Tree of any degree.

Examples Binomial Heap:

12-----10-----20
/ \ / \



A Binomial Heap with 13 nodes. It is a collection of 3 Binomial Trees of orders 0, 2 and 3 from left to right.



A Binomial Heap with 12 nodes. It is a collection of 2 Binomial Trees of orders 2 and 3 from left to right.

Binary Representation of a number and Binomial Heaps

A Binomial Heap with n nodes has number of Binomial Trees equal to the number of set bits in Binary representation of n . For example let n be 13, there 3 set bits in binary representation of n (00001101), hence 3 Binomial Trees. We can also relate degree of these Binomial Trees with positions of set bits. With this relation we can conclude that there are $O(\log n)$ Binomial Trees in a Binomial Heap with ' n ' nodes.

Operations of Binomial Heap:

The main operation in Binomial Heap is union(), all other operations mainly use this operation. The union() operation is to combine two Binomial Heaps into one. Let us first discuss other operations, we will discuss union later.

- 1) insert(H, k): Inserts a key ' k ' to Binomial Heap ' H '. This operation first creates a Binomial Heap with single key ' k ', then calls union on H and the new Binomial heap.
- 2) getMin(H): A simple way to getMin() is to traverse the list of root of Binomial Trees and return the minimum key. This implementation requires $O(\log n)$ time. It can be optimized to $O(1)$ by maintaining a pointer to minimum key root.
- 3) extractMin(H): This operation also uses union(). We first call getMin() to find the minimum key Binomial Tree, then we remove the node and create a new Binomial Heap by connecting all subtrees of the removed minimum node. Finally we call union() on H and the newly created Binomial Heap. This operation requires $O(\log n)$ time.
- 4) delete(H): Like Binary Heap, delete operation first reduces the key to minus infinite, then calls extractMin().
- 5) decreaseKey(H): decreaseKey() is also similar to Binary Heap. We compare the decreases key with it parent and if parent's key is more, we swap keys and recur for parent. We stop when we either reach a node whose parent has smaller key or we hit the root node. Time complexity of decreaseKey() is $O(\log n)$.

Union operation in Binomial Heap:

Given two Binomial Heaps H_1 and H_2 , union(H_1, H_2) creates a single Binomial Heap.

- 1) The first step is to simply merge the two Heaps in non-decreasing order of degrees. In the following diagram, figure(b) shows the result after merging.
 - 2) After the simple merge, we need to make sure that there is at-most one Binomial Tree of any order. To do this, we need to combine Binomial Trees of same order. We traverse the list of merged roots, we keep track of three pointers, prev, x and next- x . There can be following 4 cases when we traverse the list of roots.
 - Case 1: Orders of x and next- x are not same, we simply move ahead.
- In following 3 cases orders of x and next- x are same.
- Case 2: If order of next-next- x is also same, move ahead.
 - Case 3: If key of x is smaller than or equal to key of next- x , then make next- x as a child of x by linking it with x .
 - Case 4: If key of x is greater, then make x as child of next.

The following diagram is taken from 2nd Edition of CLRS book.

BinomialHeapUnion



How to represent Binomial Heap?

A Binomial Heap is a set of Binomial Trees. A Binomial Tree must be represented in a way that allows sequential access to all siblings, starting from the leftmost sibling (We need this in and extractMin() and delete()). The idea is to represent Binomial Trees as leftmost child and right-sibling representation, i.e., every node stores two pointers, one to the leftmost child and other to the right sibling.

Implementation of Binomial Heap

Sources:

Introduction to Algorithms by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.

This article is contributed by **Shivam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Fibonacci Heap | Set 1 (Introduction)

Heaps are mainly used for implementing priority queue. We have discussed below heaps in previous posts.

Binary Heap

Binomial Heap

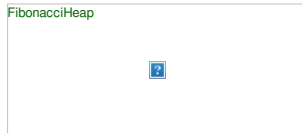
In terms of Time Complexity, Fibonacci Heap beats both Binary and Binomial Heaps.

Below are **amortized time complexities** of **Fibonacci Heap**.

- 1) Find Min: $\Theta(1)$ [Same as both Binary and Binomial]
- 2) Delete Min: $\mathbf{O(\text{Log } n)}$ [$\Theta(\text{Log } n)$ in both Binary and Binomial]
- 3) Insert: $\Theta(1)$ [$\Theta(\text{Log } n)$ in Binary and $\Theta(1)$ in Binomial]
- 4) Decrease-Key: $\Theta(1)$ [$\Theta(\text{Log } n)$ in both Binary and Binomial]
- 5) Merge: $\Theta(1)$ [$\Theta(m \text{ Log } n)$ or $\Theta(m+n)$ in Binary and $\Theta(\text{Log } n)$ in Binomial]

Like **Binomial Heap**, Fibonacci Heap is a collection of trees with min-heap or max-heap property. In Fibonacci Heap, trees can have any shape even all trees can be single nodes (This is unlike Binomial Heap where every tree has to be Binomial Tree).

Below is an example Fibonacci Heap taken from [here](#).



Fibonacci Heap maintains a pointer to minimum value (which is root of a tree). All tree roots are connected using circular doubly linked list, so all of them can be accessed using single 'min' pointer.

The main idea is to execute operations in "lazy" way. For example merge operation simply links two heaps, insert operation simply adds a new tree with single node. The operation extract minimum is the most complicated operation. It does delayed work of consolidating trees. This makes delete also complicated as delete first decreases key to minus infinite, then calls extract minimum.

Below are some interesting facts about Fibonacci Heap

1. The reduced time complexity of Decrease-Key has importance in Dijkstra and Prim algorithms. With Binary Heap, time complexity of these algorithms is $O(V \text{Log } V + E \text{Log } V)$. If Fibonacci Heap is used, then time complexity is improved to $O(V \text{Log } V + E)$
2. Although Fibonacci Heap looks promising time complexity wise, it has been found slow in practice as hidden constants are high (Source [Wiki](#)).
3. Fibonacci heap are mainly called so because Fibonacci numbers are used in the running time analysis. Also, every node in Fibonacci Heap has degree at most $O(\log n)$ and the size of a subtree rooted in a node of degree k is at least F_{k+2} , where F_k is the k th Fibonacci number.

We will soon be discussing Fibonacci Heap operations in detail.

This article is contributed by **Shivam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Heap
Fibonacci

Heap Sort

Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.

What is Binary Heap?

Let us first define a Complete Binary Tree. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible (Source [Wikipedia](#))

A **Binary Heap** is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min heap. The heap can be represented by binary tree or array.

Why array based representation for Binary Heap?

Since a Binary Heap is a Complete Binary Tree, it can be easily represented as array and array based representation is space efficient. If the parent node is stored at index i , the left child can be calculated by $2 * i + 1$ and right child by $2 * i + 2$ (assuming the indexing starts at 0).

Heap Sort Algorithm for sorting in increasing order:

1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
3. Repeat above steps while size of heap is greater than 1.

How to build the heap?

Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom up order.

Lets understand with the help of an example:

```
Input data: 4, 10, 3, 5, 1
          4(0)
        /  \
      10(1) 3(2)
     /  \
    5(3) 1(4)
```

The numbers in bracket represent the indices in the array representation of data.

Applying heapify procedure to index 1:

```
          4(0)
        /  \
      10(1) 3(2)
     /  \
    5(3) 1(4)
```

Applying heapify procedure to index 0:

```
          10(0)
         /  \
       5(1) 3(2)
      /  \
     4(3) 1(4)
```

The heapify procedure calls itself recursively to build heap in top down manner.

C++

```
// C++ program for implementation of Heap Sort
#include <iostream>
```

```

using namespace std;

// To heapify a subtree rooted with node i which is
// an index in arr[], n is size of heap
void heapify(int arr[], int n, int i)
{
    int largest = i; // Initialize largest as root
    int l = 2*i + 1; // left = 2*i + 1
    int r = 2*i + 2; // right = 2*i + 2

    // If left child is larger than root
    if (l < n && arr[l] > arr[largest])
        largest = l;

    // If right child is larger than largest so far
    if (r < n && arr[r] > arr[largest])
        largest = r;

    // If largest is not root
    if (largest != i)
    {
        swap(arr[i], arr[largest]);

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}

// main function to do heap sort
void heapSort(int arr[], int n)
{
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // One by one extract an element from heap
    for (int i = n - 1; i >= 0; i--)
    {
        // Move current root to end
        swap(arr[0], arr[i]);

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}

/* A utility function to print array of size n */
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";
    cout << "\n";
}

// Driver program
int main()
{
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr)/sizeof(arr[0]);

    heapSort(arr, n);

    cout << "Sorted array is \n";
    printArray(arr, n);
}

```

Java

```

// Java program for implementation of Heap Sort
public class HeapSort
{
    public void sort(int arr[])
    {
        int n = arr.length;

        // Build heap (rearrange array)
        for (int i = n / 2 - 1; i >= 0; i--)
            heapify(arr, n, i);

        // One by one extract an element from heap
        for (int i = n - 1; i >= 0; i--)
        {
            // Move current root to end
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;

            // call max heapify on the reduced heap
            heapify(arr, i, 0);
        }
    }

    // To heapify a subtree rooted with node i which is
    // an index in arr[], n is size of heap
    void heapify(int arr[], int n, int i)
    {
        int largest = i; // Initialize largest as root
        int l = 2*i + 1; // left = 2*i + 1
        int r = 2*i + 2; // right = 2*i + 2

        // If left child is larger than root
        if (l < n && arr[l] > arr[largest])
            largest = l;

        // If right child is larger than largest so far
        if (r < n && arr[r] > arr[largest])
            largest = r;

        // If largest is not root
        if (largest != i)
        {
            int swap = arr[i];
            arr[i] = arr[largest];

```

```

        arr[largest] = swap;

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}

/* A utility function to print array of size n */
static void printArray(int arr[])
{
    int n = arr.length;
    for (int i=0; i<n; ++i)
        System.out.print(arr[i]+" ");
    System.out.println();
}

// Driver program
public static void main(String args[])
{
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = arr.length;

    HeapSort ob = new HeapSort();
    ob.sort(arr);

    System.out.println("Sorted array is");
    printArray(arr);
}
}

```

Python

```

# Python program for implementation of heap Sort

# To heapify subtree rooted at index i.
# n is size of heap
def heapify(arr, n, i):
    largest = i # Initialize largest as root
    l = 2 * i + 1    # left = 2*i + 1
    r = 2 * i + 2    # right = 2*i + 2

    # See if left child of root exists and is
    # greater than root
    if l < n and arr[l] > arr[i]:
        largest = l

    # See if right child of root exists and is
    # greater than root
    if r < n and arr[r] > arr[i]:
        largest = r

    # Change root, if needed
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i] # swap

    # Heapify the root.
    heapify(arr, n, largest)

# The main function to sort an array of given size
def heapSort(arr):
    n = len(arr)

    # Build a maxheap.
    for i in range(n-1, -1, -1):
        heapify(arr, n, i)

    # One by one extract elements
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i] # swap
        heapify(arr, i, 0)

# Driver code to test above
arr = [12, 11, 13, 5, 6, 7]
heapSort(arr)
n = len(arr)
print ("Sorted array is")
for i in range(n):
    print ("%d" %arr[i]),
# This code is contributed by Mohit Kumra

```

Output:

```

Sorted array is
5 6 7 11 12 13

```

[Here](#) is previous C code for reference.

Notes:

Heap sort is an in-place algorithm.

Its typical implementation is not stable, but can be made stable (See [this](#))

Time Complexity: Time complexity of heapify is $O(\log n)$. Time complexity of createAndBuildHeap() is $O(n)$ and overall time complexity of Heap Sort is $O(n \log n)$.

Applications of HeapSort

1. Sort a nearly sorted (or K sorted) array
2. k largest(or smallest) elements in an array

Heap sort algorithm has limited uses because Quicksort and Mergesort are better in practice. Nevertheless, the Heap data structure itself is enormously used. See [Applications of Heap Data Structure](#)

Snapshots:

scene00505



scene00793



scene01081



scene01297



scene01513



scene02449



Quiz on Heap Sort

Other Sorting Algorithms on GeeksforGeeks/GeeksQuiz:

QuickSort, Selection Sort, Bubble Sort, Insertion Sort, Merge Sort, Heap Sort, QuickSort, Radix Sort, Counting Sort, Bucket Sort, ShellSort, Comb Sort, Pigeonhole Sort

Coding practice for sorting.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: Searching and Sorting

k largest(or smallest) elements in an array | added Min Heap method

Question: Write an efficient program for printing k largest elements in an array. Elements in array can be in any order.

For example, if given array is [1, 23, 12, 9, 30, 2, 50] and you are asked for the largest 3 elements i.e., k = 3 then your program should print 50, 30 and 23.

Method 1 (Use Bubble k times)

Thanks to Shailendra for suggesting this approach.

- 1) Modify **Bubble Sort** to run the outer loop at most k times.
- 2) Print the last k elements of the array obtained in step 1.

Time Complexity: $O(nk)$

Like Bubble sort, other sorting algorithms like **Selection Sort** can also be modified to get the k largest elements.

Method 2 (Use temporary array)

K largest elements from $arr[0..n-1]$

- 1) Store the first k elements in a temporary array $temp[0..k-1]$.
- 2) Find the smallest element in $temp[]$, let the smallest element be *min*.
- 3) For each element *x* in $arr[k]$ to $arr[n-1]$
If *x* is greater than the min then remove *min* from $temp[]$ and insert *x*.
- 4) Print final k elements of $temp[]$

Time Complexity: $O((n-k)*k)$. If we want the output sorted then $O((n-k)*k + k\log k)$

Thanks to nesamani1822 for suggesting this method.

Method 3(Use Sorting)

- 1) Sort the elements in descending order in $O(n\log n)$
- 2) Print the first k numbers of the sorted array $O(k)$.

Time complexity: $O(n\log n)$

Method 4 (Use Max Heap)

- 1) Build a Max Heap tree in $O(n)$
- 2) Use **Extract Max** k times to get k maximum elements from the Max Heap $O(k\log n)$

Time complexity: $O(n + k\log n)$

Method 5(Use Oder Statistics)

- 1) Use order statistic algorithm to find the kth largest element. Please [see the topic selection in worst-case linear time](#) $O(n)$
- 2) Use **QuickSort** Partition algorithm to partition around the kth largest number $O(n)$.
- 3) Sort the k-1 elements (elements greater than the kth largest element) $O(k\log k)$. This step is needed only if sorted output is required.

Time complexity: $O(n)$ if we don't need the sorted output, otherwise $O(n+k\log k)$

Thanks to [Shilpi](#) for suggesting the first two approaches.

Method 6 (Use Min Heap)

This method is mainly an optimization of method 1. Instead of using $temp[]$ array, use Min Heap.

Thanks to [geek4u](#) for suggesting this method.

- 1) Build a Min Heap MH of the first k elements ($arr[0]$ to $arr[k-1]$) of the given array. $O(k)$

- 2) For each element, after the kth element ($arr[k]$ to $arr[n-1]$), compare it with root of MH.

.....a) If the element is greater than the root then make it root and call **heapify** for MH

.....b) Else ignore it.

// The step 2 is $O((n-k)*\log k)$

- 3) Finally, MH has k largest elements and root of the MH is the kth largest element.

Time Complexity: $O(k + (n-k)\log k)$ without sorted output. If sorted output is needed then $O(k + (n-k)\log k + k\log k)$

All of the above methods can also be used to find the kth largest (or smallest) element.

Please write comments if you find any of the above explanations/algorithms incorrect, or find better ways to solve the same problem.

References:

http://en.wikipedia.org/wiki/Selection_algorithm

Asked by [geek4u](#)

GATE CS Corner Company Wise Coding Practice

Heap
Searching
array
Order-Statistics

Sort a nearly sorted (or K sorted) array

Given an array of n elements, where each element is at most k away from its target position, devise an algorithm that sorts in $O(n \log k)$ time.

For example, let us consider k is 2, an element at index 7 in the sorted array, can be at indexes 5, 6, 7, 8, 9 in the given array.

Source: [Nearly sorted algorithm](#)

We can use **Insertion Sort** to sort the elements efficiently. Following is the C code for standard Insertion Sort.

```
/* Function to sort an array using insertion sort */
void insertionSort(int A[], int size)
{
    int i, key, j;
    for (i = 1; i < size; i++)
    {
        key = A[i];
        j = i - 1;

        /* Move elements of A[0..i-1], that are greater than key, to one
           position ahead of their current position.
           This loop will run at most k times */
        while (j >= 0 && A[j] > key)
        {
            A[j+1] = A[j];
            j = j - 1;
        }
        A[j+1] = key;
    }
}
```

The inner loop will run at most k times. To move every element to its correct place, at most k elements need to be moved. So overall *complexity will be $O(nk)$*

We can sort such arrays **more efficiently with the help of Heap data structure**. Following is the detailed process that uses Heap.

- 1) Create a Min Heap of size k+1 with first k+1 elements. This will take $O(k)$ time (See [this](#) GFact)
- 2) One by one remove min element from heap, put it in result array, and add a new element to heap from remaining elements.

Removing an element and adding a new element to min heap will take $\log k$ time. So overall complexity will be $O(k) + O((n-k)*\log K)$

```

#include<iostream>
using namespace std;

// Prototype of a utility function to swap two integers
void swap(int *x, int *y);

// A class for Min Heap
class MinHeap
{
    int *harr; // pointer to array of elements in heap
    int heap_size; // size of min heap
public:
    // Constructor
    MinHeap(int a[], int size);

    // to heapify a subtree with root at given index
    void MinHeapify(int i);

    // to get index of left child of node at index i
    int left(int i) { return (2*i + 1); }

    // to get index of right child of node at index i
    int right(int i) { return (2*i + 2); }

    // to remove min (or root), add a new value x, and return old root
    int replaceMin(int x);

    // to extract the root which is the minimum element
    int extractMin();
};

// Given an array of size n, where every element is k away from its target
// position, sorts the array in O(nLogk) time.
int sortK(int arr[], int n, int k)
{
    // Create a Min Heap of first (k+1) elements from
    // input array
    int *harr = new int[k+1];
    for (int i = 0; i <= k && i < n; i++) // i < n condition is needed when k > n
        harr[i] = arr[i];
    MinHeap hp(harr, k+1);

    // i is index for remaining elements in arr[] and ti
    // is target index of for cuurent minimum element in
    // Min Heapm 'hp'.
    for(int i = k+1, ti = 0; ti < n; i++, ti++)
    {
        // If there are remaining elements, then place
        // root of heap at target index and add arr[i]
        // to Min Heap
        if (i < n)
            arr[ti] = hp.replaceMin(arr[i]);

        // Otherwise place root at its target index and
        // reduce heap size
        else
            arr[ti] = hp.extractMin();
    }
}

// FOLLOWING ARE IMPLEMENTATIONS OF STANDARD MIN HEAP METHODS FROM CORMEN BOOK
// Constructor: Builds a heap from a given array a[] of given size
MinHeap::MinHeap(int a[], int size)
{
    heap_size = size;
    harr = a; // store address of array
    int i = (heap_size - 1)/2;
    while (i >= 0)
    {
        MinHeapify(i);
        i--;
    }
}

// Method to remove minimum element (or root) from min heap
int MinHeap::extractMin()
{
    int root = harr[0];
    if (heap_size > 1)
    {
        harr[0] = harr[heap_size-1];
        heap_size--;
        MinHeapify(0);
    }
    return root;
}

// Method to change root with given value x, and return the old root
int MinHeap::replaceMin(int x)
{
    int root = harr[0];
    harr[0] = x;
    if (root < x)
        MinHeapify(0);
    return root;
}

// A recursive method to heapify a subtree with root at given index
// This method assumes that the subtrees are already heapified
void MinHeap::MinHeapify(int i)
{
    int l = left(i);
    int r = right(i);
    int smallest = i;
    if (l < heap_size && harr[l] < harr[i])
        smallest = l;
    if (r < heap_size && harr[r] < harr[smallest])
        smallest = r;
    if (smallest != i)
    {
        swap(&harr[i], &harr[smallest]);
        MinHeapify(smallest);
    }
}

```

```
// A utility function to swap two elements
void swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

// A utility function to print array elements
void printArray(int arr[], int size)
{
    for (int i=0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver program to test above functions
int main()
{
    int k = 3;
    int arr[] = {2, 6, 3, 12, 56, 8};
    int n = sizeof(arr)/sizeof(arr[0]);
    sortK(arr, n, k);

    cout << "Following is sorted array\n";
    printArray (arr, n);

    return 0;
}
```

Output:

```
Following is sorted array
2 3 6 8 12 56
```

The Min Heap based method takes $O(n \log k)$ time and uses $O(k)$ auxiliary space.

We can also use a **Balanced Binary Search Tree** instead of Heap to store $K+1$ elements. The *insert* and *delete* operations on Balanced BST also take $O(\log k)$ time. So Balanced BST based method will also take $O(n \log k)$ time, but the Heap based method seems to be more efficient as the minimum element will always be at root. Also, Heap doesn't need extra space for left and right pointers.

Please write comments if you find any of the above codes/algorithms incorrect, or find other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Heap
Sorting
Insertion Sort

Tournament Tree (Winner Tree) and Binary Heap

Given a team of N players. How many minimum games are required to find second best player?

We can use adversary arguments based on tournament tree (Binary Heap).

Tournament tree is a form of min (max) heap which is a complete binary tree. Every external node represents a player and internal node represents winner. In a tournament tree every internal node contains winner and every leaf node contains one player.

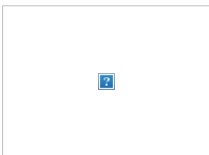
There will be $N - 1$ internal nodes in a binary tree with N leaf (external) nodes. For details see [this post](#) (put $n = 2$ in equation given in the post).

It is obvious that to select the best player among N players, $(N - 1)$ players to be eliminated, i.e. we need minimum of $(N - 1)$ games (comparisons). Mathematically we can prove it. In a binary tree $I = E - 1$, where I is number of internal nodes and E is number of external nodes. It means to find maximum or minimum element of an array, we need $N - 1$ (internal nodes) comparisons.

Second Best Player

The information explored during best player selection can be used to minimize the number of comparisons in tracing the next best players. For example, we can pick second best player in $(N + \log_2 N - 2)$ comparisons. For details read [this comment](#).

The following diagram displays a tournament tree (*winner tree*) as a max heap. Note that the concept of *loser tree* is different.



The above tree contains 4 leaf nodes that represent players and have 3 levels 0, 1 and 2. Initially 2 games are conducted at level 2, one between 5 and 3 and another one between 7 and 8. In the next move, one more game is conducted between 5 and 8 to conclude the final winner. Overall we need 3 comparisons. For second best player we need to trace the candidates participated with final winner, that leads to 7 as second best.

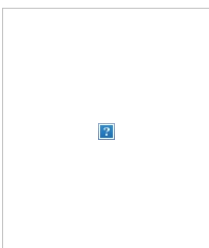
Median of Sorted Arrays

Tournament tree can effectively be used to find median of sorted arrays. Assume, given M sorted arrays of equal size L (for simplicity). We can attach all these sorted arrays to the tournament tree, one array per leaf. We need a tree of height **CEIL** ($\log_2 M$) to have atleast M external nodes.

Consider an example. Given 3 ($M = 3$) sorted integer arrays of maximum size 5 elements.

```
{ 2, 5, 7, 11, 15 } ---- Array1
{ 1, 3, 4 } ---- Array2
{ 6, 8, 12, 13, 14 } ---- Array3
```

What should be the height of tournament tree? We need to construct a tournament tree of height $\log_2 3 = 1.585 = 2$ rounded to next integer. A binary tree of height 2 will have 4 leaves to which we can attach the arrays as shown in the below figure.



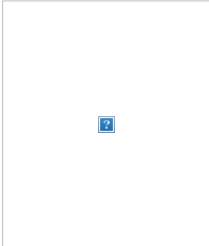
After the first tournament, the tree appears as below,



We can observe that the winner is from Array2. Hence the next element from Array2 will dive-in and games will be played along the winner path of previous tournament.

Note that infinity is used as sentinel element. Based on data being hold in nodes we can select the sentinel character. For example we usually store the pointers in nodes rather than keys, so NULL can serve as sentinel. If any of the array exhausts we will fill the corresponding leaf and upcoming internal nodes with sentinel.

After the second tournament, the tree appears as below,



The next winner is from Array1, so next element of Array1 array which is 5 will dive-in to the next round, and next tournament played along the path of 2.

The tournaments can be continued till we get median element which is $(5+3+5)/2 = 7$ th element. Note that there are even better algorithms for finding median of union of sorted arrays, for details see the related links given below.

In general with M sorted lists of size L_1, L_2, \dots, L_m requires time complexity of $O((L_1 + L_2 + \dots + L_m) * \log M)$ to merge all the arrays, and $O(m * \log M)$ time to find median, where m is median position.

Select smallest one million elements from one billion unsorted elements:

As a simple solution, we can sort the billion numbers and select first one million.

On a limited memory system sorting billion elements and picking the first one million seems to be impractical. We can use tournament tree approach. At any time only elements of tree to be in memory.

Split the large array (perhaps stored on disk) into smaller size arrays of size one million each (or even smaller that can be sorted by the machine). Sort these 1000 small size arrays and store them on disk as individual files. Construct a tournament tree which can have atleast 1000 leaf nodes (tree to be of height 10 since $2^9 < 1000 < 2^{10}$, if the individual file size is even smaller we will need more leaf nodes). Every leaf node will have an engine that picks next element from the sorted file stored on disk. We can play the tournament tree game to extract first one million elements.

Total cost = sorting 1000 lists of one million each + tree construction + tournaments

Implementation

We need to build the tree in bottom-up manner. All the leaf nodes filled first. Start at the left extreme of tree and fill along the breadth (i.e. from 2^{k-1} to $2^k - 1$ where k is depth of tree) and play the game. After practicing with few examples it will be easy to write code. Implementation is discussed in below code

[Second minimum element using minimum comparisons](#)

Related Posts

[Find the smallest and second smallest element in an array.](#)

[Second minimum element using minimum comparisons](#)

— by **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

[Heap](#)
[About Venki](#)
[Software Engineer](#)
[View all posts by Venki](#) →

Hashing | Set 1 (Introduction)

Suppose we want to design a system for storing employee records keyed using phone numbers. And we want following queries to be performed efficiently:

1. Insert a phone number and corresponding information.
2. Search a phone number and fetch the information.
3. Delete a phone number and related information.

We can think of using the following data structures to maintain information about different phone numbers.

1. Array of phone numbers and records.
2. Linked List of phone numbers and records.
3. Balanced binary search tree with phone numbers as keys.
4. Direct Access Table.

For **arrays and linked lists**, we need to search in a linear fashion, which can be costly in practice. If we use arrays and keep the data sorted, then a phone number can be searched in $O(\log n)$ time using Binary Search, but insert and delete operations become costly as we have to maintain sorted order.

With **balanced binary search tree**, we get moderate search, insert and delete times. All of these operations can be guaranteed to be in $O(\log n)$ time.

Another solution that one can think of is to use a **direct access table** where we make a big array and use phone numbers as index in the array. An entry in array is NIL if phone number is not present, else the array entry stores pointer to records corresponding to phone number. Time complexity wise this solution is the best among all, we can do all operations in $O(1)$ time. For example to insert a phone number, we create a record with details of given phone number, use phone number as index and store the pointer to the created record in table.

This solution has many practical limitations. First problem with this solution is extra space required is huge. For example if phone number is n digits, we need $O(m * 10^n)$ space for table where m is size of a pointer to record. Another problem is an integer in a programming language may not store n digits.

Due to above limitations Direct Access Table cannot always be used. **Hashing** is the solution that can be used in almost all such situations and performs extremely well compared to above data structures like Array, Linked List, Balanced BST in practice. With hashing we get $O(1)$ search time on average (under reasonable assumptions) and $O(n)$ in worst case.

Hashing is an improvement over Direct Access Table. The idea is to use hash function that converts a given phone number or any other key to a smaller number and uses the small number as index in a table called hash table.

Hash Function: A function that converts a given big phone number to a small practical integer value. The mapped integer value is used as an index in hash table. In simple terms, a hash function maps a big number or string to a small integer that can be used as index in hash table.

A good hash function should have following properties

- 1) Efficiently computable.
- 2) Should uniformly distribute the keys (Each table position equally likely for each key)

For example for phone numbers a bad hash function is to take first three digits. A better function is consider last three digits. Please note that this may not be the best hash function. There may be better ways.

Hash Table: An array that stores pointers to records corresponding to a given phone number. An entry in hash table is NIL if no existing phone number has hash function value equal to the index for the entry.

Collision Handling: Since a hash function gets us a small number for a big key, there is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some collision handling technique. Following are the ways to handle collisions:

- **Chaining:**The idea is to make each cell of hash table point to a linked list of records that have same hash function value. Chaining is simple, but requires additional memory outside the table.
- **Open Addressing:** In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we one by one examine table slots until the desired element is found or it is clear that the element is not in the table.

Next Posts:

[Separate Chaining for Collision Handling](#)

[Open Addressing for Collision Handling](#)

References:

[MIT Video Lecture](#)

[IITD Video Lecture](#)

"Introduction to Algorithms", Second Edition by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein,

<http://www.cs.princeton.edu/~rs/AlgsDS07/10Hashing.pdf>

<http://www.martinbroadhurst.com/articles/hash-table.html>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.

Category: Hash

Hashing | Set 2 (Separate Chaining)

We strongly recommend to refer below post as a prerequisite of this.

<http://quiz.geeksforgeeks.org/ hashing-set-1-introduction/>

What is Collision?

Since a hash function gets us a small number for a key which is a big integer or string, there is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some collision handling technique.

What are the chances of collisions with large table?

Collisions are very likely even if we have big table to store keys. An important observation is [Birthday Paradox](#). With only 23 persons, the probability that two people have same birthday is 50%.

How to handle Collisions?

There are mainly two methods to handle collision:

1) Separate Chaining

2) Open Addressing

In this article, only separate chaining is discussed. We will be discussing Open addressing in next post.

Separate Chaining:

The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

Let us consider a simple hash function as "**key mod 7**" and sequence of keys as 50, 700, 76, 85, 92, 73, 101.

[hashChaining](#)



Advantages:

- 1) Simple to implement.
- 2) Hash table never fills up, we can always add more elements to chain.
- 3) Less sensitive to the hash function or load factors.
- 4) It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages:

- 1) Cache performance of chaining is not good as keys are stored using linked list. Open addressing provides better cache performance as everything is stored in same table.
- 2) Wastage of Space (Some Parts of hash table are never used)
- 3) If the chain becomes long, then search time can become $O(n)$ in worst case.
- 4) Uses extra space for links.

Performance of Chaining:

Performance of hashing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of table (simple uniform hashing).

m = Number of slots in hash table
 n = Number of keys to be inserted in has table

Load factor $\alpha = n/m$

Expected time to search = $O(1 + \alpha)$

Expected time to insert/delete = $O(1 + \alpha)$

Time complexity of search insert and delete is
 $O(1)$ if α is $O(1)$

Next Post:

[Open Addressing for Collision Handling](#)

References:

http://courses.csail.mit.edu/6.006/fall09/lecture_notes/lecture05.pdf

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: Hash

Hashing | Set 3 (Open Addressing)

We strongly recommend to refer below post as a prerequisite of this.

[Hashing | Set 1 \(Introduction\)](#)

[Hashing | Set 2 \(Separate Chaining\)](#)

Open Addressing

Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself. So at any point, size of table must be greater than or equal to total number of keys (Note that we can increase table size by copying old data if needed).

Insert(k): Keep probing until an empty slot is found. Once an empty slot is found, insert k.

Search(k): Keep probing until slot's key doesn't become equal to k or an empty slot is reached.

Delete(k): **Delete operation is interesting.** If we simply delete a key, then search may fail. So slots of deleted keys are marked specially as "deleted".

Insert can insert an item in a deleted slot, but search doesn't stop at a deleted slot.

Open Addressing is done following ways:

a) Linear Probing: In linear probing, we linearly probe for next slot. For example, typical gap between two probes is 1 as taken in below example also.

let **hash(x)** be the slot index computed using hash function and **S** be the table size

If slot $\text{hash}(x) \% S$ is full, then we try $(\text{hash}(x) + 1) \% S$
If $(\text{hash}(x) + 1) \% S$ is also full, then we try $(\text{hash}(x) + 2) \% S$
If $(\text{hash}(x) + 2) \% S$ is also full, then we try $(\text{hash}(x) + 3) \% S$
.....
.....

Let us consider a simple hash function as "key mod 7" and sequence of keys as 50, 700, 76, 85, 92, 73, 101.

openAddressing



Clustering: The main problem with linear probing is clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search an element.

b) Quadratic Probing We look for i^2 th slot in i 'th iteration.

```
let hash(x) be the slot index computed using hash function.
If slot hash(x) % S is full, then we try (hash(x) + 1*1) % S
If (hash(x) + 1*1) % S is also full, then we try (hash(x) + 2*2) % S
If (hash(x) + 2*2) % S is also full, then we try (hash(x) + 3*3) % S
.....
.....
```

c) Double Hashing We use another hash function $hash2(x)$ and look for $i*hash2(x)$ slot in i 'th iteration.

```
let hash(x) be the slot index computed using hash function.
If slot hash(x) % S is full, then we try (hash(x) + 1*hash2(x)) % S
If (hash(x) + 1*hash2(x)) % S is also full, then we try (hash(x) + 2*hash2(x)) % S
If (hash(x) + 2*hash2(x)) % S is also full, then we try (hash(x) + 3*hash2(x)) % S
.....
.....
```

See [this](#) for step by step diagrams.

Comparison of above three:

Linear probing has the best cache performance, but suffers from clustering. One more advantage of Linear probing is easy to compute.

Quadratic probing lies between the two in terms of cache performance and clustering.

Double hashing has poor cache performance but no clustering. Double hashing requires more computation time as two hash functions need to be computed.

Open Addressing vs. Separate Chaining

Advantages of Chaining:

- 1) Chaining is Simpler to implement.
- 2) In chaining, Hash table never fills up, we can always add more elements to chain. In open addressing, table may become full.
- 3) Chaining is Less sensitive to the hash function or load factors.
- 4) Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.
- 5) Open addressing requires extra care for to avoid clustering and load factor.

Advantages of Open Addressing

- 1) Cache performance of chaining is not good as keys are stored using linked list. Open addressing provides better cache performance as everything is stored in same table.
- 2) Wastage of Space (Some Parts of hash table in chaining are never used). In Open addressing, a slot can be used even if an input doesn't map to it.
- 3) Chaining uses extra space for links.

Performance of Open Addressing:

Like Chaining, performance of hashing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of table (simple uniform hashing)

m = Number of slots in hash table
 n = Number of keys to be inserted in has table

Load factor $\alpha = n/m$ (

References:

<http://courses.csail.mit.edu/6.006/fall11/lectures/lecture10.pdf>

https://www.cse.cuhk.edu.hk/irwin.king/_media/teaching/csc2100b/tu6.pdf

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: Hash

Print a Binary Tree in Vertical Order | Set 2 (Hashmap based Method)

Given a binary tree, print it vertically. The following example illustrates vertical order traversal.

```
      1
     /\ 
    2  3
   /\  /\ 
  4 5 6 7
   \ \ 
    8 9
```

The output of print this tree vertically will be:

```
4
2
1 5 6
3 8
7
9
```

print-binary-tree-in-vertical-order



We strongly recommend to minimize the browser and try this yourself first.

We have discussed a $O(n^2)$ solution in the [previous post](#). In this post, an efficient solution based on hash map is discussed. We need to check the Horizontal Distances from root for all nodes. If two nodes have the same Horizontal Distance (HD), then they are on same vertical line. The idea of HD is simple. HD for root is 0, a right edge (edge connecting to right subtree) is considered as +1 horizontal distance and a left edge is considered as -1 horizontal distance. For example, in the above tree, HD for Node 4 is at -2, HD for Node 2 is -1, HD for 5 and 6 is 0 and HD for node 7 is +2.

We can do preorder traversal of the given Binary Tree. While traversing the tree, we can recursively calculate HDs. We initially pass the horizontal distance as 0 for root. For left subtree, we pass the Horizontal Distance as Horizontal distance of root minus 1. For right subtree, we pass the Horizontal Distance as Horizontal Distance of root plus 1. For every HD value, we maintain a list of nodes in a hash map. Whenever we see a node in traversal, we go to the hash map entry and add the node to the hash map using HD as a key in map.

Following is C++ implementation of the above method. Thanks to Chirag for providing the below C++ implementation.

C++

```
// C++ program for printing vertical order of a given binary tree
#include <iostream>
#include <vector>
#include <map>
using namespace std;

// Structure for a binary tree node
struct Node
{
    int key;
    Node *left, *right;
};

// A utility function to create a new node
struct Node* newNode(int key)
{
    struct Node* node = new Node;
    node->key = key;
    node->left = node->right = NULL;
    return node;
}
```

```

}

// Utility function to store vertical order in map 'm'
// 'hd' is horizontal distance of current node from root.
// 'hd' is initially passed as 0
void getVerticalOrder(Node* root, int hd, map<int, vector<int>> &m)
{
    // Base case
    if (root == NULL)
        return;

    // Store current node in map 'm'
    m[hd].push_back(root->key);

    // Store nodes in left subtree
    getVerticalOrder(root->left, hd-1, m);

    // Store nodes in right subtree
    getVerticalOrder(root->right, hd+1, m);
}

// The main function to print vertical order of a binary tree
// with given root
void printVerticalOrder(Node* root)
{
    // Create a map and store vertical order in map using
    // function getVerticalOrder()
    map < int,vector<int> > m;
    int hd = 0;
    getVerticalOrder(root, hd,m);

    // Traverse the map and print nodes at every horizontal
    // distance (hd)
    map< int,vector<int> > :: iterator it;
    for (it=m.begin(); it!=m.end(); it++)
    {
        for (int i=0; i<it->second.size(); ++i)
            cout << it->second[i] << " ";
        cout << endl;
    }
}

// Driver program to test above functions
int main()
{
    Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->right->left->right = newNode(8);
    root->right->right->right = newNode(9);
    cout << "Vertical order traversal is \n";
    printVerticalOrder(root);
    return 0;
}

```

Python

```

# Python program for printing vertical order of a given
# binary tree

# A binary tree node
class Node:
    # Constructor to create a new node
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

# Utility function to store vertical order in map 'm'
# 'hd' is horizontal distance of current node from root
# 'hd' is initially passed as 0
def getVerticalOrder(root, hd, m):

    # Base Case
    if root is None:
        return

    # Store current node in map 'm'
    try:
        m[hd].append(root.key)
    except:
        m[hd] = [root.key]

    # Store nodes in left subtree
    getVerticalOrder(root.left, hd-1, m)

    # Store nodes in right subtree
    getVerticalOrder(root.right, hd+1, m)

# The main function to print vertical order of a binary
# tree ith given root
def printVerticalOrder(root):

    # Create a map and store vertical order in map using
    # function getVerticalOrder()
    m = dict()
    hd = 0
    getVerticalOrder(root, hd, m)

    # Traverse the map and print nodes at every horizontal
    # distance (hd)
    for index, value in enumerate(sorted(m)):
        for i in m[value]:
            print i,
        print

# Driver program to test above function

```

```

root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.left = Node(6)
root.right.right = Node(7)
root.right.left.right = Node(8)
root.right.right.right = Node(9)
print "Vertical order traversal is"
printVerticalOrder(root)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

```

Vertical order traversal is
4
2
1 5 6
3 8
7
9

```

Time Complexity of hashing based solution can be considered as $O(n)$ under the assumption that we have good hashing function that allows insertion and retrieval operations in $O(1)$ time. In the above C++ implementation, [map of STL](#) is used. map in STL is typically implemented using a Self-Balancing Binary Search Tree where all operations take $O(\log n)$ time. Therefore time complexity of above implementation is $O(n \log n)$.

Note that the above solution may print nodes in same vertical order as they appear in tree. For example, the above program prints 12 before 9. See [this](#) for a sample run.

```

      1
     / \
    2   3
   /\  /\
  4 5 6 7
   /\
  8 10 9
   \
   11
   \
   12

```

Refer below post for level order traversal based solution. The below post makes sure that nodes of a vertical line are printed in same order as they appear in tree.

[Print a Binary Tree in Vertical Order | Set 3 \(Using Level Order Traversal\)](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Hash
Trees

Find whether an array is subset of another array | Added Method 3

Given two arrays: arr1[0..m-1] and arr2[0..n-1]. Find whether arr2[] is a subset of arr1[] or not. Both the arrays are not in sorted order. It may be assumed that elements in both array are distinct.

Examples:

Input: arr1[] = {11, 1, 13, 21, 3, 7}, arr2[] = {11, 3, 7, 1}

Output: arr2[] is a subset of arr1[]

Input: arr1[] = {1, 2, 3, 4, 5, 6}, arr2[] = {1, 2, 4}

Output: arr2[] is a subset of arr1[]

Input: arr1[] = {10, 5, 2, 23, 19}, arr2[] = {19, 5, 3}

Output: arr2[] is not a subset of arr1[]

Method 1 (Simple)

Use two loops: The outer loop picks all the elements of arr2[] one by one. The inner loop linearly searches for the element picked by outer loop. If all elements are found then return 1, else return 0.

```

#include<stdio.h>

/* Return 1 if arr2[] is a subset of arr1[] */
bool isSubset(int arr1[], int arr2[], int m, int n)
{
    int i = 0;
    int j = 0;
    for (i=0; i<n; i++)
    {
        for (j = 0; j<m; j++)
        {
            if(arr2[i] == arr1[j])
                break;
        }

        /* If the above inner loop was not broken at all then
        arr2[i] is not present in arr1[] */
        if (j == m)
            return 0;
    }

    /* If we reach here then all elements of arr2[]
    are present in arr1[] */
    return 1;
}

int main()
{
    int arr1[] = {11, 1, 13, 21, 3, 7};
    int arr2[] = {11, 3, 7, 1};

    int m = sizeof(arr1)/sizeof(arr1[0]);
    int n = sizeof(arr2)/sizeof(arr2[0]);

    if(isSubset(arr1, arr2, m, n))
        printf("arr2[] is subset of arr1[] ");
    else
        printf("arr2[] is not a subset of arr1[] ");

    getchar();
    return 0;
}

```

Time Complexity: $O(m \cdot n)$

Method 2 (Use Sorting and Binary Search)

- 1) Sort `arr1[]` $O(m \log m)$
- 2) For each element of `arr2[]`, do binary search for it in sorted `arr1[]`.
 - a) If the element is not found then return 0.
- 3) If all elements are present then return 1.

```
#include<stdio.h>

/* Function prototypes */
void quickSort(int *arr, int si, int ei);
int binarySearch(int arr[], int low, int high, int x);

/* Return 1 if arr2[] is a subset of arr1[] */
bool isSubset(int arr1[], int arr2[], int m, int n)
{
    int i = 0;

    quickSort(arr1, 0, m-1);
    for (i=0; i<n; i++)
    {
        if (binarySearch(arr1, 0, m-1, arr2[i]) == -1)
            return 0;
    }

    /* If we reach here then all elements of arr2[]
    are present in arr1[] */
    return 1;
}

/* FOLLOWING FUNCTIONS ARE ONLY FOR SEARCHING AND SORTING PURPOSE */
/* Standard Binary Search function */
int binarySearch(int arr[], int low, int high, int x)
{
    if(high >= low)
    {
        int mid = (low + high)/2; /*low + (high - low)/2;*/

        /* Check if arr[mid] is the first occurrence of x.
        arr[mid] is first occurrence if x is one of the following
        is true:
        (i) mid == 0 and arr[mid] == x
        (ii) arr[mid-1] < x and arr[mid] == x
        */
        if(( mid == 0 || x > arr[mid-1]) && (arr[mid] == x))
            return mid;
        else if(x > arr[mid])
            return binarySearch(arr, (mid + 1), high, x);
        else
            return binarySearch(arr, low, (mid - 1), x);
    }
    return -1;
}

void exchange(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}

int partition(int A[], int si, int ei)
{
    int x = A[ei];
    int i = (si - 1);
    int j;

    for (j = si; j <= ei - 1; j++)
    {
        if(A[j] <= x)
        {
            i++;
            exchange(&A[i], &A[j]);
        }
    }
    exchange (&A[i + 1], &A[ei]);
    return (i + 1);
}

/* Implementation of Quick Sort
A[] --> Array to be sorted
si --> Starting index
ei --> Ending index
*/
void quickSort(int A[], int si, int ei)
{
    int pi; /* Partitioning index */
    if(si < ei)
    {
        pi = partition(A, si, ei);
        quickSort(A, si, pi - 1);
        quickSort(A, pi + 1, ei);
    }
}

/*Driver program to test above functions */
int main()
{
    int arr1[] = {11, 1, 13, 21, 3, 7};
    int arr2[] = {11, 3, 7, 1};

    int m = sizeof(arr1)/sizeof(arr1[0]);
    int n = sizeof(arr2)/sizeof(arr2[0]);

    if(isSubset(arr1, arr2, m, n))
        printf("arr2[] is subset of arr1[] ");
    else
        printf("arr2[] is not a subset of arr1[] ");

    getchar();
    return 0;
}
```



```
}
```

Time Complexity: $O(m \log m + n \log m)$. Please note that this will be the complexity if an $m \log m$ algorithm is used for sorting which is not the case in above code. In above code Quick Sort is used and worst case time complexity of Quick Sort is $O(m^2)$

Method 3 (Use Sorting and Merging)

- Sort both arrays: `arr1[]` and `arr2[]` $O(m \log m + n \log n)$
- Use Merge type of process to see if all elements of sorted `arr2[]` are present in sorted `arr1[]`.

Thanks to [Parthsarathi](#) for suggesting this method.

```
/* Return 1 if arr2[] is a subset of arr1[] */
bool isSubset(int arr1[], int arr2[], int m, int n)
{
    int i = 0, j = 0;

    if(m < n)
        return 0;

    quickSort(arr1, 0, m-1);
    quickSort(arr2, 0, n-1);
    while( i < n && j < m )
    {
        if( arr1[i] < arr2[j] )
            j++;
        else if( arr1[i] == arr2[j] )
        {
            j++;
            i++;
        }
        else if( arr1[i] > arr2[j] )
            return 0;
    }

    if( i < n )
        return 0;
    else
        return 1;
}
```

Time Complexity: $O(m \log m + n \log n)$ which is better than method 2. Please note that this will be the complexity if an $n \log n$ algorithm is used for sorting both arrays which is not the case in above code. In above code Quick Sort is used and worst case time complexity of Quick Sort is $O(n^2)$

Method 4 (Use Hashing)

- Create a Hash Table for all the elements of `arr1[]`.
- Traverse `arr2[]` and search for each element of `arr2[]` in the Hash Table. If element is not found then return 0.
- If all elements are found then return 1.

Note that method 1, method 2 and method 4 don't handle the cases when we have duplicates in `arr2[]`. For example, {1, 4, 4, 2} is not a subset of {1, 4, 2}, but these methods will print it as a subset.

Source: <http://geeksforgeeks.org/forum/topic/if-an-array-is-subset-of-another>

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Hash
Sorting

Union and Intersection of two Linked Lists

Given two Linked Lists, create union and intersection lists that contain union and intersection of the elements present in the given lists. Order of elements in output lists doesn't matter.

Example:

```
Input:
List1: 10->15->4->20
List2: 8->4->2->10
Output:
Intersection List: 4->10
Union List: 2->8->20->4->15->10
```

Method 1 (Simple)

Following are simple algorithms to get union and intersection lists respectively.

Intersection (list1, list2)

Initialize result list as NULL. Traverse list1 and look for its each element in list2, if the element is present in list2, then add the element to result.

Union (list1, list2):

Initialize result list as NULL. Traverse list1 and add all of its elements to the result.

Traverse list2. If an element of list2 is already present in result then do not insert it to result, otherwise insert.

This method assumes that there are no duplicates in the given lists.

Thanks to Shekhu for suggesting this method. Following are C and Java implementations of this method.

C/C++

```
// C/C++ program to find union and intersection of two unsorted
// linked lists
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* A utility function to insert a node at the beginning of
a linked list */
void push(struct node** head_ref, int new_data);

/* A utility function to check if given data is present in a list */
bool isPresent(struct node *head, int data);

/* Function to get union of two linked lists head1 and head2 */
struct node *getUnion(struct node *head1, struct node *head2)
{
}
```

```

struct node *result = NULL;
struct node *t1 = head1, *t2 = head2;

// Insert all elements of list1 to the result list
while (t1 != NULL)
{
    push(&result, t1->data);
    t1 = t1->next;
}

// Insert those elements of list2 which are not
// present in result list
while (t2 != NULL)
{
    if (!isPresent(result, t2->data))
        push(&result, t2->data);
    t2 = t2->next;
}

return result;
}

/* Function to get intersection of two linked lists
head1 and head2 */
struct node *getIntersection(struct node *head1,
                             struct node *head2)
{
    struct node *result = NULL;
    struct node *t1 = head1;

    // Traverse list1 and search each element of it in
    // list2. If the element is present in list 2, then
    // insert the element to result
    while (t1 != NULL)
    {
        if (isPresent(head2, t1->data))
            push (&result, t1->data);
        t1 = t1->next;
    }

    return result;
}

/* A utility function to insert a node at the beginning of a linked list*/
void push (struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* A utility function to print a linked list*/
void printList (struct node *node)
{
    while (node != NULL)
    {
        printf ("%d ", node->data);
        node = node->next;
    }
}

/* A utility function that returns true if data is
present in linked list else return false */
bool isPresent (struct node *head, int data)
{
    struct node *t = head;
    while (t != NULL)
    {
        if (t->data == data)
            return 1;
        t = t->next;
    }
    return 0;
}

/* Driver program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head1 = NULL;
    struct node* head2 = NULL;
    struct node* intersecn = NULL;
    struct node* unin = NULL;

    /*create a linked lits 10->15->5->20 */
    push (&head1, 20);
    push (&head1, 4);
    push (&head1, 15);
    push (&head1, 10);

    /*create a linked lits 8->4->2->10 */
    push (&head2, 10);
    push (&head2, 2);
    push (&head2, 4);
    push (&head2, 8);

    intersecn = getIntersection (head1, head2);
    unin = getUnion (head1, head2);

    printf ("n First list is \n");
    printList (head1);

    printf ("n Second list is \n");
    printList (head2);

    printf ("n Intersection list is \n");
    printList (intersecn);
}

```

```

printf ("\n Union list is \n");
printList (unin);

return 0;
}

```

Java

```

// Java program to find union and intersection of two unsorted
// linked lists
class LinkedList
{
    Node head; // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;
        Node(int d)
        {
            data = d;
            next = null;
        }
    }

    /* Function to get Union of 2 Linked Lists */
    void getUnion(Node head1, Node head2)
    {
        Node t1 = head1, t2 = head2;

        //insert all elements of list1 in the result
        while (t1 != null)
        {
            push(t1.data);
            t1 = t1.next;
        }

        // insert those elements of list2 that are not present
        while (t2 != null)
        {
            if (!isPresent(head, t2.data))
                push(t2.data);
            t2 = t2.next;
        }
    }

    void getIntersection(Node head1, Node head2)
    {
        Node result = null;
        Node t1 = head1;

        // Traverse list1 and search each element of it in list2.
        // If the element is present in list 2, then insert the
        // element to result
        while (t1 != null)
        {
            if (isPresent(head2, t1.data))
                push(t1.data);
            t1 = t1.next;
        }
    }

    /* Utility function to print list */
    void printList()
    {
        Node temp = head;
        while(temp != null)
        {
            System.out.print(temp.data+" ");
            temp = temp.next;
        }
        System.out.println();
    }

    /* Inserts a node at start of linked list */
    void push(int new_data)
    {
        /* 1 & 2: Allocate the Node &
           Put in the data*/
        Node new_node = new Node(new_data);

        /* 3. Make next of new Node as head */
        new_node.next = head;

        /* 4. Move the head to point to new Node */
        head = new_node;
    }

    /* A utility function that returns true if data is present
    in linked list else return false */
    boolean isPresent (Node head, int data)
    {
        Node t = head;
        while (t != null)
        {
            if (t.data == data)
                return true;
            t = t.next;
        }
        return false;
    }

    /* Driver program to test above functions */
    public static void main(String args[])
    {
        LinkedList llist1 = new LinkedList();
        LinkedList llist2 = new LinkedList();
        LinkedList unin = new LinkedList();
    }
}

```

```

LinkedList intersecn = new LinkedList();

/*create a linked lits 10->15->5->20 */
l1list1.push(20);
l1list1.push(4);
l1list1.push(15);
l1list1.push(10);

/*create a linked lits 8->4->2->10 */
l1list2.push(10);
l1list2.push(2);
l1list2.push(4);
l1list2.push(8);

intersecn.getIntersection(l1list1.head, l1list2.head);
unin.getUnion(l1list1.head, l1list2.head);

System.out.println("First List is");
l1list1.printList();

System.out.println("Second List is");
l1list2.printList();

System.out.println("Intersection List is");
intersecn.printList();

System.out.println("Union List is");
unin.printList();
}
} /* This code is contributed by Rajat Mishra */

```

Output:

```

First list is
10 15 4 20
Second list is
8 4 2 10
Intersection list is
4 10
Union list is
2 8 20 4 15 10

```

Time Complexity: $O(mn)$ for both union and intersection operations. Here m is the number of elements in first list and n is the number of elements in second list.

Method 2 (Use Merge Sort)

In this method, algorithms for Union and Intersection are very similar. First we sort the given lists, then we traverse the sorted lists to get union and intersection. Following are the steps to be followed to get union and intersection lists.

- 1) Sort the first Linked List using merge sort. This step takes $O(m \log m)$ time. Refer [this post](#) for details of this step.
- 2) Sort the second Linked List using merge sort. This step takes $O(n \log n)$ time. Refer [this post](#) for details of this step.
- 3) Linearly scan both sorted lists to get the union and intersection. This step takes $O(m + n)$ time. This step can be implemented using the same algorithm as sorted arrays algorithm discussed [here](#).

Time complexity of this method is $O(m \log m + n \log n)$ which is better than method 1's time complexity.

Method 3 (Use Hashing)

Union (list1, list2)

Initialize the result list as NULL and create an empty hash table. Traverse both lists one by one, for each element being visited, look the element in hash table. If the element is not present, then insert the element to result list. If the element is present, then ignore it.

Intersection (list1, list2)

Initialize the result list as NULL and create an empty hash table. Traverse list1. For each element being visited in list1, insert the element in hash table. Traverse list2, for each element being visited in list2, look the element in hash table. If the element is present, then insert the element to result list. If the element is not present, then ignore it.

Both of the above methods assume that there are no duplicates.

Time complexity of this method depends on the hashing technique used and the distribution of elements in input lists. In practical, this approach may turn out to be better than above 2 methods.

Source: <http://geeksforgeeks.org/forum/topic/union-intersection-of-unsorted-lists>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

[Linked Lists](#)

Given an array A[] and a number x, check for pair in A[] with sum as x

Write a C program that, given an array A[] of n numbers and another number x, determines whether or not there exist two elements in S whose sum is exactly x.

We strongly recommend that you click here and practice it, before moving on to the solution.

METHOD 1 (Use Sorting)

Algorithm:

```

hasArrayTwoCandidates (A[], ar_size, sum)
1) Sort the array in non-decreasing order.
2) Initialize two index variables to find the candidate
   elements in the sorted array.
   (a) Initialize first to the leftmost index: l = 0
   (b) Initialize second the rightmost index: r = ar_size-1
3) Loop while l
Time Complexity: Depends on what sorting algorithm we use. If we use Merge Sort or Heap Sort then  $O(n \log n)$  in worst case. If we use Quick Sort then  $O(n^2)$  in worst case.

```

Auxiliary Space : Again, depends on sorting algorithm. For example auxiliary space is $O(n)$ for merge sort and $O(1)$ for Heap Sort.

Example:

Let Array be {1, 4, 45, 6, 10, -8} and sum to find be 16

Sort the array

A = {-8, 1, 4, 6, 10, 45}

Initialize l = 0, r = 5

$A[l] + A[r] (-8 + 45) > 16 \Rightarrow$ decrement r . Now $r = 10$

$A[l] + A[r] (-8 + 10)$ increment l . Now $l = 1$

$A[l] + A[r] (1 + 10)$ increment l . Now $l = 2$

$A[l] + A[r] (4 + 10)$ increment l . Now $l = 3$

$A[l] + A[r] (6 + 10) == 16 \Rightarrow$ Found candidates (return 1)

Note: If there are more than one pair having the given sum then this algorithm reports only one. Can be easily extended for this though.

Implementation:

C

```
# include <stdio.h>
# define bool int

void quickSort(int *, int, int);

bool hasArrayTwoCandidates(int A[], int arr_size, int sum)
{
    int l, r;

    /* Sort the elements */
    quickSort(A, 0, arr_size-1);

    /* Now look for the two candidates in the sorted array */
    l = 0;
    r = arr_size-1;
    while (l < r)
    {
        if(A[l] + A[r] == sum)
            return 1;
        else if(A[l] + A[r] < sum)
            l++;
        else // A[l] + A[r] > sum
            r--;
    }
    return 0;
}

/* Driver program to test above function */
int main()
{
    int A[] = {1, 4, 45, 6, 10, -8};
    int n = 16;
    int arr_size = 6;

    if( hasArrayTwoCandidates(A, arr_size, n))
        printf("Array has two elements with sum 16");
    else
        printf("Array doesn't have two elements with sum 16 ");

    getchar();
    return 0;
}

/* FOLLOWING FUNCTIONS ARE ONLY FOR SORTING PURPOSE */
void exchange(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}

int partition(int A[], int si, int ei)
{
    int x = A[ei];
    int i = (si - 1);
    int j;

    for (j = si; j <= ei - 1; j++)
    {
        if(A[j] <= x)
        {
            i++;
            exchange(&A[i], &A[j]);
        }
    }
    exchange (&A[i + 1], &A[ei]);
    return (i + 1);
}

/* Implementation of Quick Sort
A[] --> Array to be sorted
si --> Starting index
ei --> Ending index
*/
void quickSort(int A[], int si, int ei)
{
    int pi; /* Partitioning index */
    if(si < ei)
    {
        pi = partition(A, si, ei);
        quickSort(A, si, pi - 1);
        quickSort(A, pi + 1, ei);
    }
}
```

Python

```
# Python program to check for the sum condition to be satisfied
def hasArrayTwoCandidates(A,arr_size,sum):

    # sort the array
    quickSort(A,0,arr_size-1)
    l = 0
    r = arr_size-1

    # traverse the array for the two elements
    while l<r:
        if (A[l] + A[r] == sum):
            return 1
        elif (A[l] + A[r] < sum):
            l += 1
        else:
            r -= 1
    return 0

# Implementation of Quick Sort
# A[] --> Array to be sorted
# si --> Starting index
# ei --> Ending index
def quickSort(A, si, ei):
    if si < ei:
        pi=partition(A,si,ei)
        quickSort(A,si,pi-1)
        quickSort(A,pi+1,ei)

# Utility function for partitioning the array(used in quick sort)
def partition(A, si, ei):
    x = A[ei]
    i = (si-1)
    for j in range(si,ei):
        if A[j] <= x:
            i += 1

    # This operation is used to swap two variables is python
    A[i], A[ei] = A[j], A[i]

    A[i+1], A[ei] = A[ei], A[i+1]

    return i+1

# Driver program to test the functions
A = [1,4,45,6,10,-8]
n = 16
if (hasArrayTwoCandidates(A, len(A), n)):
    print("Array has two elements with the given sum")
else:
    print("Array doesn't have two elements with the given sum")

## This code is contributed by __Devesh Agrawal__
```

Output:

```
Array has two elements with the given sum
```

METHOD 2 (Use Hash Map)

Thanks to Bindu for suggesting this method and thanks to Shekhu for providing code.

This method works in O(n) time if range of numbers is known.

Let sum be the given sum and A[] be the array in which we need to find pair.

- 1) Initialize Binary Hash Map M[] = {0, 0, ...}
- 2) Do following for each element A[i] in A[]
 - (a) If M[x - A[i]] is set then print the pair (A[i], x - A[i])
 - (b) Set M[A[i]]

Implementation:

C/C++

```
#include <stdio.h>
#define MAX 100000

void printPairs(int arr[], int arr_size, int sum)
{
    int i, temp;
    bool binMap[MAX] = {0}; /*initialize hash map as 0*/

    for (i = 0; i < arr_size; i++)
    {
        temp = sum - arr[i];
        if (temp >= 0 && binMap[temp] == 1)
            printf("Pair with given sum %d is (%d, %d) \n",
                sum, arr[i], temp);
    }
}
```

```

        binMap[arr[i]] = 1;
    }
}

/* Driver program to test above function */
int main()
{
    int A[] = {1, 4, 45, 6, 10, 8};
    int n = 16;
    int arr_size = sizeof(A)/sizeof(A[0]);

    printPairs(A, arr_size, n);

    getchar();
    return 0;
}

```

Java

```

// Java implementation using Hashing
import java.io.*;

class PairSum
{
    private static final int MAX = 100000; // Max size of Hashmap

    static void printpairs(int arr[],int sum)
    {
        // Declares and initializes the whole array as false
        boolean[] binmap = new boolean[MAX];

        for (int i=0; i<arr.length; ++i)
        {
            int temp = sum-arr[i];

            // checking for condition
            if (temp>=0 && binmap[temp])
            {
                System.out.println("Pair with given sum " +
                                   sum + " is (" + arr[i] +
                                   ", "+temp+")");
            }
            binmap[arr[i]] = true;
        }
    }

    // Main to test the above function
    public static void main (String[] args)
    {
        int A[] = {1, 4, 45, 6, 10, 8};
        int n = 16;
        printpairs(A, n);
    }
}

// This article is contributed by Aakash Hasija

```

Python

```

# Python program to find if there are two elements with given sum
CONST_MAX = 100000

# function to check for the given sum in the array
def printPairs(arr, arr_size, sum):

    # initialize hash map as 0
    binmap = [0]*CONST_MAX

    for i in range(0,arr_size):
        temp = sum-arr[i]
        if (temp>=0 and binmap[temp]==1):
            print "Pair with the given sum is", arr[i], "and", temp
            binmap[arr[i]]=1

# driver program to check the above function
A = [1,4,45,6,10,-8]
n = 16
printPairs(A, len(A), n)

# This code is contributed by __Devesh Agrawal__

```

Time Complexity: O(n)

Output:

```
Pair with given sum 16 is (10, 6)
```

Auxiliary Space: O(R) where R is range of integers.

If range of numbers include negative numbers then also it works. All we have to do for negative numbers is to make everything positive by adding the absolute value of smallest negative integer to all numbers.

Please write comments if you find any of the above codes/algorithms incorrect, or find other ways to solve the same problem.

GATE CS Corner

Company Wise Coding Practice

Arrays
Hash
Amazon-Question
CareWale-Question
Hashing

Check if a given array contains duplicate elements within k distance from each other

Given an unsorted array that may contain duplicates. Also given a number k which is smaller than size of array. Write a function that returns true if array contains duplicates within k distance.

Examples:

Input: k = 3, arr[] = {1, 2, 3, 4, 1, 2, 3, 4}
Output: false
All duplicates are more than k distance away.

Input: k = 3, arr[] = {1, 2, 3, 1, 4, 5}
Output: true
1 is repeated at distance 3.

Input: k = 3, arr[] = {1, 2, 3, 4, 5}
Output: false

Input: k = 3, arr[] = {1, 2, 3, 4, 4}
Output: true

We strongly recommend that you click here and practice it, before moving on to the solution.

A **Simple Solution** is to run two loops. The outer loop picks every element 'arr[i]' as a starting element, the inner loop compares all elements which are within k distance of 'arr[i]'. The time complexity of this solution is $O(kn)$.

We can solve this problem in **$O(n)$ time using Hashing**. The idea is to one by one add elements to hash. We also remove elements which are at more than k distance from current element. Following is detailed algorithm.

- 1) Create an empty hashtable.
 - 2) Traverse all elements from left to right. Let the current element be 'arr[i]'
-a) If current element 'arr[i]' is present in hashtable, then return true.
....b) Else add arr[i] to hash and remove arr[i-k] from hash if i is greater than or equal to k

```
/* Java program to Check if a given array contains duplicate
elements within k distance from each other */
import java.util.*;

class Main
{
    static boolean checkDuplicatesWithinK(int arr[], int k)
    {
        // Creates an empty hashset
        HashSet<Integer> set = new HashSet<>();

        // Traverse the input array
        for (int i=0; i<arr.length; i++)
        {
            // If already present in hash, then we found
            // a duplicate within k distance
            if (set.contains(arr[i]))
                return true;

            // Add this item to hashset
            set.add(arr[i]);
        }
    }
}
```



```

// Remove the k+1 distant item
if (i >= k)
    set.remove(arr[i-k]);
}
return false;
}

// Driver method to test above method
public static void main (String[] args)
{
    int arr[] = {10, 5, 3, 4, 3, 5, 6};
    if (checkDuplicatesWithinK(arr, 3))
        System.out.println("Yes");
    else
        System.out.println("No");
}
}

```

Output:

Yes

This article is contributed by **Anuj**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Hash
Hashing

Find Itinerary from a given list of tickets

Given a list of tickets, find itinerary in order using the given list.

Example:

```

Input:
"Chennai" -> "Banglore"
"Bombay" -> "Delhi"
"Goa" -> "Chennai"
"Delhi" -> "Goa"

Output:
Bombay->Delhi, Delhi->Goa, Goa->Chennai, Chennai->Banglore,

```

It may be assumed that the input list of tickets is not cyclic and there is one ticket from every city except final destination.

One Solution is to build a graph and do **Topological Sorting** of the graph. Time complexity of this solution is $O(n)$.

We can also use **hashing** to avoid building a graph. The idea is to first find the starting point. A starting point would never be on 'to' side of a ticket. Once we find the starting point, we can simply traverse the given map to print itinerary in order. Following are steps.

- 1) Create a HashMap of given pair of tickets. Let the created HashMap be 'dataset'. Every entry of 'dataset' is of the form "from->to" like "Chennai" -> "Banglore"
- 2) Find the starting point of itinerary.
 - a) Create a reverse HashMap. Let the reverse be 'reverseMap'. Entries of 'reverseMap' are of the form "to->from". Following is 'reverseMap' for above example.


```

"Banglore" -> "Chennai"
"Delhi" -> "Bombay"
"Chennai" -> "Goa"
"Goa" -> "Delhi"

```
 - b) Traverse 'dataset'. For every key of dataset, check if it is there in 'reverseMap'. If a key is not present, then we found the starting point. In the above example, "Bombay" is starting point.
- 3) Start from above found starting point and traverse the 'dataset' to print itinerary.

All of the above steps require $O(n)$ time so overall time complexity is $O(n)$.

Below is Java implementation of above idea.

Java

```

// Java program to print itinerary in order
import java.util.HashMap;
import java.util.Map;

public class printItinerary
{
    // Driver function
    public static void main(String[] args)
    {
        Map<String, String> dataSet = new HashMap<String, String>();
        dataSet.put("Chennai", "Banglore");
        dataSet.put("Bombay", "Delhi");
        dataSet.put("Goa", "Chennai");
        dataSet.put("Delhi", "Goa");

        printResult(dataSet);
    }

    // This function populates 'result' for given input 'dataset'
    private static void printResult(Map<String, String> dataSet)
    {
        // To store reverse of given map
        Map<String, String> reverseMap = new HashMap<String, String>();

        // To fill reverse map, iterate through the given map
        for (Map.Entry<String, String> entry: dataSet.entrySet())
            reverseMap.put(entry.getValue(), entry.getKey());

        // Find the starting point of itinerary
        String start = null;
        for (Map.Entry<String, String> entry: dataSet.entrySet())

```

```

    {
        if (!reverseMap.containsKey(entry.getKey()))
        {
            start = entry.getKey();
            break;
        }
    }

    // If we could not find a starting point, then something wrong
    // with input
    if (start == null)
    {
        System.out.println("Invalid Input");
        return;
    }

    // Once we have starting point, we simple need to go next, next
    // of next using given hash map
    String to = dataSet.get(start);
    while (to != null)
    {
        System.out.print(start + ">" + to + ", ");
        start = to;
        to = dataSet.get(to);
    }
}
}

```

C++

```

#include <iostream>
#include <map>
#include <string>
using namespace std;

void printItinerary(map<string, string> dataSet)
{
    // To store reverse of given map
    map<string, string> reverseMap;
    map<string, string>::iterator it;

    // To fill reverse map, iterate through the given map
    for (it = dataSet.begin(); it!=dataSet.end(); it++)
        reverseMap[it->second] = it->first;

    // Find the starting point of itinerary
    string start;

    for (it = dataSet.begin(); it!=dataSet.end(); it++)
    {
        if (reverseMap.find(it->first) == reverseMap.end())
        {
            start = it->first;
            break;
        }
    }

    // If we could not find a starting point, then something wrong with input
    if (start.empty())
    {
        cout << "Invalid Input" << endl;
        return;
    }

    // Once we have starting point, we simple need to go next,
    //next of next using given hash map
    it = dataSet.find(start);
    while (it != dataSet.end())
    {
        cout << it->first << ">" << it->second << endl;
        it = dataSet.find(it->second);
    }
}

int main()
{
    map<string, string> dataSet;
    dataSet["Chennai"] = "Banglore";
    dataSet["Bombay"] = "Delhi";
    dataSet["Goa"] = "Chennai";
    dataSet["Delhi"] = "Goa";

    printItinerary(dataSet);

    return 0;
}
// C++ implementation is contributed by Aditya Goel

```

Output:

```
Bombay->Delhi, Delhi->Goa, Goa->Chennai, Chennai->Banglore,
```

This article is compiled by **Rahul Jain**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Hash
Hashing
Topological Sorting

Find number of Employees Under every Employee

Given a dictionary that contains mapping of employee and his manager as a number of (employee, manager) pairs like below.

```

{ "A", "C" },
{ "B", "C" },
{ "C", "F" },
{ "D", "E" },

```

```
{ "E", "F" },  
{ "F", "F" }
```

In this example C is manager of A,
C is also manager of B, F is manager
of C and so on.

Write a function to get no of employees under each manager in the hierarchy not just their direct reports. It may be assumed that an employee directly reports to only one manager. In the above dictionary the root node/ceo is listed as reporting to himself.

Output should be a Dictionary that contains following.

```
A - 0  
B - 0  
C - 2  
D - 0  
E - 1  
F - 5
```

Source: Microsoft Interview

This question might be solved differently but i followed this and found interesting, so sharing:

1. Create a reverse map with Manager->DirectReportingEmployee combination. Off-course employee will be multiple so Value in Map is List of Strings.

```
"C" --> "A", "B",  
"E" --> "D"  
"F" --> "C", "E", "F"
```

2. Now use the given employee-manager map to iterate and at the same time use newly reverse map to find the count of employees under manager.

Let the map created in step 2 be 'mngrempMap'

Do following for every employee 'emp'.

a) If 'emp' is not present in 'mngrempMap'

Count under 'emp' is 0 [Nobody reports to 'emp']

b) If 'emp' is present in 'mngrempMap'

Use the list of direct reports from map 'mngrempMap' and recursively calculate number of total employees under 'emp'.

A trick in step 2.b is to use memorization(Dynamic programming) while finding number of employees under a manager so that we don't need to find number of employees again for any of the employees. In the below code populateResultUtil() is the recursive function that uses memoization to avoid re-computation of same results.

Below is Java implementation of above ideas

```
// Java program to find number of persons under every employee  
import java.util.ArrayList;  
import java.util.HashMap;  
import java.util.List;  
import java.util.Map;  
  
public class NumberEmployeeUnderManager  
{  
    // A hashmap to store result. It stores count of employees  
    // under every employee, the count may be 0 also  
    static Map<String,Integer> result =  
        new HashMap<String, Integer>();  
  
    // Driver function  
    public static void main(String[] args)  
    {  
        Map<String, String> dataSet = new HashMap<String, String>();  
        dataSet.put("A", "C");  
        dataSet.put("B", "C");  
        dataSet.put("C", "F");  
        dataSet.put("D", "E");  
        dataSet.put("E", "F");  
        dataSet.put("F", "F");  
  
        populateResult(dataSet);  
        System.out.println("result = " + result);  
    }  
  
    // This function populates 'result' for given input 'dataset'  
    private static void populateResult(Map<String, String> dataSet)  
    {  
        // To store reverse of original map, each key will have 0  
        // to multiple values  
        Map<String, List<String>> mngrempMap =  
            new HashMap<String, List<String>>();  
  
        // To fill mngrempMap, iterate through the given map  
        for (Map.Entry<String,String> entry: dataSet.entrySet())  
        {  
            String emp = entry.getKey();  
            String mgr = entry.getValue();  
            if (!emp.equals(mngr)) // excluding emp-emp entry  
            {  
                // Get the previous list of direct reports under  
                // current 'mgr' and add the current 'emp' to the list  
                List<String> directReportList = mngrempMap.get(mngr);  
  
                // If 'emp' is the first employee under 'mgr'  
                if (directReportList == null)  
                    directReportList = new ArrayList<String>();  
  
                directReportList.add(emp);  
  
                // Replace old value for 'mgr' with new  
                // directReportList  
                mngrempMap.put(mngr, directReportList);  
            }  
        }  
  
        // Now use manager-Emp map built above to populate result  
        // with use of populateResultUtil()  
  
        // note- we are iterating over original emp-manager map and  
        // will use mngremp map in helper to get the count  
        for (String mgr: dataSet.keySet())
```

```

        populateResultUtil(mngr, mngrEmpMap);
    }

    // This is a recursive function to fill count for 'mngr' using
    // mngrEmpMap. This function uses memoization to avoid re-
    // computations of subproblems.
    private static int populateResultUtil(String mngr,
        Map<String, List<String>> mngrEmpMap)
    {
        int count = 0;

        // means employee is not a manager of any other employee
        if (!mngrEmpMap.containsKey(mngr))
        {
            result.put(mngr, 0);
            return 0;
        }

        // this employee count has already been done by this
        // method, so avoid re-computation
        else if (result.containsKey(mngr))
            count = result.get(mngr);

        else
        {
            List<String> directReportEmpList = mngrEmpMap.get(mngr);
            count = directReportEmpList.size();
            for (String directReportEmp: directReportEmpList)
                count += populateResultUtil(directReportEmp, mngrEmpMap);

            result.put(mngr, count);
        }
        return count;
    }
}

```

Output:

```
result = {D=0, E=1, F=5, A=0, B=0, C=2}
```

This article is contributed by **Chandan Prakash**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Hash
Hashing

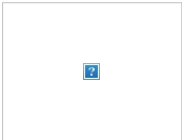
Graph and its representations

Graph is a data structure that consists of following two components:

1. A finite set of vertices also called as nodes.
2. A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not same as (v, u) in case of directed graph(di-graph). The pair of form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

Graphs are used to represent many real life applications: Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedin, facebook. For example, in facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender and locale. See [this](#) for more applications of graph.

Following is an example undirected graph with 5 vertices.



Following two are the most commonly used representations of graph.

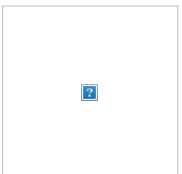
1. Adjacency Matrix
2. Adjacency List

There are other representations also like, Incidence Matrix and Incidence List. The choice of the graph representation is situation specific. It totally depends on the type of operations to be performed and ease of use.

Adjacency Matrix:

Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[i][j]. a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

The adjacency matrix for the above example graph is:



Adjacency Matrix
Representation of the above
graph

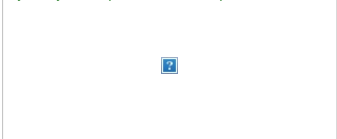
Pros: Representation is easier to implement and follow. Removing an edge takes O(1) time. Queries like whether there is an edge from vertex 'u' to vertex 'v' are efficient and can be done O(1).

Cons: Consumes more space O(V²). Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is O(V²) time.

Adjacency List:

An array of linked lists is used. Size of the array is equal to number of vertices. Let the array be array[]. An entry array[i] represents the linked list of vertices adjacent to the ith vertex. This representation can also be used to represent a weighted graph. The weights of edges can be stored in nodes of linked lists. Following is adjacency list representation of the above graph.

Adjacency List Representation of Graph



Adjacency List Representation of the above Graph

Below is C code for adjacency list representation of an undirected graph:

```
// A C Program to demonstrate adjacency list representation of graphs

#include <stdio.h>
#include <stdlib.h>

// A structure to represent an adjacency list node
struct AdjListNode
{
    int dest;
    struct AdjListNode* next;
};

// A structure to represent an adjacency list
struct AdjList
{
    struct AdjListNode *head; // pointer to head node of list
};

// A structure to represent a graph. A graph is an array of adjacency lists.
// Size of array will be V (number of vertices in graph)
struct Graph
{
    int V;
    struct AdjList* array;
};

// A utility function to create a new adjacency list node
struct AdjListNode* newAdjListNode(int dest)
{
    struct AdjListNode* newNode =
        (struct AdjListNode*) malloc(sizeof(struct AdjListNode));
    newNode->dest = dest;
    newNode->next = NULL;
    return newNode;
}

// A utility function that creates a graph of V vertices
struct Graph* createGraph(int V)
{
    struct Graph* graph = (struct Graph*) malloc(sizeof(struct Graph));
    graph->V = V;

    // Create an array of adjacency lists. Size of array will be V
    graph->array = (struct AdjList*) malloc(V * sizeof(struct AdjList));

    // Initialize each adjacency list as empty by making head as NULL
    int i;
    for (i = 0; i < V; ++i)
        graph->array[i].head = NULL;

    return graph;
}

// Adds an edge to an undirected graph
void addEdge(struct Graph* graph, int src, int dest)
{
    // Add an edge from src to dest. A new node is added to the adjacency
    // list of src. The node is added at the beginning
    struct AdjListNode* newNode = newAdjListNode(dest);
    newNode->next = graph->array[src].head;
    graph->array[src].head = newNode;

    // Since graph is undirected, add an edge from dest to src also
    newNode = newAdjListNode(src);
    newNode->next = graph->array[dest].head;
    graph->array[dest].head = newNode;
}

// A utility function to print the adjacency list representation of graph
void printGraph(struct Graph* graph)
{
    int v;
    for (v = 0; v < graph->V; ++v)
    {
        struct AdjListNode* pCrawl = graph->array[v].head;
        printf("\n Adjacency list of vertex %d\n head ", v);
        while (pCrawl)
        {
            printf("-> %d", pCrawl->dest);
            pCrawl = pCrawl->next;
        }
        printf("\n");
    }
}

// Driver program to test above functions
int main()
{
    // create the graph given in above figure
    int V = 5;
    struct Graph* graph = createGraph(V);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 4);
    addEdge(graph, 1, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 1, 4);
    addEdge(graph, 2, 3);
    addEdge(graph, 3, 4);

    // print the adjacency list representation of the above graph
    printGraph(graph);

    return 0;
}
```

Output:

Adjacency list of vertex 0
head -> 4-> 1

Adjacency list of vertex 1

```
head -> 4-> 3-> 2-> 0
```

```
Adjacency list of vertex 2  
head -> 3-> 1
```

```
Adjacency list of vertex 3  
head -> 4-> 2-> 1
```

```
Adjacency list of vertex 4  
head -> 3-> 1-> 0
```

Pros: Saves space $O(|V|+|E|)$. In the worst case, there can be $C(V, 2)$ number of edges in a graph thus consuming $O(V^2)$ space. Adding a vertex is easier.

Cons: Queries like whether there is an edge from vertex u to vertex v are not efficient and can be done $O(V)$.

Reference:

http://en.wikipedia.org/wiki/Graph_%28abstract_data_type%29

Related Post:

[Graph representation using STL for competitive programming | Set 1 \(DFS of Unweighted and Undirected\)](#)

[Graph implementation using STL for competitive programming | Set 2 \(Weighted graph\)](#)

This article is compiled by [Aashish Barnwal](#) and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Graph
Graph

Breadth First Traversal or BFS for a Graph

Breadth First Traversal (or Search) for a graph is similar to Breadth First Traversal of a tree (See method 2 of [this post](#)). The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Breadth First Traversal of the following graph is 2, 0, 3, 1.



Following are C++ and Java implementations of simple Breadth First Traversal from a given source.

The C++ implementation uses [adjacency list representation](#) of graphs. STL's [list container](#) is used to store lists of adjacent nodes and queue of nodes needed for BFS traversal.

C++

```
// Program to print BFS traversal from a given source vertex. BFS(int s)
// traverses vertices reachable from s.
#include<iostream>
#include <list>

using namespace std;

// This class represents a directed graph using adjacency list representation
class Graph
{
    int V; // No. of vertices
    list<int> *adj; // Pointer to an array containing adjacency lists
public:
    Graph(int V); // Constructor
    void addEdge(int v, int w); // function to add an edge to graph
    void BFS(int s); // prints BFS traversal from a given source s
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

void Graph::BFS(int s)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;
```

```

// Create a queue for BFS
list<int> queue;

// Mark the current node as visited and enqueue it
visited[s] = true;
queue.push_back(s);

// 'i' will be used to get all adjacent vertices of a vertex
list<int>::iterator i;

while(!queue.empty())
{
    // Dequeue a vertex from queue and print it
    s = queue.front();
    cout << s << " ";
    queue.pop_front();

    // Get all adjacent vertices of the dequeued vertex s
    // If a adjacent has not been visited, then mark it visited
    // and enqueue it
    for(i = adj[s].begin(); i != adj[s].end(); ++i)
    {
        if(!visited[*i])
        {
            visited[*i] = true;
            queue.push_back(*i);
        }
    }
}

// Driver program to test methods of graph class
int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Following is Breadth First Traversal "
    << "(starting from vertex 2) \n";
    g.BFS(2);

    return 0;
}

```

Java

```

// Java program to print BFS traversal from a given source vertex.
// BFS(int s) traverses vertices reachable from s.
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency list
// representation
class Graph
{
    private int V; // No. of vertices
    private LinkedList<Integer> adj[]; //Adjacency Lists

    // Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[V];
        for (int i=0; i<V; ++i)
            adj[i] = new LinkedList();
    }

    // Function to add an edge into the graph
    void addEdge(int v,int w)
    {
        adj[v].add(w);
    }

    // prints BFS traversal from a given source s
    void BFS(int s)
    {
        // Mark all the vertices as not visited (By default
        // set as false)
        boolean visited[] = new boolean[V];

        // Create a queue for BFS
        LinkedList<Integer> queue = new LinkedList<Integer>();

        // Mark the current node as visited and enqueue it
        visited[s]=true;
        queue.add(s);

        while (queue.size() != 0)
        {
            // Dequeue a vertex from queue and print it
            s = queue.poll();
            System.out.print(s+" ");

            // Get all adjacent vertices of the dequeued vertex s
            // If a adjacent has not been visited, then mark it
            // visited and enqueue it
            Iterator<Integer> i = adj[s].listIterator();
            while (i.hasNext())
            {
                int n = i.next();
                if (!visited[n])
                {
                    visited[n] = true;
                    queue.add(n);
                }
            }
        }
    }
}

```

```

}

// Driver method to
public static void main(String args[])
{
    Graph g = new Graph(4);

    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    System.out.println("Following is Breadth First Traversal "+
        "(starting from vertex 2)");

    g.BFS(2);
}
}
// This code is contributed by Aakash Hasija

```

Python

```

# Program to print BFS traversal from a given source
# vertex. BFS(int s) traverses vertices reachable
# from s.
from collections import defaultdict

# This class represents a directed graph using adjacency
# list representation
class Graph:

    # Constructor
    def __init__(self):

        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    # Function to print a BFS of graph
    def BFS(self, s):

        # Mark all the vertices as not visited
        visited = [False]*(len(self.graph))

        # Create a queue for BFS
        queue = []

        # Mark the source node as visited and enqueue it
        queue.append(s)
        visited[s] = True

        while queue:

            # Dequeue a vertex from queue and print it
            s = queue.pop(0)
            print s,

            # Get all adjacent vertices of the dequeued
            # vertex s. If a adjacent has not been visited,
            # then mark it visited and enqueue it
            for i in self.graph[s]:
                if visited[i] == False:
                    queue.append(i)
                    visited[i] = True

# Driver code
# Create a graph given in the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print "Following is Breadth First Traversal (starting from vertex 2)"
g.BFS(2)

# This code is contributed by Neelam Yadav

```

Output:

```

Following is Breadth First Traversal (starting from vertex 2)
2 0 3 1

```

Note that the above code traverses only the vertices reachable from a given source vertex. All the vertices may not be reachable from a given vertex (example Disconnected graph). To print all the vertices, we can modify the BFS function to do traversal starting from all nodes one by one (Like the [DFS modified version](#)) .

Time Complexity: $O(V+E)$ where V is number of vertices in the graph and E is number of edges in the graph.

You may like to see below also :

- [Depth First Traversal](#)
- [Applications of Breadth First Traversal](#)
- [Applications of Depth First Search](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

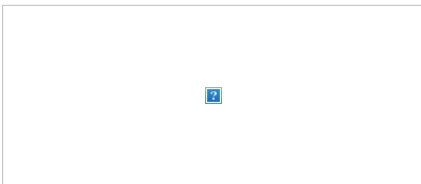
GATE CS Corner Company Wise Coding Practice

Graph
Queue
BFS
Stack-Queue

Depth First Traversal or DFS for a Graph

Depth First Traversal (or Search) for a graph is similar to [Depth First Traversal of a tree](#). The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array.

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Depth First Traversal of the following graph is 2, 0, 1, 3.



See [this post](#) for all applications of Depth First Traversal.

Following are implementations of simple Depth First Traversal. The C++ implementation uses [adjacency list representation](#) of graphs. STL's [list container](#) is used to store lists of adjacent nodes.

C++

```
// C++ program to print DFS traversal from a given vertex in a given graph
#include<iostream>
#include<list>

using namespace std;

// Graph class represents a directed graph using adjacency list representation
class Graph
{
    int V; // No. of vertices
    list<int>* adj; // Pointer to an array containing adjacency lists
    void DFSUtil(int v, bool visited[]); // A function used by DFS
public:
    Graph(int V); // Constructor
    void addEdge(int v, int w); // function to add an edge to graph
    void DFS(int v); // DFS traversal of the vertices reachable from v
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and print it
    visited[v] = true;
    cout << v << " ";

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}

// DFS traversal of the vertices reachable from v.
// It uses recursive DFSUtil()
void Graph::DFS(int v)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;
```

```

// Call the recursive helper function to print DFS traversal
DFSUtil(v, visited);
}

int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Following is Depth First Traversal (starting from vertex 2) \n";
    g.DFS(2);

    return 0;
}

```

Java

```

// Java program to print DFS traversal from a given given graph
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency list
// representation
class Graph
{
    private int V; // No. of vertices

    // Array of lists for Adjacency List Representation
    private LinkedList<Integer> adj[];

    // Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[V];
        for (int i=0; i<V; ++i)
            adj[i] = new LinkedList();
    }

    //Function to add an edge into the graph
    void addEdge(int v, int w)
    {
        adj[v].add(w); // Add w to v's list.
    }

    // A function used by DFS
    void DFSUtil(int v,boolean visited[])
    {
        // Mark the current node as visited and print it
        visited[v] = true;
        System.out.print(v+" ");

        // Recur for all the vertices adjacent to this vertex
        Iterator<Integer> i = adj[v].listIterator();
        while (i.hasNext())
        {
            int n = i.next();
            if (!visited[n])
                DFSUtil(n, visited);
        }
    }

    // The function to do DFS traversal. It uses recursive DFSUtil()
    void DFS(int v)
    {
        // Mark all the vertices as not visited(set as
        // false by default in java)
        boolean visited[] = new boolean[V];

        // Call the recursive helper function to print DFS traversal
        DFSUtil(v, visited);
    }

    public static void main(String args[])
    {
        Graph g = new Graph(4);

        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 3);

        System.out.println("Following is Depth First Traversal "+
            "(starting from vertex 2)");

        g.DFS(2);
    }
}
// This code is contributed by Aakash Hasija

```

Python

```

# Python program to print DFS traversal from a
# given given graph
from collections import defaultdict

# This class represents a directed graph using
# adjacency list representation
class Graph:

    # Constructor
    def __init__(self):

```

```

# default dictionary to store graph
self.graph = defaultdict(list)

# function to add an edge to graph
def addEdge(self,u,v):
    self.graph[u].append(v)

# A function used by DFS
def DFSUtil(self,v,visited):

    # Mark the current node as visited and print it
    visited[v]= True
    print v,

    # Recur for all the vertices adjacent to this vertex
    for i in self.graph[v]:
        if visited[i] == False:
            self.DFSUtil(i, visited)

# The function to do DFS traversal. It uses
# recursive DFSUtil()
def DFS(self,v):

    # Mark all the vertices as not visited
    visited = [False]*(len(self.graph))

    # Call the recursive helper function to print
    # DFS traversal
    self.DFSUtil(v,visited)

# Driver code
# Create a graph given in the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print "Following is DFS from (starting from vertex 2)"
g.DFS(2)

# This code is contributed by Neelam Yadav

```

Output:

```

Following is Depth First Traversal (starting from vertex 2)
2 0 1 3

```

Note that the above code traverses only the vertices reachable from a given source vertex. All the vertices may not be reachable from a given vertex (example Disconnected graph). To do complete DFS traversal of such graphs, we must call DFSUtil() for every vertex. Also, before calling DFSUtil(), we should check if it is already printed by some other call of DFSUtil(). Following implementation does the complete graph traversal even if the nodes are unreachable. The differences from the above code are highlighted in the below code.

C++

```

// C++ program to print DFS traversal for a given given graph
#include<iostream>
#include <list>
using namespace std;

class Graph
{
    int V; // No. of vertices
    list<int> *adj; // Pointer to an array containing adjacency lists
    void DFSUtil(int v, bool visited[]); // A function used by DFS
public:
    Graph(int V); // Constructor
    void addEdge(int v, int w); // function to add an edge to graph
    void DFS(); // prints DFS traversal of the complete graph
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and print it
    visited[v] = true;
    cout << v << " ";

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for(i = adj[v].begin(); i != adj[v].end(); ++i)
        if(!visited[*i])
            DFSUtil(*i, visited);
}

// The function to do DFS traversal. It uses recursive DFSUtil()
void Graph::DFS()
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to print DFS traversal
    // starting from all vertices one by one
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            DFSUtil(i, visited);
}

```

```

int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Following is Depth First Traversal\n";
    g.DFS();

    return 0;
}

```

Java

```

// Java program to print DFS traversal from a given graph
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency list
// representation
class Graph
{
    private int V; // No. of vertices

    // Array of lists for Adjacency List Representation
    private LinkedList<Integer> adj[];

    // Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[V];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    //Function to add an edge into the graph
    void addEdge(int v, int w)
    {
        adj[v].add(w); // Add w to v's list.
    }

    // A function used by DFS
    void DFSUtil(int v, boolean visited[])
    {
        // Mark the current node as visited and print it
        visited[v] = true;
        System.out.print(v+" ");

        // Recur for all the vertices adjacent to this vertex
        Iterator<Integer> i = adj[v].listIterator();
        while (i.hasNext())
        {
            int n = i.next();
            if (!visited[n])
                DFSUtil(n, visited);
        }
    }

    // The function to do DFS traversal. It uses recursive DFSUtil()
    void DFS()
    {
        // Mark all the vertices as not visited (set as
        // false by default in java)
        boolean visited[] = new boolean[V];

        // Call the recursive helper function to print DFS traversal
        // starting from all vertices one by one
        for (int i=0; i<V; ++i)
            if (!visited[i])
                DFSUtil(i, visited);
    }

    public static void main(String args[])
    {
        Graph g = new Graph(4);

        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 3);

        System.out.println("Following is Depth First Traversal");

        g.DFS();
    }
}
// This code is contributed by Aakash Hasija

```

Python

```

# Python program to print DFS traversal for complete graph
from collections import defaultdict

# This class represents a directed graph using adjacency
# list representation
class Graph:

    # Constructor
    def __init__(self):

        # default dictionary to store graph
        self.graph = defaultdict(list)

```

```

# function to add an edge to graph
def addEdge(self,u,v):
    self.graph[u].append(v)

# A function used by DFS
def DFSUtil(self, v, visited):

    # Mark the current node as visited and print it
    visited[v]= True
    print v,

    # Recur for all the vertices adjacent to
    # this vertex
    for i in self.graph[v]:
        if visited[i] == False:
            self.DFSUtil(i, visited)

# The function to do DFS traversal. It uses
# recursive DFSUtil()
def DFS(self):
    V = len(self.graph) #total vertices

    # Mark all the vertices as not visited
    visited =[False]*(V)

    # Call the recursive helper function to print
    # DFS traversal starting from all vertices one
    # by one
    for i in range(V):
        if visited[i] == False:
            self.DFSUtil(i, visited)

# Driver code
# Create a graph given in the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print "Following is Depth First Traversal"
g.DFS()

# This code is contributed by Neelam Yadav

```

Output:

```

Following is Depth First Traversal
0 1 2 3

```

Time Complexity: $O(V+E)$ where V is number of vertices in the graph and E is number of edges in the graph.

Breadth First Traversal for a Graph

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Graph
DFS

Applications of Depth First Search

Depth-first search (DFS) is an algorithm (or technique) for traversing a graph.

Following are the problems that use DFS as a building block.

1) For an unweighted graph, DFS traversal of the graph produces the minimum spanning tree and all pair shortest path tree.

2) Detecting cycle in a graph

A graph has cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edges. (See [this](#) for details)

3) Path Finding

We can specialize the DFS algorithm to find a path between two given vertices u and z.

i) Call DFS(G, u) with u as the start vertex.

ii) Use a stack S to keep track of the path between the start vertex and the current vertex.

iii) As soon as destination vertex z is encountered, return the path as the contents of the stack

See [this](#) for details.

4) Topological Sorting

Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs. In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in makefiles, data serialization, and resolving symbol dependencies in linkers [2].

5) To test if a graph is bipartite

We can augment either BFS or DFS when we first discover a new vertex, color it opposite its parents, and for each other edge, check it doesn't link two vertices of the same color. The first vertex in any connected component can be red or black! See [this](#) for details.

6) Finding Strongly Connected Components of a graph A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex. (See [this](#) for DFS based algo for finding Strongly Connected Components)

7) Solving puzzles with only one solution, such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)

Sources:

<http://www8.cs.umu.se/kurser/TDBAII/VT06/algorithms/LEC/LECTUR16/NODE16.HTM>

http://en.wikipedia.org/wiki/Depth-first_search

<http://www.personal.kent.edu/~muhamma/Algorithms/MyAlgorithms/GraphAlgor/depthSearch.htm>

<http://ww3.algorithmdesign.net/handouts/DFS.pdf>

GATE CS Corner Company Wise Coding Practice

Graph
DFS

Applications of Breadth First Traversal

We have earlier discussed [Breadth First Traversal Algorithm](#) for Graphs. We have also discussed [Applications of Depth First Traversal](#). In this article, applications of Breadth First Search are discussed.

1) Shortest Path and Minimum Spanning Tree for unweighted graph In unweighted graph, the shortest path is the path with least number of edges. With Breadth First, we always reach a vertex from given source using minimum number of edges. Also, in case of unweighted graphs, any spanning tree is Minimum Spanning Tree and we can use either Depth or Breadth first traversal for finding a spanning tree.

2) Peer to Peer Networks. In Peer to Peer Networks like BitTorrent, Breadth First Search is used to find all neighbor nodes.

3) Crawlers in Search Engines: Crawlers build index using Breadth First. The idea is to start from source page and follow all links from source and keep doing same. Depth First Traversal can also be used for crawlers, but the advantage with Breadth First Traversal is, depth or levels of built tree can be limited.

4) Social Networking Websites: In social networks, we can find people within a given distance 'k' from a person using Breadth First Search till 'k' levels.

5) GPS Navigation systems: Breadth First Search is used to find all neighboring locations.

6) Broadcasting in Network: In networks, a broadcasted packet follows Breadth First Search to reach all nodes.

7) In Garbage Collection: Breadth First Search is used in copying garbage collection using [Cheney's algorithm](#). Refer [this](#) and for details. Breadth First Search is preferred over Depth First Search because of better locality of reference:

8) Cycle detection in undirected graph: In undirected graphs, either Breadth First Search or Depth First Search can be used to detect cycle. In directed graph, only depth first search can be used.

9) Ford-Fulkerson algorithm In Ford-Fulkerson algorithm, we can either use Breadth First or Depth First Traversal to find the maximum flow. Breadth First Traversal is preferred as it reduces worst case time complexity to $O(VE^2)$.

10) To test if a graph is Bipartite We can either use Breadth First or Depth First Traversal.

11) Path Finding We can either use Breadth First or Depth First Traversal to find if there is a path between two vertices.

12) Finding all nodes within one connected component: We can either use Breadth First or Depth First Traversal to find all nodes reachable from a given node.

Many algorithms like [Prim's Minimum Spanning Tree](#) and [Dijkstra's Single Source Shortest Path](#) use structure similar to Breadth First Search.

There can be many more applications as Breadth First Search is one of the core algorithm for Graphs.

This article is contributed by **Neeraj Jain**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Graph
BFS

Detect Cycle in a Directed Graph

Given a directed graph, check whether the graph contains a cycle or not. Your function should return true if the given graph contains at least one cycle, else return false. For example, the following graph contains three cycles 0->2->0, 0->1->2->0 and 3->3, so your function must return true.

Depth First Traversal can be used to detect cycle in a Graph. DFS for a connected graph produces a tree. There is a cycle in a graph only if there is a [back edge](#) present in the graph. A back edge is an edge that is from a

node to itself (selfloop) or one of its ancestor in the tree produced by DFS. In the following graph, there are 3 back edges, marked with cross sign. We can observe that these 3 back edges indicate 3 cycles present in the graph.



For a disconnected graph, we get the DFS forest as output. To detect cycle, we can check for cycle in individual trees by checking back edges.

To detect a back edge, we can keep track of vertices currently in recursion stack of function for DFS traversal. If we reach a vertex that is already in the recursion stack, then there is a cycle in the tree. The edge that connects current vertex to the vertex in the recursion stack is back edge. We have used recStack[] array to keep track of vertices in the recursion stack.

```
// A C++ Program to detect cycle in a graph
#include<iostream>
#include <list>
#include <limits.h>

using namespace std;

class Graph
{
    int V; // No. of vertices
    list<int> *adj; // Pointer to an array containing adjacency lists
    bool isCyclicUtil(int v, bool visited[], bool *rs); // used by isCyclic()
public:
    Graph(int V); // Constructor
    void addEdge(int v, int w); // to add an edge to graph
    bool isCyclic(); // returns true if there is a cycle in this graph
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

// This function is a variation of DFSUtil() in http://www.geeksforgeeks.org/archives/18212
bool Graph::isCyclicUtil(int v, bool visited[], bool *recStack)
{
    if(visited[v] == false)
    {
        // Mark the current node as visited and part of recursion stack
        visited[v] = true;
        recStack[v] = true;

        // Recur for all the vertices adjacent to this vertex
        list<int>::iterator i;
        for(i = adj[v].begin(); i != adj[v].end(); ++i)
        {
            if ( !visited[*i] && isCyclicUtil(*i, visited, recStack) )
                return true;
            else if (recStack[*i])
                return true;
        }
    }
    recStack[v] = false; // remove the vertex from recursion stack
    return false;
}

// Returns true if the graph contains a cycle, else false.
// This function is a variation of DFS() in http://www.geeksforgeeks.org/archives/18212
bool Graph::isCyclic()
{
    // Mark all the vertices as not visited and not part of recursion
    // stack
    bool *visited = new bool[V];
    bool *recStack = new bool[V];
    for(int i = 0; i < V; i++)
    {
        visited[i] = false;
        recStack[i] = false;
    }

    // Call the recursive helper function to detect cycle in different
    // DFS trees
    for(int i = 0; i < V; i++)
        if (isCyclicUtil(i, visited, recStack))
            return true;

    return false;
}

int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    if(g.isCyclic())
        cout << "Graph contains cycle";
    else
        cout << "Graph doesn't contain cycle";
    return 0;
}
```

Output:

Graph contains cycle

Time Complexity of this method is same as time complexity of [DFS traversal](#) which is $O(V+E)$.

In the below article, another $O(V + E)$ method is discussed :

[Detect Cycle in a direct graph using colors](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Graph
DFS

Union-Find Algorithm | Set 1 (Detect Cycle in an Undirected Graph)

A [disjoint-set data structure](#) is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets. A [union-find algorithm](#) is an algorithm that performs two useful operations on such a data structure:

Find: Determine which subset a particular element is in. This can be used for determining if two elements are in the same subset.

Union: Join two subsets into a single subset.

In this post, we will discuss an application of Disjoint Set Data Structure. The application is to check whether a given graph contains a cycle or not.

Union-Find Algorithm can be used to check whether an undirected graph contains cycle or not. Note that we have discussed an [algorithm to detect cycle](#). This is another method based on *Union-Find*. This method assumes that graph doesn't contain any self-loops.

We can keep track of the subsets in a 1D array, let's call it parent[].

Let us consider the following graph:

cycle-in-graph



For each edge, make subsets using both the vertices of the edge. If both the vertices are in the same subset, a cycle is found.

Initially, all slots of parent array are initialized to -1 (means there is only one item in every subset).

```
0 1 2
-1 -1 -1
```

Now process all edges one by one.

Edge 0-1: Find the subsets in which vertices 0 and 1 are. Since they are in different subsets, we take the union of them. For taking the union, either make node 0 as parent of node 1 or vice-versa.

```
0 1 2
```

Edge 1-2: 1 is in subset 1 and 2 is in subset 2. So, take union.

```
0 1 2
```

Edge 0-2: 0 is in subset 2 and 2 is also in subset 2. Hence, including this edge forms a cycle.

How subset of 0 is same as 2?

0->1->2 // 1 is parent of 0 and 2 is parent of 1

Based on the above explanation, below are implementations:

C/C++

```
// A union-find algorithm to detect cycle in a graph
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// a structure to represent an edge in graph
struct Edge
{
    int src, dest;
};

// a structure to represent a graph
struct Graph
{
    // V-> Number of vertices, E-> Number of edges
```



```

int V, E;

// graph is represented as an array of edges
struct Edge* edge;
};

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    struct Graph* graph =
        (struct Graph*) malloc( sizeof(struct Graph) );
    graph->V = V;
    graph->E = E;

    graph->edge =
        (struct Edge*) malloc( graph->E * sizeof( struct Edge ) );

    return graph;
}

// A utility function to find the subset of an element i
int find(int parent[], int i)
{
    if (parent[i] == -1)
        return i;
    return find(parent, parent[i]);
}

// A utility function to do union of two subsets
void Union(int parent[], int x, int y)
{
    int xset = find(parent, x);
    int yset = find(parent, y);
    parent[xset] = yset;
}

// The main function to check whether a given graph contains
// cycle or not
int isCycle( struct Graph* graph )
{
    // Allocate memory for creating V subsets
    int *parent = (int*) malloc( graph->V * sizeof(int) );

    // Initialize all subsets as single element sets
    memset(parent, -1, sizeof(int) * graph->V);

    // Iterate through all edges of graph, find subset of both
    // vertices of every edge, if both subsets are same, then
    // there is cycle in graph.
    for(int i = 0; i < graph->E; ++i)
    {
        int x = find(parent, graph->edge[i].src);
        int y = find(parent, graph->edge[i].dest);

        if (x == y)
            return 1;

        Union(parent, x, y);
    }
    return 0;
}

// Driver program to test above functions
int main()
{
    /* Let us create following graph
    0
    | \
    1  \
    |  \
    1----2 */
    int V = 3, E = 3;
    struct Graph* graph = createGraph(V, E);

    // add edge 0-1
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;

    // add edge 1-2
    graph->edge[1].src = 1;
    graph->edge[1].dest = 2;

    // add edge 0-2
    graph->edge[2].src = 0;
    graph->edge[2].dest = 2;

    if (isCycle(graph))
        printf( "graph contains cycle" );
    else
        printf( "graph doesn't contain cycle" );

    return 0;
}

```

Java

```

// Java Program for union-find algorithm to detect cycle in a graph
import java.util.*;
import java.lang.*;
import java.io.*;

class Graph
{
    int V, E; // V-> no. of vertices & E->no of edges
    Edge edge[]; //collection of all edges

    class Edge
    {
        int src, dest;

```

```

};

// Creates a graph with V vertices and E edges
Graph(int v,int e)
{
    V = v;
    E = e;
    edge = new Edge[E];
    for (int i=0; i<e; ++i)
        edge[i] = new Edge();
}

// A utility function to find the subset of an element i
int find(int parent[], int i)
{
    if (parent[i] == -1)
        return i;
    return find(parent, parent[i]);
}

// A utility function to do union of two subsets
void Union(int parent[], int x, int y)
{
    int xset = find(parent, x);
    int yset = find(parent, y);
    parent[xset] = yset;
}

// The main function to check whether a given graph
// contains cycle or not
int isCycle( Graph graph)
{
    // Allocate memory for creating V subsets
    int parent[] = new int[graph.V];

    // Initialize all subsets as single element sets
    for (int i=0; i<graph.V; ++i)
        parent[i]=-1;

    // Iterate through all edges of graph, find subset of both
    // vertices of every edge, if both subsets are same, then
    // there is cycle in graph.
    for (int i = 0; i < graph.E; ++i)
    {
        int x = graph.find(parent, graph.edge[i].src);
        int y = graph.find(parent, graph.edge[i].dest);

        if (x == y)
            return 1;

        graph.Union(parent, x, y);
    }
    return 0;
}

// Driver Method
public static void main (String[] args)
{
    /* Let us create following graph
    0
    | \
    1 \
    1 \
    1-----2 */
    int V = 3, E = 3;
    Graph graph = new Graph(V, E);

    // add edge 0-1
    graph.edge[0].src = 0;
    graph.edge[0].dest = 1;

    // add edge 1-2
    graph.edge[1].src = 1;
    graph.edge[1].dest = 2;

    // add edge 0-2
    graph.edge[2].src = 0;
    graph.edge[2].dest = 2;

    if (graph.isCycle(graph)==1)
        System.out.println( "graph contains cycle" );
    else
        System.out.println( "graph doesn't contain cycle" );
}
}

```

Python

Python Program for union-find algorithm to detect cycle in a undirected graph
we have one edge for any two vertex i.e 1-2 is either 1-2 or 2-1 but not both

from collections import defaultdict

#This class represents a undirected graph using adjacency list representation
class Graph:

```

def __init__(self,vertices):
    self.V= vertices #No. of vertices
    self.graph = defaultdict(list) # default dictionary to store graph

```

```

# function to add an edge to graph
def addEdge(self,u,v):
    self.graph[u].append(v)

```

```

# A utility function to find the subset of an element i
def find_parent(self, parent,i):
    if parent[i] == -1:

```

```

    return i
    if parent[i] != -1:
        return self.find_parent(parent, parent[i])

# A utility function to do union of two subsets
def union(self, parent, x, y):
    x_set = self.find_parent(parent, x)
    y_set = self.find_parent(parent, y)
    parent[x_set] = y_set

# The main function to check whether a given graph
# contains cycle or not
def isCyclic(self):

# Allocate memory for creating V subsets and
# Initialize all subsets as single element sets
parent = [-1]*(self.V)

# Iterate through all edges of graph, find subset of both
# vertices of every edge, if both subsets are same, then
# there is cycle in graph.
for i in self.graph:
    for j in self.graph[i]:
        x = self.find_parent(parent, i)
        y = self.find_parent(parent, j)
        if x == y:
            return True
    self.union(parent, x, y)

# Create a graph given in the above diagram
g = Graph(3)
g.addEdge(0, 1)
g.addEdge(1, 2)
g.addEdge(2, 0)

if g.isCyclic():
    print "Graph contains cycle"
else :
    print "Graph does not contain cycle "

#This code is contributed by Neelam Yadav

```

Output:

```
graph contains cycle
```

Note that the implementation of *union()* and *find()* is naive and takes $O(n)$ time in worst case. These methods can be improved to $O(\text{Log}n)$ using *Union by Rank* or *Height*. We will soon be discussing *Union by Rank* in a separate post.

Related Articles :

[Union-Find Algorithm | Set 2 \(Union By Rank and Path Compression\)](#)

[Disjoint Set Data Structures \(Java Implementation\)](#)

[Greedy Algorithms | Set 2 \(Kruskal's Minimum Spanning Tree Algorithm\)](#)

[Job Sequencing Problem | Set 2 \(Using Disjoint Set\)](#)

This article is compiled by [Aashish Barnwal](#) and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Detect cycle in an undirected graph

Given an undirected graph, how to check if there is a cycle in the graph? For example, the following graph has a cycle 1-0-2-1.

cycleGraph



We have discussed [cycle detection for directed graph](#). We have also discussed a [union-find algorithm for cycle detection in undirected graphs](#). The time complexity of the union-find algorithm is $O(E \log V)$. Like directed graphs, we can use [DFS](#) to detect cycle in an undirected graph in $O(V+E)$ time. We do a DFS traversal of the given graph. For every visited vertex 'v', if there is an adjacent 'u' such that u is already visited and u is not parent of v, then there is a cycle in graph. If we don't find such an adjacent for any vertex, we say that there is no cycle. The assumption of this approach is that there are no parallel edges between any two vertices.

C++

```
// A C++ Program to detect cycle in an undirected graph
#include<iostream>
#include <list>
#include <limits.h>
using namespace std;

// Class for an undirected graph
class Graph
{
    int V; // No. of vertices
    list<int> *adj; // Pointer to an array containing adjacency lists
    bool isCyclicUtil(int v, bool visited[], int parent);
public:
    Graph(int V); // Constructor
    void addEdge(int v, int w); // to add an edge to graph
    bool isCyclic(); // returns true if there is a cycle
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
    adj[w].push_back(v); // Add v to w's list.
}

// A recursive function that uses visited[] and parent to detect
// cycle in subgraph reachable from vertex v.
bool Graph::isCyclicUtil(int v, bool visited[], int parent)
{
    // Mark the current node as visited
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
    {
        // If an adjacent is not visited, then recur for that adjacent
        if (!visited[*i])
        {
            if (isCyclicUtil(*i, visited, v))
                return true;
        }

        // If an adjacent is visited and not parent of current vertex,
        // then there is a cycle.
        else if (*i != parent)
            return true;
    }
    return false;
}

// Returns true if the graph contains a cycle, else false.
bool Graph::isCyclic()
{
    // Mark all the vertices as not visited and not part of recursion
    // stack
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to detect cycle in different
    // DFS trees
    for (int u = 0; u < V; u++)
        if (visited[u]) // Don't recur for u if it is already visited
            if (isCyclicUtil(u, visited, -1))
                return true;

    return false;
}

// Driver program to test above functions
int main()
{
    Graph g1(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(2, 0);
    g1.addEdge(0, 3);
    g1.addEdge(3, 4);
    g1.isCyclic()? cout << "Graph contains cycle\n":
        cout << "Graph doesn't contain cycle\n";
}
```

```

Graph g2(3);
g2.addEdge(0, 1);
g2.addEdge(1, 2);
g2.isCyclic()? cout << "Graph contains cycle\n":
    cout << "Graph doesn't contain cycle\n";

return 0;
}

```

Java

```

// A Java Program to detect cycle in an undirected graph
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency list
// representation
class Graph
{
    private int V; // No. of vertices
    private LinkedList<Integer> adj[]; // Adjacency List Representation

    // Constructor
    Graph(int v) {
        V = v;
        adj = new LinkedList[V];
        for(int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    // Function to add an edge into the graph
    void addEdge(int v,int w) {
        adj[v].add(w);
        adj[w].add(v);
    }

    // A recursive function that uses visited[] and parent to detect
    // cycle in subgraph reachable from vertex v.
    Boolean isCyclicUtil(int v, Boolean visited[], int parent)
    {
        // Mark the current node as visited
        visited[v] = true;
        Integer i;

        // Recur for all the vertices adjacent to this vertex
        Iterator<Integer> it = adj[v].iterator();
        while (it.hasNext())
        {
            i = it.next();

            // If an adjacent is not visited, then recur for that
            // adjacent
            if (!visited[i])
            {
                if (isCyclicUtil(i, visited, v))
                    return true;
            }

            // If an adjacent is visited and not parent of current
            // vertex, then there is a cycle.
            else if (i != parent)
                return true;
        }
        return false;
    }

    // Returns true if the graph contains a cycle, else false.
    Boolean isCyclic()
    {
        // Mark all the vertices as not visited and not part of
        // recursion stack
        Boolean visited[] = new Boolean[V];
        for (int i = 0; i < V; i++)
            visited[i] = false;

        // Call the recursive helper function to detect cycle in
        // different DFS trees
        for (int u = 0; u < V; u++)
            if (!visited[u]) // Don't recur for u if already visited
                if (isCyclicUtil(u, visited, -1))
                    return true;

        return false;
    }

    // Driver method to test above methods
    public static void main(String args[])
    {
        // Create a graph given in the above diagram
        Graph g1 = new Graph(5);
        g1.addEdge(1, 0);
        g1.addEdge(0, 2);
        g1.addEdge(2, 0);
        g1.addEdge(0, 3);
        g1.addEdge(3, 4);
        if (g1.isCyclic())
            System.out.println("Graph contains cycle");
        else
            System.out.println("Graph doesn't contains cycle");

        Graph g2 = new Graph(3);
        g2.addEdge(0, 1);
        g2.addEdge(1, 2);
        if (g2.isCyclic())
            System.out.println("Graph contains cycle");
        else
            System.out.println("Graph doesn't contains cycle");
    }
}
// This code is contributed by Aakash Hasija

```

Python

```
# Python Program to detect cycle in an undirected graph

from collections import defaultdict

#This class represents a undirected graph using adjacency list representation
class Graph:

    def __init__(self,vertices):
        self.V= vertices #No. of vertices
        self.graph = defaultdict(list) # default dictionary to store graph

    # function to add an edge to graph
    def addEdge(self,v,w):
        self.graph[v].append(w) #Add w to v_s list
        self.graph[w].append(v) #Add v to w_s list

    # A recursive function that uses visited[] and parent to detect
    # cycle in subgraph reachable from vertex v.
    def isCyclicUtil(self,v,visited,parent):

        #Mark the current node as visited
        visited[v]= True

        #Recur for all the vertices adjacent to this vertex
        for i in self.graph[v]:
            # If the node is not visited then recurse on it
            if visited[i]==False :
                if(self.isCyclicUtil(i,visited,v)):
                    return True
            # If an adjacent vertex is visited and not parent of current vertex,
            # then there is a cycle
            elif parent!=i:
                return True

        return False

    #Returns true if the graph contains a cycle, else false.
    def isCyclic(self):
        # Mark all the vertices as not visited
        visited =[False]*(self.V)
        # Call the recursive helper function to detect cycle in different
        # DFS trees
        for i in range(self.V):
            if visited[i] ==False: #Don't recur for u if it is already visited
                if(self.isCyclicUtil(i,visited,-1))== True:
                    return True

        return False

# Create a graph given in the above diagram
g = Graph(5)
g.addEdge(1, 0)
g.addEdge(0, 2)
g.addEdge(2, 0)
g.addEdge(0, 3)
g.addEdge(3, 4)

if g.isCyclic():
    print "Graph contains cycle"
else :
    print "Graph does not contain cycle "
g1 = Graph(3)
g1.addEdge(0,1)
g1.addEdge(1,2)

if g1.isCyclic():
    print "Graph contains cycle"
else :
    print "Graph does not contain cycle "

#This code is contributed by Neelam Yadav
```

Output:

```
Graph contains cycle
Graph doesn't contain cycle
```

Time Complexity: The program does a simple DFS Traversal of graph and graph is represented using adjacency list. So the time complexity is $O(V+E)$

Exercise: Can we use BFS to detect cycle in an undirected graph in $O(V+E)$ time? What about directed graphs?

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner **Company Wise Coding Practice**

Graph

Longest Path in a Directed Acyclic Graph

Given a **Weighted Directed Acyclic Graph (DAG)** and a source vertex s in it, find the longest distances from s to all other vertices in the given graph.

The longest path problem for a general graph is not as easy as the shortest path problem because the longest path problem doesn't have **optimal substructure property**. In fact, the **Longest Path problem is NP-Hard for a general graph**. However, the longest path problem has a linear time solution for directed acyclic graphs. The idea is similar to **linear time solution for shortest path in a directed acyclic graph**, we use **Topological Sorting**.

We initialize distances to all vertices as minus infinite and distance to source as 0, then we find a **topological sorting** of the graph. Topological Sorting of a graph represents a linear ordering of the graph (See below, figure (b) is a linear representation of figure (a)). Once we have topological order (or linear representation), we one by one process all vertices in topological order. For every vertex being processed, we update distances of its adjacent using distance of current vertex.

Following figure shows step by step process of finding longest paths.

LongestPath



Following is complete algorithm for finding longest distances.

1) Initialize $dist[] = [NINF, NINF, \dots]$ and $dist[s] = 0$ where s is the source vertex. Here NINF means negative infinite.

2) Create a topological order of all vertices.

3) Do following for every vertex u in topological order.

.....Do following for every adjacent vertex v of u

.....if ($dist[v] < dist[u] + weight(u, v)$)

..... $dist[v] = dist[u] + weight(u, v)$

Following is C++ implementation of the above algorithm.

```
// A C++ program to find single source longest distances in a DAG
#include <iostream>
#include <list>
#include <stack>
#include <limits.h>
#define NINF INT_MIN
using namespace std;

// Graph is represented using adjacency list. Every node of adjacency list
// contains vertex number of the vertex to which edge connects. It also
// contains weight of the edge
class AdjListNode
{
    int v;
    int weight;
public:
    AdjListNode(int _v, int _w) { v = _v; weight = _w; }
    int getV() { return v; }
    int getWeight() { return weight; }
};

// Class to represent a graph using adjacency list representation
class Graph
{
    int V; // No. of vertices

    // Pointer to an array containing adjacency lists
    list<AdjListNode> *adj;

    // A function used by longestPath
    void topologicalSortUtil(int v, bool visited[], stack<int> &Stack);
public:
    Graph(int V); // Constructor

    // function to add an edge to graph
    void addEdge(int u, int v, int weight);

    // Finds longest distances from given source vertex
    void longestPath(int s);
};

Graph::Graph(int V) // Constructor
{
    this->V = V;
    adj = new list<AdjListNode>[V];
}

void Graph::addEdge(int u, int v, int weight)
{
    AdjListNode node(v, weight);
    adj[u].push_back(node); // Add v to u's list
}

// A recursive function used by longestPath. See below link for details
// http://www.geeksforgeeks.org/topological-sorting/
```

```

void Graph::topologicalSortUtil(int v, bool visited[], stack<int> &Stack)
{
    // Mark the current node as visited
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<AdjListNode>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
    {
        AdjListNode node = *i;
        if (!visited[node.getV()])
            topologicalSortUtil(node.getV(), visited, Stack);
    }

    // Push current vertex to stack which stores topological sort
    Stack.push(v);
}

// The function to find longest distances from a given vertex. It uses
// recursive topologicalSortUtil() to get topological sorting.
void Graph::longestPath(int s)
{
    stack<int> Stack;
    int dist[V];

    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to store Topological Sort
    // starting from all vertices one by one
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            topologicalSortUtil(i, visited, Stack);

    // Initialize distances to all vertices as infinite and distance
    // to source as 0
    for (int i = 0; i < V; i++)
        dist[i] = NINF;
    dist[s] = 0;

    // Process vertices in topological order
    while (Stack.empty() == false)
    {
        // Get the next vertex from topological order
        int u = Stack.top();
        Stack.pop();

        // Update distances of all adjacent vertices
        list<AdjListNode>::iterator i;
        if (dist[u] != NINF)
        {
            for (i = adj[u].begin(); i != adj[u].end(); ++i)
                if (dist[*i->getV()] < dist[u] + i->getWeight())
                    dist[*i->getV()] = dist[u] + i->getWeight();
        }
    }

    // Print the calculated longest distances
    for (int i = 0; i < V; i++)
        (dist[i] == NINF) ? cout << "INF " : cout << dist[i] << " ";
}

// Driver program to test above functions
int main()
{
    // Create a graph given in the above diagram. Here vertex numbers are
    // 0, 1, 2, 3, 4, 5 with following mappings:
    // 0→r, 1→s, 2→t, 3→x, 4→y, 5→z
    Graph g(6);
    g.addEdge(0, 1, 5);
    g.addEdge(0, 2, 3);
    g.addEdge(1, 3, 6);
    g.addEdge(1, 2, 2);
    g.addEdge(2, 4, 4);
    g.addEdge(2, 5, 2);
    g.addEdge(2, 3, 7);
    g.addEdge(3, 5, 1);
    g.addEdge(3, 4, -1);
    g.addEdge(4, 5, -2);

    int s = 1;
    cout << "Following are longest distances from source vertex " << s << "\n";
    g.longestPath(s);

    return 0;
}

```

Output:

```

Following are longest distances from source vertex 1
INF 0 2 9 8 10

```

Time Complexity: Time complexity of topological sorting is $O(V+E)$. After finding topological order, the algorithm process all vertices and for every vertex, it runs a loop for all adjacent vertices. Total adjacent vertices in a graph is $O(E)$. So the inner loop runs $O(V+E)$ times. Therefore, overall time complexity of this algorithm is $O(V+E)$.

Exercise: The above solution print longest distances, extend the code to print paths also.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Graph
shortest path
Topological Sorting

Topological Sorting

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv , vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

For example, a topological sorting of the following graph is "5 4 2 3 1 0". There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is "4 5 2 3 1 0". The first

vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no in-coming edges).

graph



Topological Sorting vs Depth First Traversal (DFS):

In [DFS](#), we print a vertex and then recursively call DFS for its adjacent vertices. In topological sorting, we need to print a vertex before its adjacent vertices. For example, in the given graph, the vertex '5' should be printed before vertex '0', but unlike [DFS](#), the vertex '4' should also be printed before vertex '0'. So Topological sorting is different from DFS. For example, a DFS of the above graph is "5 2 3 1 0 4", but it is not a topological sorting

Algorithm to find Topological Sorting:

We recommend to first see implementation of DFS [here](#). We can modify [DFS](#) to find Topological Sorting of a graph. In [DFS](#), we start from a vertex, we first print it and then recursively call DFS for its adjacent vertices. In topological sorting, we use a temporary stack. We don't print the vertex immediately, we first recursively call topological sorting for all its adjacent vertices, then push it to a stack. Finally, print contents of stack. Note that a vertex is pushed to stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in stack.

Following are C++ and Java implementations of topological sorting. Please see the code for [Depth First Traversal for a disconnected Graph](#) and note the differences between the second code given there and the below code.

C++

```
// A C++ program to print topological sorting of a DAG
#include<iostream>
#include <list>
#include <stack>
using namespace std;

// Class to represent a graph
class Graph
{
    int V; // No. of vertices

    // Pointer to an array containing adjacency lists
    list<int> *adj;

    // A function used by topologicalSort
    void topologicalSortUtil(int v, bool visited[], stack<int> &Stack);
public:
    Graph(int V); // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w);

    // prints a Topological Sort of the complete graph
    void topologicalSort();
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

// A recursive function used by topologicalSort
void Graph::topologicalSortUtil(int v, bool visited[],
                                stack<int> &Stack)
{
    // Mark the current node as visited.
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            topologicalSortUtil(*i, visited, Stack);

    // Push current vertex to stack which stores result
    Stack.push(v);
}

// The function to do Topological Sort. It uses recursive
// topologicalSortUtil()
void Graph::topologicalSort()
{
    stack<int> Stack;

    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to store Topological
    // Sort starting from all vertices one by one
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            topologicalSortUtil(i, visited, Stack);

    // Print contents of stack
    while (Stack.empty() == false)
    {
        cout << Stack.top() << " ";
        Stack.pop();
    }
}

// Driver program to test above functions
int main()
{
    // Create a graph given in the above diagram
    Graph g(6);
```

```

g.addEdge(5, 2);
g.addEdge(5, 0);
g.addEdge(4, 0);
g.addEdge(4, 1);
g.addEdge(2, 3);
g.addEdge(3, 1);

cout << "Following is a Topological Sort of the given graph \n";
g.topologicalSort();

return 0;
}

```

Java

```

// A Java program to print topological sorting of a DAG
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency
// list representation
class Graph
{
    private int V; // No. of vertices
    private LinkedList<Integer> adj[]; // Adjacency List

    //Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[V];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    // Function to add an edge into the graph
    void addEdge(int v,int w) { adj[v].add(w); }

    // A recursive function used by topologicalSort
    void topologicalSortUtil(int v, boolean visited[],
        Stack stack)
    {
        // Mark the current node as visited.
        visited[v] = true;
        Integer i;

        // Recur for all the vertices adjacent to this
        // vertex
        Iterator<Integer> it = adj[v].iterator();
        while (it.hasNext())
        {
            i = it.next();
            if (!visited[i])
                topologicalSortUtil(i, visited, stack);
        }

        // Push current vertex to stack which stores result
        stack.push(new Integer(v));
    }

    // The function to do Topological Sort. It uses
    // recursive topologicalSortUtil()
    void topologicalSort()
    {
        Stack stack = new Stack();

        // Mark all the vertices as not visited
        boolean visited[] = new boolean[V];
        for (int i = 0; i < V; i++)
            visited[i] = false;

        // Call the recursive helper function to store
        // Topological Sort starting from all vertices
        // one by one
        for (int i = 0; i < V; i++)
            if (visited[i] == false)
                topologicalSortUtil(i, visited, stack);

        // Print contents of stack
        while (stack.empty()==false)
            System.out.print(stack.pop() + " ");
    }

    // Driver method
    public static void main(String args[])
    {
        // Create a graph given in the above diagram
        Graph g = new Graph(6);
        g.addEdge(5, 2);
        g.addEdge(5, 0);
        g.addEdge(4, 0);
        g.addEdge(4, 1);
        g.addEdge(2, 3);
        g.addEdge(3, 1);

        System.out.println("Following is a Topological " +
            "sort of the given graph");
        g.topologicalSort();
    }
}
// This code is contributed by Aakash Hasija

```

Python

```

#Python program to print topological sorting of a DAG
from collections import defaultdict

#Class to represent a graph
class Graph:
    def __init__(self,vertices):
        self.graph = defaultdict(list) #dictionary containing adjacency List

```

```

self.V = vertices #No. of vertices

# function to add an edge to graph
def addEdge(self,u,v):
    self.graph[u].append(v)

# A recursive function used by topologicalSort
def topologicalSortUtil(self,v,visited,stack):

    # Mark the current node as visited.
    visited[v] = True

    # Recur for all the vertices adjacent to this vertex
    for i in self.graph[v]:
        if visited[i] == False:
            self.topologicalSortUtil(i,visited,stack)

    # Push current vertex to stack which stores result
    stack.insert(0,v)

# The function to do Topological Sort. It uses recursive
# topologicalSortUtil()
def topologicalSort(self):
    # Mark all the vertices as not visited
    visited = [False]*self.V
    stack =[]

    # Call the recursive helper function to store Topological
    # Sort starting from all vertices one by one
    for i in range(self.V):
        if visited[i] == False:
            self.topologicalSortUtil(i,visited,stack)

    # Print contents of stack
    print stack

g= Graph(6)
g.addEdge(5, 2);
g.addEdge(5, 0);
g.addEdge(4, 0);
g.addEdge(4, 1);
g.addEdge(2, 3);
g.addEdge(3, 1);

print "Following is a Topological Sort of the given graph"
g.topologicalSort()
#This code is contributed by Neelam Yadav

```

Output:

```

Following is a Topological Sort of the given graph
5 4 2 3 1 0

```

Time Complexity: The above algorithm is simply DFS with an extra stack. So time complexity is same as DFS which is $O(V+E)$.

Applications:

Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs. In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in makefiles, data serialization, and resolving symbol dependencies in linkers [2].

Related Articles:

[Kahn's algorithm for Topological Sorting](#) : Another $O(V + E)$ algorithm.

[All Topological Sorts of a Directed Acyclic Graph](#)

References:

<http://www.personal.kent.edu/~muhamma/Algorithms/MyAlgorithms/GraphAlgor/topoSort.htm>

http://en.wikipedia.org/wiki/Topological_sorting

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Graph
DFS
Topological Sorting

Check whether a given graph is Bipartite or not

A **Bipartite Graph** is a graph whose vertices can be divided into two independent sets, U and V such that every edge (u, v) either connects a vertex from U to V or a vertex from V to U. In other words, for every edge (u, v), either u belongs to U and v to V, or u belongs to V and v to U. We can also say that there is no edge that connects vertices of same set.

Bipartite1



A bipartite graph is possible if the graph coloring is possible using two colors such that vertices in a set are colored with the same color. Note that it is possible to color a cycle graph with even cycle using two colors. For example, see the following graph.



It is not possible to color a cycle graph with odd cycle using two colors.



Algorithm to check if a graph is Bipartite:

One approach is to check whether the graph is 2-colorable or not using [backtracking algorithm m coloring problem](#).

Following is a simple algorithm to find out whether a given graph is Bipartite or not using Breadth First Search (BFS).

1. Assign RED color to the source vertex (putting into set U).
2. Color all the neighbors with BLUE color (putting into set V).
3. Color all neighbor's neighbor with RED color (putting into set U).
4. This way, assign color to all vertices such that it satisfies all the constraints of m way coloring problem where $m = 2$.
5. While assigning colors, if we find a neighbor which is colored with same color as current vertex, then the graph cannot be colored with 2 vertices (or graph is not Bipartite)

C++

```
// C++ program to find out whether a given graph is Bipartite or not
#include <iostream>
#include <queue>
#define V 4
using namespace std;

// This function returns true if graph G[V][V] is Bipartite, else false
bool isBipartite(int G[][V], int src)
{
    // Create a color array to store colors assigned to all vertices. Vertex
    // number is used as index in this array. The value '-1' of colorArr[i]
    // is used to indicate that no color is assigned to vertex 'i'. The value
    // 1 is used to indicate first color is assigned and value 0 indicates
    // second color is assigned.
    int colorArr[V];
    for (int i = 0; i < V; ++i)
        colorArr[i] = -1;

    // Assign first color to source
    colorArr[src] = 1;

    // Create a queue (FIFO) of vertex numbers and enqueue source vertex
    // for BFS traversal
    queue <int> q;
    q.push(src);

    // Run while there are vertices in queue (Similar to BFS)
    while (!q.empty())
    {
        // Dequeue a vertex from queue ( Refer http://goo.gl/35oz8 )
        int u = q.front();
        q.pop();

        // Find all non-colored adjacent vertices
        for (int v = 0; v < V; ++v)
        {
            // An edge from u to v exists and destination v is not colored
            if (G[u][v] && colorArr[v] == -1)
            {
                // Assign alternate color to this adjacent v of u
                colorArr[v] = 1 - colorArr[u];
                q.push(v);
            }

            // An edge from u to v exists and destination v is colored with
            // same color as u
            else if (G[u][v] && colorArr[v] == colorArr[u])
                return false;
        }
    }

    // If we reach here, then all adjacent vertices can be colored with
    // alternate color
    return true;
}

// Driver program to test above function
int main()
{
    int G[][V] = {{0, 1, 0, 1},
                  {1, 0, 1, 0},
                  {0, 1, 0, 1},
                  {1, 0, 1, 0}};

    isBipartite(G, 0) ? cout << "Yes" : cout << "No";
    return 0;
}
```

Java

```
// Java program to find out whether a given graph is Bipartite or not
import java.util.*;
import java.lang.*;
import java.io.*;
```

```

class Bipartite
{
    final static int V = 4; // No. of Vertices

    // This function returns true if graph G[V][V] is Bipartite, else false
    boolean isBipartite(int G[][],int src)
    {
        // Create a color array to store colors assigned to all vertices.
        // Vertex number is used as index in this array. The value '-1'
        // of colorArr[] is used to indicate that no color is assigned
        // to vertex 'i'. The value 1 is used to indicate first color
        // is assigned and value 0 indicates second color is assigned.
        int colorArr[] = new int[V];
        for (int i=0; i<V; ++i)
            colorArr[i] = -1;

        // Assign first color to source
        colorArr[src] = 1;

        // Create a queue (FIFO) of vertex numbers and enqueue
        // source vertex for BFS traversal
        LinkedList<Integer> q = new LinkedList<Integer>();
        q.add(src);

        // Run while there are vertices in queue (Similar to BFS)
        while (q.size() != 0)
        {
            // Dequeue a vertex from queue
            int u = q.poll();

            // Find all non-colored adjacent vertices
            for (int v=0; v<V; ++v)
            {
                // An edge from u to v exists and destination v is
                // not colored
                if (G[u][v]==1 && colorArr[v]==-1)
                {
                    // Assign alternate color to this adjacent v of u
                    colorArr[v] = 1-colorArr[u];
                    q.add(v);
                }

                // An edge from u to v exists and destination v is
                // colored with same color as u
                else if (G[u][v]==1 && colorArr[v]==colorArr[u])
                    return false;
            }
        }
        // If we reach here, then all adjacent vertices can
        // be colored with alternate color
        return true;
    }

    // Driver program to test above function
    public static void main (String[] args)
    {
        int G[][] = {{0, 1, 0, 1},
                     {1, 0, 1, 0},
                     {0, 1, 0, 1},
                     {1, 0, 1, 0}
                    };
        Bipartite b = new Bipartite();
        if (b.isBipartite(G, 0))
            System.out.println("Yes");
        else
            System.out.println("No");
    }
}

// Contributed by Aakash Hasija

```

Output:

```
Yes
```

The above algorithm works only if the graph is strongly connected. In above code, we always start with source 0 and assume that vertices are visited from it. One important observation is a graph with no edges is also Bipartite. Note that the Bipartite condition says all edges should be from one set to another.

We can extend the above code to handle cases when a graph is not connected. The idea is repeatedly call above method for all not yet visited vertices.

```

// C++ program to find out whether a given graph is Bipartite or not.
// It works for disconnected graph also.
#include <bits/stdc++.h>
using namespace std;

const int V = 4;

// This function returns true if graph G[V][V] is Bipartite,
// else false
bool isBipartiteUtil(int G[][V], int src, int colorArr[])
{
    colorArr[src] = 1;

    // Create a queue (FIFO) of vertex numbers and enqueue
    // source vertex for BFS traversal
    queue <int> q;
    q.push(src);

    // Run while there are vertices in queue (Similar to BFS)
    while (!q.empty())
    {
        // Dequeue a vertex from queue ( Refer http://goo.gl/35oz8 )
        int u = q.front();
        q.pop();

        // Find all non-colored adjacent vertices
        for (int v = 0; v < V; ++v)
        {
            // An edge from u to v exists and
            // destination v is not colored
            if (G[u][v] && colorArr[v] == -1)
            {
                // Assign alternate color to this

```

```

    // adjacent v of u
    colorArr[v] = 1 - colorArr[u];
    q.push(v);
}

// An edge from u to v exists and destination
// v is colored with same color as u
else if (G[u][v] && colorArr[v] == colorArr[u])
    return false;
}
}

// If we reach here, then all adjacent vertices can
// be colored with alternate color
return true;
}

// Returns true if G[] is Bipartite, else false
bool isBipartite(int G[][V])
{
    // Create a color array to store colors assigned to all
    // vertices. Vertex number is used as index in this
    // array. The value '-1' of colorArr[i] is used to
    // indicate that no color is assigned to vertex 'i'.
    // The value 1 is used to indicate first color is
    // assigned and value 0 indicates second color is
    // assigned.
    int colorArr[V];
    for (int i = 0; i < V; ++i)
        colorArr[i] = -1;

    // This code is to handle disconnected graph
    for (int i = 0; i < V; i++)
        if (colorArr[i] == -1)
            if (!isBipartiteUtil(G, i, colorArr))
                return false;

    return true;
}

// Driver program to test above function
int main()
{
    int G[][V] = {{0, 1, 0, 1},
                  {1, 0, 1, 0},
                  {0, 1, 0, 1},
                  {1, 0, 1, 0}};

    isBipartite(G) ? cout << "Yes" : cout << "No";
    return 0;
}

```

Output:

Yes

Time Complexity of the above approach is same as that Breadth First Search. In above implementation is $O(V^2)$ where V is number of vertices. If graph is represented using adjacency list, then the complexity becomes $O(V+E)$.

Exercise:

1. Can DFS algorithm be used to check the bipartite-ness of a graph? If yes, how?

References:

http://en.wikipedia.org/wiki/Graph_coloring

http://en.wikipedia.org/wiki/Bipartite_graph

This article is compiled by [Aashish Barnwal](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

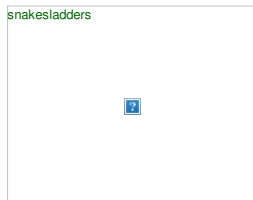
GATE CS Corner Company Wise Coding Practice

Graph
BFS

Snake and Ladder Problem

Given a snake and ladder board, find the minimum number of dice throws required to reach the destination or last cell from source or 1st cell. Basically, the player has total control over outcome of dice throw and wants to find out minimum number of throws required to reach last cell.

If the player reaches a cell which is base of a ladder, the player has to climb up that ladder and if reaches a cell is mouth of the snake, has to go down to the tail of snake without a dice throw.



For example consider the board shown on right side (taken from [here](#)), the minimum number of dice throws required to reach cell 30 from cell 1 is 3. Following are steps.

- First throw two on dice to reach cell number 3 and then ladder to reach 22
- Then throw 6 to reach 28.
- Finally through 2 to reach 30.

There can be other solutions as well like (2, 2, 6), (2, 4, 4), (2, 3, 5).. etc.

We strongly recommend to minimize the browser and try this yourself first.

The idea is to consider the given snake and ladder board as a directed graph with number of vertices equal to the number of cells in the board. The problem reduces to finding the shortest path in a graph. Every vertex of the graph has an edge to next six vertices if next 6 vertices do not have a snake or ladder. If any of the next six vertices has a snake or ladder, then the edge from current vertex goes to the top of the ladder or tail of the snake. Since all edges are of equal weight, we can efficiently find shortest path using [Breadth First Search](#) of the graph.

Following is C++ implementation of the above idea. The input is represented by two things, first is 'N' which is number of cells in the given board, second is an array 'move[0...N-1]' of size N. An entry move[i] is -1 if there is no snake and no ladder from i, otherwise move[i] contains index of destination cell for the snake or the ladder at i.

```

// C++ program to find minimum number of dice throws required to
// reach last cell from first cell of a given snake and ladder
// board

```

```

#include<iostream>
#include <queue>
using namespace std;

// An entry in queue used in BFS
struct queueEntry
{
    int v; // Vertex number
    int dist; // Distance of this vertex from source
};

// This function returns minimum number of dice throws required to
// Reach last cell from 0'th cell in a snake and ladder game.
// move[] is an array of size N where N is no. of cells on board
// If there is no snake or ladder from cell i, then move[i] is -1
// Otherwise move[i] contains cell to which snake or ladder at i
// takes to.
int getMinDiceThrows(int move[], int N)
{
    // The graph has N vertices. Mark all the vertices as
    // not visited
    bool *visited = new bool[N];
    for (int i = 0; i < N; i++)
        visited[i] = false;

    // Create a queue for BFS
    queue<queueEntry> q;

    // Mark the node 0 as visited and enqueue it.
    visited[0] = true;
    queueEntry s = {0, 0}; // distance of 0'th vertex is also 0
    q.push(s); // Enqueue 0'th vertex

    // Do a BFS starting from vertex at index 0
    queueEntry qe; // A queue entry (qe)
    while (!q.empty())
    {
        qe = q.front();
        int v = qe.v; // vertex no. of queue entry

        // If front vertex is the destination vertex,
        // we are done
        if (v == N-1)
            break;

        // Otherwise dequeue the front vertex and enqueue
        // its adjacent vertices (or cell numbers reachable
        // through a dice throw)
        q.pop();
        for (int j=v+1; j<=(v+6) && j<N; ++j)
        {
            // If this cell is already visited, then ignore
            if (visited[j])
                continue;

            // Otherwise calculate its distance and mark it
            // as visited
            queueEntry a;
            a.dist = (qe.dist + 1);
            visited[j] = true;

            // Check if there a snake or ladder at 'j'
            // then tail of snake or top of ladder
            // become the adjacent of 'j'
            if (move[j] != -1)
                a.v = move[j];
            else
                a.v = j;
            q.push(a);
        }
    }

    // We reach here when 'qe' has last vertex
    // return the distance of vertex in 'qe'
    return qe.dist;
}

// Driver program to test methods of graph class
int main()
{
    // Let us construct the board given in above diagram
    int N = 30;
    int moves[N];
    for (int i = 0; i < N; i++)
        moves[i] = -1;

    // Ladders
    moves[2] = 21;
    moves[4] = 7;
    moves[10] = 25;
    moves[19] = 28;

    // Snakes
    moves[26] = 0;
    moves[20] = 8;
    moves[16] = 3;
    moves[18] = 6;

    cout << "Min Dice throws required is " << getMinDiceThrows(moves, N);
    return 0;
}

```

Output:

```
Min Dice throws required is 3
```

Time complexity of the above solution is $O(N)$ as every cell is added and removed only once from queue. And a typical enqueue or dequeue operation takes $O(1)$ time.

This article is contributed by **Siddharth**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Graph
BFS
shortest path

Minimize Cash Flow among a given set of friends who have borrowed money from each other

Given a number of friends who have to give or take some amount of money from one another. Design an algorithm by which the total cash flow among all the friends is minimized.

Example:

Following diagram shows input debts to be settled.

cashFlow



Above debts can be settled in following optimized way

cashFlow



We strongly recommend to minimize your browser and try this yourself first.

The idea is to use **Greedy algorithm** where at every step, settle all amounts of one person and recur for remaining n-1 persons.

How to pick the first person? To pick the first person, calculate the net amount for every person where net amount is obtained by subtracting all debts (amounts to pay) from all credits (amounts to be paid). Once net amount for every person is evaluated, find two persons with maximum and minimum net amounts. These two persons are the most creditors and debtors. The person with minimum of two is our first person to be settled and removed from list. Let the minimum of two amounts be x. We pay 'x' amount from the maximum debtor to maximum creditor and settle one person. If x is equal to the maximum debit, then maximum debtor is settled, else maximum creditor is settled.

The following is detailed algorithm.

Do following for every person P_i where i is from 0 to n-1.

1) Compute the net amount for every person. The net amount for person 'i' can be computed by subtracting sum of all debts from sum of all credits.

2) Find the two persons that are maximum creditor and maximum debtor. Let the maximum amount to be credited maximum creditor be maxCredit and maximum amount to be debited from maximum debtor be maxDebit. Let the maximum debtor be Pd and maximum creditor be Pc.

3) Find the minimum of maxDebit and maxCredit. Let minimum of two be x. Debit 'x' from Pd and credit this amount to Pc

4) If x is equal to maxCredit, then remove Pc from set of persons and recur for remaining (n-1) persons.

5) If x is equal to maxDebit, then remove Pd from set of persons and recur for remaining (n-1) persons.

Thanks to Balaji S for suggesting this method in a comment [here](#).

The following is C++ implementation of above algorithm.

```
// C++ program to find maximum cash flow among a set of persons
#include<iostream>
using namespace std;

// Number of persons (or vertices in the graph)
#define N 3

// A utility function that returns index of minimum value in arr[]
int getMin(int arr[])
{
    int minInd = 0;
    for (int i=1; i<N; i++)
        if (arr[i] < arr[minInd])
            minInd = i;
    return minInd;
}

// A utility function that returns index of maximum value in arr[]
int getMax(int arr[])
{
    int maxInd = 0;
    for (int i=1; i<N; i++)
        if (arr[i] > arr[maxInd])
            maxInd = i;
    return maxInd;
}

// A utility function to return minimum of 2 values
int minOf2(int x, int y)
{
    return (x<y)? x: y;
}

// amount[p] indicates the net amount to be credited/debited
// to/from person 'p'
// If amount[p] is positive, then i'th person will amount[i]
// If amount[p] is negative, then i'th person will give -amount[i]
void minCashFlowRec(int amount[])
{
    // Find the indexes of minimum and maximum values in amount[]
    // amount[mxCredit] indicates the maximum amount to be given
    // (or credited) to any person.
    // And amount[mxDebit] indicates the maximum amount to be taken
    // (or debited) from any person.
    // So if there is a positive value in amount[], then there must
    // be a negative value
    int mxCredit = getMax(amount), mxDebit = getMin(amount);

    // If both amounts are 0, then all amounts are settled
    if (amount[mxCredit] == 0 && amount[mxDebit] == 0)
        return;

    // Find the minimum of two amounts
    int min = minOf2(-amount[mxDebit], amount[mxCredit]);
```



```

amount[mxCredit] -= min;
amount[mxDebit] += min;

// If minimum is the maximum amount to be
cout << "Person " << mxDebit << " pays " << min
<< " to " << "Person " << mxCredit << endl;

// Recur for the amount array. Note that it is guaranteed that
// the recursion would terminate as either amount[mxCredit]
// or amount[mxDebit] becomes 0
minCashFlowRec(amount);
}

// Given a set of persons as graph[] where graph[i][j] indicates
// the amount that person i needs to pay person j, this function
// finds and prints the minimum cash flow to settle all debts.
void minCashFlow(int graph[][N])
{
    // Create an array amount[], initialize all value in it as 0.
    int amount[N] = {0};

    // Calculate the net amount to be paid to person 'p', and
    // stores it in amount[p]. The value of amount[p] can be
    // calculated by subtracting debts of 'p' from credits of 'p'
    for (int p=0; p<N; p++)
        for (int i=0; i<N; i++)
            amount[p] += (graph[i][p] - graph[p][i]);

    minCashFlowRec(amount);
}

// Driver program to test above function
int main()
{
    // graph[i][j] indicates the amount that person i needs to
    // pay person j
    int graph[N][N] = { {0, 1000, 2000},
                        {0, 0, 5000},
                        {0, 0, 0} };

    // Print the solution
    minCashFlow(graph);
    return 0;
}

```

Output:

```

Person 1 pays 4000 to Person 2
Person 0 pays 3000 to Person 2

```

Algorithmic Paradigm: Greedy

Time Complexity: $O(N^2)$ where N is the number of persons.

This article is contributed by **Gaurav**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Graph
Greedy
Greedy Algorithm

Boggle (Find all possible words in a board of characters) | Set 1

Given a dictionary, a method to do lookup in dictionary and a M x N board where every cell has one character. Find all possible words that can be formed by a sequence of adjacent characters. Note that we can move to any of 8 adjacent characters, but a word should not have multiple instances of same cell.

Example:

```

Input: dictionary[] = {"GEEKS", "FOR", "QUIZ", "GO"};
boggle[][] = {{'G','I','Z'},
               {'U','E','K'},
               {'Q','S','E'}};
isWord(str): returns true if str is present in dictionary
            else false.

Output: Following words of dictionary are present
GEEKS
QUIZ

```

Boggle



We strongly recommend that you click here and practice it, before moving on to the solution.

The idea is to consider every character as a starting character and find all words starting with it. All words starting from a character can be found using [Depth First Traversal](#). We do depth first traversal starting from every cell. We keep track of visited cells to make sure that a cell is considered only once in a word.

```

// C++ program for Boggle game
#include<iostream>
#include<string>
using namespace std;

#define M 3
#define N 3

// Let the given dictionary be following
string dictionary[] = {"GEEKS", "FOR", "QUIZ", "GO"};
int n = sizeof(dictionary)/sizeof(dictionary[0]);

// A given function to check if a given string is present in

```

```

// dictionary. The implementation is naive for simplicity. As
// per the question dictionary is given to us.
bool isWord(string &str)
{
    // Linearly search all words
    for (int i=0; i<n; i++)
        if (str.compare(dictionary[i]) == 0)
            return true;
    return false;
}

// A recursive function to print all words present on boggle
void findWordsUtil(char boggle[M][N], bool visited[M][N], int i,
    int j, string &str)
{
    // Mark current cell as visited and append current character
    // to str
    visited[i][j] = true;
    str = str + boggle[i][j];

    // If str is present in dictionary, then print it
    if (isWord(str))
        cout << str << endl;

    // Traverse 8 adjacent cells of boggle[i][j]
    for (int row=i-1; row<=i+1 && row<M; row++)
        for (int col=j-1; col<=j+1 && col<N; col++)
            if (row>=0 && col>=0 && !visited[row][col])
                findWordsUtil(boggle, visited, row, col, str);

    // Erase current character from string and mark visited
    // of current cell as false
    str.erase(str.length()-1);
    visited[i][j] = false;
}

// Prints all words present in dictionary.
void findWords(char boggle[M][N])
{
    // Mark all characters as not visited
    bool visited[M][N] = {{false}};

    // Initialize current string
    string str = "";

    // Consider every character and look for all words
    // starting with this character
    for (int i=0; i<M; i++)
        for (int j=0; j<N; j++)
            findWordsUtil(boggle, visited, i, j, str);
}

// Driver program to test above function
int main()
{
    char boggle[M][N] = {{'G','I','Z'},
        {'U','E','K'},
        {'Q','S','E'}};

    cout << "Following words of dictionary are present\n";
    findWords(boggle);
    return 0;
}

```

Output:

```

Following words of dictionary are present
GEEKS
QUIZ

```

Note that the above solution may print same word multiple times. For example, if we add "SEEK" to dictionary, it is printed multiple times. To avoid this, we can use hashing to keep track of all printed words.

In below set 2, we have discussed Trie based optimized solution:

[Boggle | Set 2 \(Using Trie\)](#)

This article is contributed by **Rishabh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

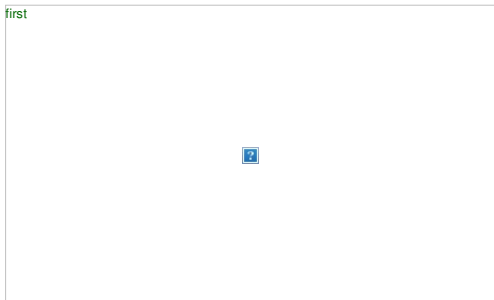
GATE CS Corner Company Wise Coding Practice

Graph
DFS

Assign directions to edges so that the directed graph remains acyclic

Given a graph with both directed and undirected edges. It is given that the directed edges don't form cycle. How to assign directions to undirected edges so that the graph (with all directed edges) remains acyclic even after the assignment?

For example, in the below graph, blue edges don't have directions.



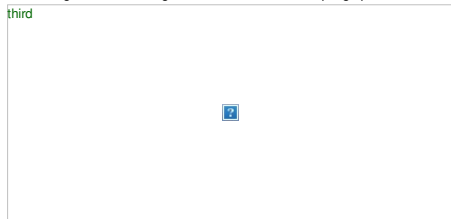
We strongly recommend to minimize your browser and try this yourself first.

The idea is to use [Topological Sorting](#). Following are two steps used in the algorithm.

1) Consider the subgraph with directed edges only and find topological sorting of the subgraph. In the above example, topological sorting is {0, 5, 1, 2, 3, 4}. Below diagram shows topological sorting for the above example graph.



2) Use above topological sorting to assign directions to undirected edges. For every undirected edge (u, v) , assign it direction from u to v if u comes before v in topological sorting, else assign it direction from v to u . Below diagram shows assigned directions in the example graph.



Source: <http://courses.csail.mit.edu/6.006/oldquizzes/solutions/q2-f2009-sol.pdf>

This article is contributed by **Aditya Agrawal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Graph
graph-connectivity
Topological Sorting

Memory efficient doubly linked list

Asked by Varun Bhatia.

Question:

Write a code for implementation of doubly linked list with use of single pointer in each node.

Solution:

This question is solved and very well explained at <http://www.linuxjournal.com/article/6828>.

We also recommend to read http://en.wikipedia.org/wiki/XOR_linked_list

GATE CS Corner Company Wise Coding Practice

Linked Lists
XOR

XOR Linked List – A Memory Efficient Doubly Linked List | Set 1

An ordinary Doubly Linked List requires space for two address fields to store the addresses of previous and next nodes. A memory efficient version of Doubly Linked List can be created using only one space for address field with every node. This memory efficient Doubly Linked List is called XOR Linked List or Memory Efficient as the list uses bitwise XOR operation to save space for one address. In the XOR linked list, instead of storing actual memory addresses, every node stores the XOR of addresses of previous and next nodes.



Consider the above Doubly Linked List. Following are the Ordinary and XOR (or Memory Efficient) representations of the Doubly Linked List.

Ordinary Representation:

Node A:

prev = NULL, next = add(B) // previous is NULL and next is address of B

Node B:

prev = add(A), next = add(C) // previous is address of A and next is address of C

Node C:

prev = add(B), next = add(D) // previous is address of B and next is address of D

Node D:

prev = add(C), next = NULL // previous is address of C and next is NULL

XOR List Representation:

Let us call the address variable in XOR representation npx (XOR of next and previous)

Node A:

npx = 0 XOR add(B) // bitwise XOR of zero and address of B

Node B:

npx = add(A) XOR add(C) // bitwise XOR of address of A and address of C

Node C:

`npx = add(B) XOR add(D) // bitwise XOR of address of B and address of D`

Node D:

`npx = add(C) XOR 0 // bitwise XOR of address of C and 0`

Traversal of XOR Linked List:

We can traverse the XOR list in both forward and reverse direction. While traversing the list we need to remember the address of the previously accessed node in order to calculate the next node's address. For example when we are at node C, we must have address of B. XOR of `add(B)` and `npx` of C gives us the `add(D)`. The reason is simple: `npx(C)` is "`add(B) XOR add(D)`". If we do xor of `npx(C)` with `add(B)`, we get the result as "`add(B) XOR add(D) XOR add(B)`" which is "`add(D) XOR 0`" which is "`add(D)`". So we have the address of next node. Similarly we can traverse the list in backward direction.

We have covered more on XOR Linked List in the following post.

[XOR Linked List – A Memory Efficient Doubly Linked List | Set 2](#)

References:

http://en.wikipedia.org/wiki/XOR_linked_list

<http://www.linuxjournal.com/article/6828?page=0,0>

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Linked Lists
Advanced Data Structures
XOR

XOR Linked List – A Memory Efficient Doubly Linked List | Set 2

In the [previous post](#), we discussed how a Doubly Linked can be created using only one space for address field with every node. In this post, we will discuss implementation of memory efficient doubly linked list. We will mainly discuss following two simple functions.

- 1) A function to insert a new node at the beginning.
- 2) A function to traverse the list in forward direction.

In the following code, `insert()` function inserts a new node at the beginning. We need to change the head pointer of Linked List, that is why a double pointer is used (See [this](#)). Let us first discuss few things again that have been discussed in the [previous post](#). We store XOR of next and previous nodes with every node and we call it `npx`, which is the only address member we have with every node. When we insert a new node at the beginning, `npx` of new node will always be XOR of NULL and current head. And `npx` of current head must be changed to XOR of new node and node next to current head.

`printList()` traverses the list in forward direction. It prints data values from every node. To traverse the list, we need to get pointer to the next node at every point. We can get the address of next node by keeping track of current node and previous node. If we do XOR of `curr->npx` and `prev`, we get the address of next node.

```
/* C/C++ Implementation of Memory efficient Doubly Linked List */
#include <stdio.h>
#include <stdlib.h>

// Node structure of a memory efficient doubly linked list
struct node
{
    int data;
    struct node* npx; /* XOR of next and previous node */
};

/* returns XORed value of the node addresses */
struct node* XOR (struct node *a, struct node *b)
{
    return (struct node*) (((unsigned int) (a) ^ (unsigned int) (b)));
}

/* Insert a node at the beginning of the XORed linked list and makes the
newly inserted node as head */
void insert(struct node **head_ref, int data)
{
    // Allocate memory for new node
    struct node *new_node = (struct node *) malloc (sizeof (struct node) );
    new_node->data = data;

    /* Since new node is being inserted at the beginning, npx of new node
will always be XOR of current head and NULL */
    new_node->npx = XOR(*head_ref, NULL);

    /* If linked list is not empty, then npx of current head node will be XOR
of new node and node next to current head */
    if (*head_ref != NULL)
    {
        /* (*head_ref)->npx is XOR of NULL and next. So if we do XOR of
        // it with NULL, we get next
        struct node* next = XOR((*head_ref)->npx, NULL);
        (*head_ref)->npx = XOR(new_node, next);
    }

    // Change head
    *head_ref = new_node;
}

// prints contents of doubly linked list in forward direction
void printList (struct node *head)
{
    struct node *curr = head;
    struct node *prev = NULL;
    struct node *next;

    printf ("Following are the nodes of Linked List: \n");

    while (curr != NULL)
    {
        // print current node
        printf ("%d ", curr->data);

        // get address of next node: curr->npx is next*prev, so curr->npx^prev
        // will be next*prev*prev which is next
        next = XOR (prev, curr->npx);

        // update prev and curr for next iteration
        prev = curr;
        curr = next;
    }
}

// Driver program to test above functions
int main ()
{
    /* Create following Doubly Linked List
    head->40<->30<->20<->10 */
}
```

```

struct node *head = NULL;
insert(&head, 10);
insert(&head, 20);
insert(&head, 30);
insert(&head, 40);

// print the created list
printList(head);

return (0);
}

```

Output:

```

Following are the nodes of Linked List:
40 30 20 10

```

Note that XOR of pointers is not defined by C/C++ standard. So the above implementation may not work on all platforms.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Linked Lists
Advanced Data Structures
XOR

Skip List | Set 1 (Introduction)

Can we search in a sorted linked list in better than $O(n)$ time?

The worst case search time for a sorted linked list is $O(n)$ as we can only linearly traverse the list and cannot skip nodes while searching. For a Balanced Binary Search Tree, we skip almost half of the nodes after one comparison with root. For a sorted array, we have random access and we can apply Binary Search on arrays.

Can we augment sorted linked lists to make the search faster? The answer is [Skip List](#). The idea is simple, we create multiple layers so that we can skip some nodes. See the following example list with 16 nodes and two layers. The upper layer works as an "express lane" which connects only main outer stations, and the lower layer works as a "normal lane" which connects every station. Suppose we want to search for 50, we start from first node of "express lane" and keep moving on "express lane" till we find a node whose next is greater than 50. Once we find such a node (30 is the node in following example) on "express lane", we move to "normal lane" using pointer from this node, and linearly search for 50 on "normal lane". In following example, we start from 30 on "normal lane" and with linear search, we find 50.



What is the time complexity with two layers? The worst case time complexity is number of nodes on "express lane" plus number of nodes in a segment (A segment is number of "normal lane" nodes between two "express lane" nodes) of "normal lane". So if we have n nodes on "normal lane", \sqrt{n} (square root of n) nodes on "express lane" and we equally divide the "normal lane", then there will be \sqrt{n} nodes in every segment of "normal lane". \sqrt{n} is actually optimal division with two layers. With this arrangement, the number of nodes traversed for a search will be $O(\sqrt{n})$. Therefore, with $O(\sqrt{n})$ extra space, we are able to reduce the time complexity to $O(\sqrt{n})$.

Can we do better?

The time complexity of skip lists can be reduced further by adding more layers. In fact, the time complexity of search, insert and delete can become $O(\log n)$ in average case. We will soon be publishing more posts on Skip Lists.

References

MIT Video Lecture on Skip Lists
http://en.wikipedia.org/wiki/Skip_list

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Advanced Data Structures

Self Organizing List | Set 1 (Introduction)

The worst case search time for a sorted linked list is $O(n)$. With a Balanced Binary Search Tree, we can skip almost half of the nodes after one comparison with root. For a sorted array, we have random access and we can apply Binary Search on arrays.

One idea to make search faster for Linked Lists is [Skip List](#). Another idea (which is discussed in this post) is to *place more frequently accessed items closer to head*. There can be two possibilities. offline (we know the complete search sequence in advance) and online (we don't know the search sequence).

In case of offline, we can put the nodes according to decreasing frequencies of search (The element having maximum search count is put first). For many practical applications, it may be difficult to obtain search sequence in advance. A [Self Organizing list](#) reorders its nodes based on searches which are done. The idea is to use locality of reference (In a typical database, 80% of the access are to 20% of the items). Following are different strategies used by Self Organizing Lists.

1) Move-to-Front Method: Any node searched is moved to the front. This strategy is easy to implement, but it may over-reward infrequently accessed items as it always move the item to front.

2) Count Method: Each node stores count of the number of times it was searched. Nodes are ordered by decreasing count. This strategy requires extra space for storing count.

3) Transpose Method: Any node searched is swapped with the preceding node. Unlike Move-to-front, this method does not adapt quickly to changing access patterns.

Competitive Analysis:

The worst case time complexity of all methods is $O(n)$. In worst case, the searched element is always the last element in list. For [average case analysis](#), we need probability distribution of search sequences which is not available many times.

For online strategies and algorithms like above, we have a totally different way of analyzing them called *competitive analysis* where performance of an online algorithm is compared to the performance of an optimal offline algorithm (that can view the sequence of requests in advance). Competitive analysis is used in many practical algorithms like caching, disk paging, high performance computers. The best thing about competitive analysis is, we don't need to assume anything about probability distribution of input. The Move-to-front method is 4-competitive, means it never does more than a factor of 4 operations than offline algorithm (See [the MIT video lecture](#) for proof).

We will soon be discussing implementation and proof of the analysis given in the video lecture.

References:

http://en.wikipedia.org/wiki/Self-organizing_list
MIT Video Lecture
http://www.eecs.yorku.ca/course_archive/2003-04/F/2011/2011A/DatStr_071_SOLists.pdf
[http://en.wikipedia.org/wiki/Competitive_analysis_\(online_algorithm\)](http://en.wikipedia.org/wiki/Competitive_analysis_(online_algorithm))

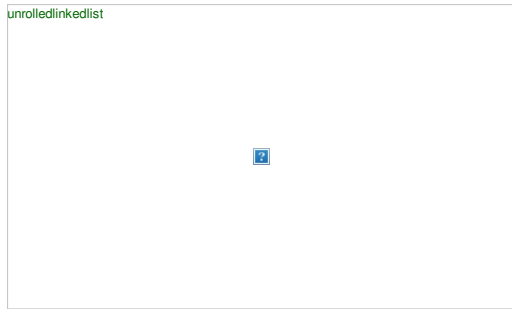
This article is compiled by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Advanced Data Structures

Unrolled Linked List | Set 1 (Introduction)

Like array and linked list, unrolled Linked List is also a linear data structure and is a variant of linked list. Unlike simple linked list, it stores multiple elements at each node. That is, instead of storing single element at a node, unrolled linked lists store an array of elements at a node. Unrolled linked list covers advantages of both array and linked list as it reduces the memory overhead in comparison to simple linked lists by storing multiple elements at each node and it also has the advantage of fast insertion and deletion as that of a linked list.



Advantages:

- Because of the Cache behavior, linear search is much faster in unrolled linked lists.
- In comparison to ordinary linked list, it requires less storage space for pointers/references.
- It performs operations like insertion, deletion and traversal more quickly than ordinary linked lists (because search is faster).

Disadvantages:

- The overhead per node is comparatively high than singly linked lists. Refer an example node in below code.

Simple Implementation in C

The below program creates a simple unrolled linked list with 3 nodes containing variable number of elements in each. It also traverses the created list.

```
// C program to implement unrolled linked list
// and traversing it.
#include<stdio.h>
#include<stdlib.h>
#define maxElements 4

// Unrolled Linked List Node
struct Node
{
    int numElements;
    int array[maxElements];
    struct Node *next;
};

/* Function to traverse an unrolled linked list
and print all the elements*/
void printUnrolledList(struct Node *n)
{
    while (n != NULL)
    {
        // Print elements in current node
        for (int i=0; i<n->numElements; i++)
            printf("%d ", n->array[i]);

        // Move to next node
        n = n->next;
    }
}

// Program to create an unrolled linked list
// with 3 Nodes
int main()
{
    struct Node* head = NULL;
    struct Node* second = NULL;
    struct Node* third = NULL;

    // allocate 3 Nodes
    head = (struct Node*)malloc(sizeof(struct Node));
    second = (struct Node*)malloc(sizeof(struct Node));
    third = (struct Node*)malloc(sizeof(struct Node));

    // Let us put some values in second node (Number
    // of values must be less than or equal to
    // maxElement)
    head->numElements = 3;
    head->array[0] = 1;
    head->array[1] = 2;
    head->array[2] = 3;

    // Link first Node with the second Node
    head->next = second;

    // Let us put some values in second node (Number
    // of values must be less than or equal to
    // maxElement)
    second->numElements = 3;
    second->array[0] = 4;
    second->array[1] = 5;
    second->array[2] = 6;

    // Link second Node with the third Node
    second->next = third;

    // Let us put some values in third node (Number
    // of values must be less than or equal to
    // maxElement)
    third->numElements = 3;
    third->array[0] = 7;
    third->array[1] = 8;
    third->array[2] = 9;
    third->next = NULL;

    printUnrolledList(head);

    return 0;
}
```

Output:

In this article, we have introduced unrolled list and advantages of it. We have also shown how to traverse the list. In the next article, we will be discussing insertion, deletion and values of maxElements/numElements in detail.

This article is contributed by **Harsh Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Linked Lists

Segment Tree | Set 1 (Sum of given range)

Let us consider the following problem to understand Segment Trees.

We have an array $arr[0 \dots n-1]$. We should be able to

1 Find the sum of elements from index l to r where $0 \leq l \leq r \leq n-1$

2 Change value of a specified element of the array to a new value x . We need to do $arr[i] = x$ where $0 \leq i \leq n-1$.

A **simple solution** is to run a loop from l to r and calculate sum of elements in given range. To update a value, simply do $arr[i] = x$. The first operation takes $O(n)$ time and second operation takes $O(1)$ time.

Another solution is to create another array and store sum from start to i at the i th index in this array. Sum of a given range can now be calculated in $O(1)$ time, but update operation takes $O(n)$ time now. This works well if the number of query operations are large and very few updates.

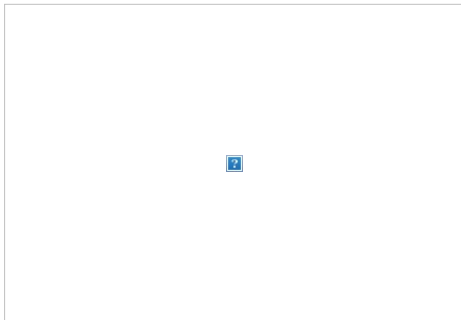
What if the number of query and updates are equal? **Can we perform both the operations in $O(\log n)$ time once given the array?** We can use a Segment Tree to do both operations in $O(\log n)$ time.

Representation of Segment trees

1. Leaf Nodes are the elements of the input array.

2. Each internal node represents some merging of the leaf nodes. The merging may be different for different problems. For this problem, merging is sum of leaves under a node.

An array representation of tree is used to represent Segment Trees. For each node at index i , the left child is at index $2i+1$, right child at $2i+2$ and the parent is at $\lfloor i/2 \rfloor$.



Construction of Segment Tree from given array

We start with a segment $arr[0 \dots n-1]$. and every time we divide the current segment into two halves(if it has not yet become a segment of length 1), and then call the same procedure on both halves, and for each such segment we store the sum in corresponding node.

All levels of the constructed segment tree will be completely filled except the last level. Also, the tree will be a **Full Binary Tree** because we always divide segments in two halves at every level. Since the constructed tree is always full binary tree with n leaves, there will be $n-1$ internal nodes. So total number of nodes will be $2n - 1$.

Height of the segment tree will be $\lceil \log_2 n \rceil$. Since the tree is represented using array and relation between parent and child indexes must be maintained, size of memory allocated for segment tree will be $4n$.

Query for Sum of given range

Once the tree is constructed, how to get the sum using the constructed segment tree. Following is algorithm to get the sum of elements.

```
int getSum(node, l, r)
{
    if range of node is within l and r
        return value in node
    else if range of node is completely outside l and r
        return 0
    else
        return getSum(node's left child, l, r) +
               getSum(node's right child, l, r)
}
```

Update a value

Like tree construction and query operations, update can also be done recursively. We are given an index which needs to be updated. Let $diff$ be the value to be added. We start from root of the segment tree, and add $diff$ to all nodes which have given index in their range. If a node doesn't have given index in its range, we don't make any changes to that node.

Implementation:

Following is implementation of segment tree. The program implements construction of segment tree for any given array. It also implements query and update operations.

C

```
// C program to show segment tree operations like construction, query
// and update
#include <stdio.h>
#include <math.h>

// A utility function to get the middle index from corner indexes.
int getMid(int s, int e) { return s + (e - s)/2; }

/* A recursive function to get the sum of values in given range
of the array. The following are parameters for this function.

st --> Pointer to segment tree
si --> Index of current node in the segment tree. Initially
      0 is passed as root is always at index 0
ss & se --> Starting and ending indexes of the segment represented
           by current node, i.e., st[si]
qs & qe --> Starting and ending indexes of query range */
int getSumUtil(int *st, int ss, int se, int qs, int qe, int si)
{
    if (qs <= ss & qe >= se)
        return st[si];
    if (qs > se || qe < ss)
        return 0;
    int mid = getMid(ss, se);
    return getSumUtil(st, ss, mid, qs, qe, 2*si+1) +
           getSumUtil(st, mid+1, se, qs, qe, 2*si+2);
}
```

```

// If segment of this node is a part of given range, then return
// the sum of the segment
if (qs <= ss && qe >= se)
    return st[si];

// If segment of this node is outside the given range
if (se < qs || ss > qe)
    return 0;

// If a part of this segment overlaps with the given range
int mid = getMid(ss, se);
return getSumUtil(st, ss, mid, qs, qe, 2*si+1) +
    getSumUtil(st, mid+1, se, qs, qe, 2*si+2);
}

/* A recursive function to update the nodes which have the given
index in their range. The following are parameters
st, si, ss and se are same as getSumUtil()
i --> index of the element to be updated. This index is
in input array.
diff --> Value to be added to all nodes which have i in range */
void updateValueUtil(int *st, int ss, int se, int i, int diff, int si)
{
    // Base Case: If the input index lies outside the range of
    // this segment
    if (i < ss || i > se)
        return;

    // If the input index is in range of this node, then update
    // the value of the node and its children
    st[si] = st[si] + diff;
    if (se != ss)
    {
        int mid = getMid(ss, se);
        updateValueUtil(st, ss, mid, i, diff, 2*si + 1);
        updateValueUtil(st, mid+1, se, i, diff, 2*si + 2);
    }
}

// The function to update a value in input array and segment tree.
// It uses updateValueUtil() to update the value in segment tree
void updateValue(int arr[], int *st, int n, int i, int new_val)
{
    // Check for erroneous input index
    if (i < 0 || i > n-1)
    {
        printf("Invalid Input");
        return;
    }

    // Get the difference between new value and old value
    int diff = new_val - arr[i];

    // Update the value in array
    arr[i] = new_val;

    // Update the values of nodes in segment tree
    updateValueUtil(st, 0, n-1, i, diff, 0);
}

// Return sum of elements in range from index qs (query start)
// to qe (query end). It mainly uses getSumUtil()
int getSum(int *st, int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n-1 || qs > qe)
    {
        printf("Invalid Input");
        return -1;
    }

    return getSumUtil(st, 0, n-1, qs, qe, 0);
}

// A recursive function that constructs Segment Tree for array[ss..se].
// si is index of current node in segment tree st
int constructSTUtil(int arr[], int ss, int se, int *st, int si)
{
    // If there is one element in array, store it in current node of
    // segment tree and return
    if (ss == se)
    {
        st[si] = arr[ss];
        return arr[ss];
    }

    // If there are more than one elements, then recur for left and
    // right subtrees and store the sum of values in this node
    int mid = getMid(ss, se);
    st[si] = constructSTUtil(arr, ss, mid, st, si*2+1) +
        constructSTUtil(arr, mid+1, se, st, si*2+2);
    return st[si];
}

/* Function to construct segment tree from given array. This function
allocates memory for segment tree and calls constructSTUtil() to
fill the allocated memory */
int *constructST(int arr[], int n)
{
    // Allocate memory for segment tree

    //Height of segment tree
    int x = (int)(ceil(log2(n)));

    //Maximum size of segment tree
    int max_size = 2*(int)pow(2, x) - 1;

    // Allocate memory
    int *st = new int[max_size];

    // Fill the allocated memory st
    constructSTUtil(arr, 0, n-1, st, 0);

    // Return the constructed segment tree
    return st;
}

```



```
// Driver program to test above functions
int main()
{
    int arr[] = {1, 3, 5, 7, 9, 11};
    int n = sizeof(arr)/sizeof(arr[0]);

    // Build segment tree from given array
    int *st = constructST(arr, n);

    // Print sum of values in array from index 1 to 3
    printf("Sum of values in given range = %d\n",
        getSum(st, n, 1, 3));

    // Update: set arr[1] = 10 and update corresponding
    // segment tree nodes
    updateValue(arr, st, n, 1, 10);

    // Find sum after the value is updated
    printf("Updated sum of values in given range = %d\n",
        getSum(st, n, 1, 3));
    return 0;
}
```

Java

```
// Java Program to show segment tree operations like construction,
// query and update
class SegmentTree
{
    int st[]; // The array that stores segment tree nodes

    /* Constructor to construct segment tree from given array. This
    constructor allocates memory for segment tree and calls
    constructSTUtil() to fill the allocated memory */
    SegmentTree(int arr[], int n)
    {
        // Allocate memory for segment tree
        //Height of segment tree
        int x = (int) (Math.ceil(Math.log(n) / Math.log(2)));

        //Maximum size of segment tree
        int max_size = 2 * (int) Math.pow(2, x) - 1;

        st = new int[max_size]; // Memory allocation

        constructSTUtil(arr, 0, n - 1, 0);
    }

    // A utility function to get the middle index from corner indexes.
    int getMid(int s, int e) {
        return s + (e - s) / 2;
    }

    /* A recursive function to get the sum of values in given range
    of the array. The following are parameters for this function.

    st --> Pointer to segment tree
    si --> Index of current node in the segment tree. Initially
    0 is passed as root is always at index 0
    ss & se --> Starting and ending indexes of the segment represented
    by current node, i.e., st[si]
    qs & qe --> Starting and ending indexes of query range */
    int getSumUtil(int ss, int se, int qs, int qe, int si)
    {
        // If segment of this node is a part of given range, then return
        // the sum of the segment
        if (qs <= ss && qe >= se)
            return st[si];

        // If segment of this node is outside the given range
        if (se < qs || ss > qe)
            return 0;

        // If a part of this segment overlaps with the given range
        int mid = getMid(ss, se);
        return getSumUtil(ss, mid, qs, qe, 2 * si + 1) +
            getSumUtil(mid + 1, se, qs, qe, 2 * si + 2);
    }

    /* A recursive function to update the nodes which have the given
    index in their range. The following are parameters
    st, si, ss and se are same as getSumUtil()
    i --> index of the element to be updated. This index is in
    input array.
    diff --> Value to be added to all nodes which have i in range */
    void updateValueUtil(int ss, int se, int i, int diff, int si)
    {
        // Base Case: If the input index lies outside the range of
        // this segment
        if (i < ss || i > se)
            return;

        // If the input index is in range of this node, then update the
        // value of the node and its children
        st[si] = st[si] + diff;
        if (se != ss) {
            int mid = getMid(ss, se);
            updateValueUtil(ss, mid, i, diff, 2 * si + 1);
            updateValueUtil(mid + 1, se, i, diff, 2 * si + 2);
        }
    }

    // The function to update a value in input array and segment tree.
    // It uses updateValueUtil() to update the value in segment tree
    void updateValue(int arr[], int n, int i, int new_val)
    {
        // Check for erroneous input index
        if (i < 0 || i > n - 1) {
            System.out.println("Invalid Input");
            return;
        }
    }
```

```

// Get the difference between new value and old value
int diff = new_val - arr[j];

// Update the value in array
arr[j] = new_val;

// Update the values of nodes in segment tree
updateValueUtil(0, n - 1, i, diff, 0);
}

// Return sum of elements in range from index qs (query start) to
// qe (query end). It mainly uses getSumUtil()
int getSum(int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n - 1 || qs > qe) {
        System.out.println("Invalid Input");
        return -1;
    }
    return getSumUtil(0, n - 1, qs, qe, 0);
}

// A recursive function that constructs Segment Tree for array[ss..se].
// si is index of current node in segment tree st
int constructSTUtil(int arr[], int ss, int se, int si)
{
    // If there is one element in array, store it in current node of
    // segment tree and return
    if (ss == se) {
        st[si] = arr[ss];
        return arr[ss];
    }

    // If there are more than one elements, then recur for left and
    // right subtrees and store the sum of values in this node
    int mid = getMid(ss, se);
    st[si] = constructSTUtil(arr, ss, mid, si * 2 + 1) +
        constructSTUtil(arr, mid + 1, se, si * 2 + 2);
    return st[si];
}

// Driver program to test above functions
public static void main(String args[])
{
    int arr[] = {1, 3, 5, 7, 9, 11};
    int n = arr.length;
    SegmentTree tree = new SegmentTree(arr, n);

    // Build segment tree from given array

    // Print sum of values in array from index 1 to 3
    System.out.println("Sum of values in given range = " +
        tree.getSum(n, 1, 3));

    // Update: set arr[1] = 10 and update corresponding segment
    // tree nodes
    tree.updateValue(arr, n, 1, 10);

    // Find sum after the value is updated
    System.out.println("Updated sum of values in given range = " +
        tree.getSum(n, 1, 3));
}
}
//This code is contributed by Ankur Narain Verma

```

Output:

```

Sum of values in given range = 15
Updated sum of values in given range = 22

```

Time Complexity:

Time Complexity for tree construction is $O(n)$. There are total $2n-1$ nodes, and value of every node is calculated only once in tree construction.

Time complexity to query is $O(\text{Logn})$. To query a sum, we process at most four nodes at every level and number of levels is $O(\text{Logn})$.

The time complexity of update is also $O(\text{Logn})$. To update a leaf value, we process one node at every level and number of levels is $O(\text{Logn})$.

Segment Tree | Set 2 (Range Minimum Query)

References:

<http://www.cse.iitk.ac.in/users/aca/top12/slides/06.pdf>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Advance Data Structures
Advanced Data Structures
Segment-Tree

Segment Tree | Set 2 (Range Minimum Query)

We have introduced [segment tree with a simple example](#) in the previous post. In this post, [Range Minimum Query](#) problem is discussed as another example where Segment Tree can be used. Following is problem statement.

We have an array $arr[0 \dots n-1]$. We should be able to efficiently find the minimum value from index qs (query start) to qe (query end) where $0 \leq qs \leq qe < n$.

A **simple solution** is to run a loop from qs to qe and find minimum element in given range. This solution takes $O(n)$ time in worst case.

Another solution is to create a 2D array where an entry $[i, j]$ stores the minimum value in range $arr[i..j]$. Minimum of a given range can now be calculated in $O(1)$ time, but preprocessing takes $O(n^2)$ time. Also, this approach needs $O(n^2)$ extra space which may become huge for large input arrays.

Segment tree can be used to do preprocessing and query in moderate time. With segment tree, preprocessing time is $O(n)$ and time to for range minimum query is $O(\text{Logn})$. The extra space required is $O(n)$ to store the segment tree.

Representation of Segment trees

1. Leaf Nodes are the elements of the input array.
2. Each internal node represents minimum of all leaves under it.

An array representation of tree is used to represent Segment Trees. For each node at index i , the left child is at index 2^{i+1} , right child at 2^{i+2} and the parent is at $\lfloor i/2 \rfloor$.



Construction of Segment Tree from given array

We start with a segment $arr[0 \dots n-1]$, and every time we divide the current segment into two halves (if it has not yet become a segment of length 1), and then call the same procedure on both halves, and for each such segment, we store the minimum value in a segment tree node.

All levels of the constructed segment tree will be completely filled except the last level. Also, the tree will be a **Full Binary Tree** because we always divide segments in two halves at every level. Since the constructed tree is always full binary tree with n leaves, there will be $n-1$ internal nodes. So total number of nodes will be $2*n - 1$.

Height of the segment tree will be . Since the tree is represented using array and relation between parent and child indexes must be maintained, size of memory allocated for segment tree will be .

Query for minimum value of given range

Once the tree is constructed, how to do range minimum query using the constructed segment tree. Following is algorithm to get the minimum.

```
// qs --> query start index, qe --> query end index
int RMQ(node, qs, qe)
{
    if range of node is within qs and qe
        return value in node
    else if range of node is completely outside qs and qe
        return INFINITE
    else
        return min( RMQ(node's left child, qs, qe), RMQ(node's right child, qs, qe) )
}
```

Implementation:

C

```
// C program for range minimum query using segment tree
#include <stdio.h>
#include <math.h>
#include <limits.h>

// A utility function to get minimum of two numbers
int minVal(int x, int y) { return (x < y)? x: y; }

// A utility function to get the middle index from corner indexes.
int getMid(int s, int e) { return s + (e - s)/2; }

/* A recursive function to get the minimum value in a given range
of array indexes. The following are parameters for this function.

st --> Pointer to segment tree
index --> Index of current node in the segment tree. Initially
0 is passed as root is always at index 0
ss & se --> Starting and ending indexes of the segment represented
by current node, i.e., st[index]
qs & qe --> Starting and ending indexes of query range */
int RMQUtil(int *st, int ss, int se, int qs, int qe, int index)
{
    // If segment of this node is a part of given range, then return
    // the min of the segment
    if (qs <= ss && qe >= se)
        return st[index];

    // If segment of this node is outside the given range
    if (se < qs || ss > qe)
        return INT_MAX;

    // If a part of this segment overlaps with the given range
    int mid = getMid(ss, se);
    return minVal(RMQUtil(st, ss, mid, qs, qe, 2*index+1),
                  RMQUtil(st, mid+1, se, qs, qe, 2*index+2));
}

// Return minimum of elements in range from index qs (query start) to
// qe (query end). It mainly uses RMQUtil()
int RMQ(int *st, int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n-1 || qs > qe)
    {
        printf("Invalid Input");
        return -1;
    }

    return RMQUtil(st, 0, n-1, qs, qe, 0);
}

// A recursive function that constructs Segment Tree for array[ss..se].
// si is index of current node in segment tree st
int constructSTUtil(int arr[], int ss, int se, int *st, int si)
{
    // If there is one element in array, store it in current node of
    // segment tree and return
    if (ss == se)
    {
        st[si] = arr[ss];
        return arr[ss];
    }

    // If there are more than one elements, then recur for left and
    // right subtrees and store the minimum of two values in this node
    int mid = getMid(ss, se);
    st[si] = minVal(constructSTUtil(arr, ss, mid, st, si*2+1),
```

```

        constructSTUtil(arr, mid+1, se, st, si*2+2));
    return st[si];
}

/* Function to construct segment tree from given array. This function
allocates memory for segment tree and calls constructSTUtil() to
fill the allocated memory */
int *constructST(int arr[], int n)
{
    // Allocate memory for segment tree

    //Height of segment tree
    int x = (int)(ceil(log2(n)));

    // Maximum size of segment tree
    int max_size = 2*(int)pow(2, x) - 1;

    int *st = new int[max_size];

    // Fill the allocated memory st
    constructSTUtil(arr, 0, n-1, st, 0);

    // Return the constructed segment tree
    return st;
}

// Driver program to test above functions
int main()
{
    int arr[] = { 1, 3, 2, 7, 9, 11};
    int n = sizeof(arr)/sizeof(arr[0]);

    // Build segment tree from given array
    int *st = constructST(arr, n);

    int qs = 1; // Starting index of query range
    int qe = 5; // Ending index of query range

    // Print minimum value in arr[qs..qe]
    printf("Minimum of values in range [%d, %d] is = %d\n",
        qs, qe, RMQ(st, n, qs, qe));

    return 0;
}

```

Java

```

// Program for range minimum query using segment tree
class SegmentTreeRMQ
{
    int st[]; //array to store segment tree

    // A utility function to get minimum of two numbers
    int minVal(int x, int y) {
        return (x < y) ? x : y;
    }

    // A utility function to get the middle index from corner
    // indexes.
    int getMid(int s, int e) {
        return s + (e - s) / 2;
    }

    /* A recursive function to get the minimum value in a given
    range of array indexes. The following are parameters for
    this function.

    st --> Pointer to segment tree
    index --> Index of current node in the segment tree. Initially
    0 is passed as root is always at index 0
    ss & se --> Starting and ending indexes of the segment
    represented by current node, i.e., st[index]
    qs & qe --> Starting and ending indexes of query range */
    int RMQUUtil(int ss, int se, int qs, int qe, int index)
    {
        // If segment of this node is a part of given range, then
        // return the min of the segment
        if (qs <= ss && qe >= se)
            return st[index];

        // If segment of this node is outside the given range
        if (se < qs || ss > qe)
            return Integer.MAX_VALUE;

        // If a part of this segment overlaps with the given range
        int mid = getMid(ss, se);
        return minVal(RMQUUtil(ss, mid, qs, qe, 2 * index + 1),
            RMQUUtil(mid + 1, se, qs, qe, 2 * index + 2));
    }

    // Return minimum of elements in range from index qs (query
    // start) to qe (query end). It mainly uses RMQUUtil()
    int RMQ(int n, int qs, int qe)
    {
        // Check for erroneous input values
        if (qs < 0 || qe > n - 1 || qs > qe) {
            System.out.println("Invalid Input");
            return -1;
        }

        return RMQUUtil(0, n - 1, qs, qe, 0);
    }

    // A recursive function that constructs Segment Tree for
    // array[ss..se]. si is index of current node in segment tree st
    int constructSTUtil(int arr[], int ss, int se, int si)
    {
        // If there is one element in array, store it in current
        // node of segment tree and return
        if (ss == se) {
            st[si] = arr[ss];
            return arr[ss];
        }
    }
}

```

```

// If there are more than one elements, then recur for left and
// right subtrees and store the minimum of two values in this node
int mid = getMid(ss, se);
st[si] = minVal(constructSTUtil(arr, ss, mid, si * 2 + 1),
               constructSTUtil(arr, mid + 1, se, si * 2 + 2));
return st[si];
}

/* Function to construct segment tree from given array. This function
allocates memory for segment tree and calls constructSTUtil() to
fill the allocated memory */
void constructST(int arr[], int n)
{
    // Allocate memory for segment tree

    //Height of segment tree
    int x = (int) (Math.ceil(Math.log(n) / Math.log(2)));

    //Maximum size of segment tree
    int max_size = 2 * (int) Math.pow(2, x) - 1;
    st = new int[max_size]; // allocate memory

    // Fill the allocated memory st
    constructSTUtil(arr, 0, n - 1, 0);
}

// Driver program to test above functions
public static void main(String args[])
{
    int arr[] = { 1, 3, 2, 7, 9, 11};
    int n = arr.length;
    SegmentTreeRMQ tree = new SegmentTreeRMQ();

    // Build segment tree from given array
    tree.constructST(arr, n);

    int qs = 1; // Starting index of query range
    int qe = 5; // Ending index of query range

    // Print minimum value in arr[qs..qe]
    System.out.println("Minimum of values in range [" + qs + ", "
                       + qe + "] is = " + tree.RMQ(n, qs, qe));
}
}
// This code is contributed by Ankur Narain Verma

```

Output:

```
Minimum of values in range [1, 5] is = 2
```

Time Complexity:

Time Complexity for tree construction is $O(n)$. There are total $2n-1$ nodes, and value of every node is calculated only once in tree construction.

Time complexity to query is $O(\log n)$. To query a range minimum, we process at most two nodes at every level and number of levels is $O(\log n)$.

Please refer following links for more solutions to range minimum query problem.

<http://www.geeksforgeeks.org/range-minimum-query-for-static-array/>

[http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor#Range_Minimum_Query_\(RMQ\)](http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor#Range_Minimum_Query_(RMQ))

http://wcipeg.com/wiki/Range_minimum_query

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
 Advanced Data Structures
 Segment-Tree

Lazy Propagation in Segment Tree

Segment tree is introduced in [previous post](#) with an example of range sum problem. We have used the same "Sum of given Range" problem to explain Lazy propagation



How does update work in Simple Segment Tree?

In the [previous post](#), update function was called to update only a single value in array. Please note that a single value update in array may cause multiple updates in Segment Tree as there may be many segment tree nodes that have a single array element in their ranges.

Below is simple logic used in previous post.

- 1) Start with root of segment tree.
- 2) If array index to be updated is not in current node's range, then return
- 3) Else update current node and recur for children.

Below is code taken from previous post.

```

/* A recursive function to update the nodes which have the given
index in their range. The following are parameters
tree[] --> segment tree
si --> index of current node in segment tree.
Initial value is passed as 0.
ss and se --> Starting and ending indexes of array elements
covered under this node of segment tree.

```

```

        Initial values passed as 0 and n-1.
i    --> index of the element to be updated. This index
      is in input array.
diff --> Value to be added to all nodes which have array
        index i in range */
void updateValueUtil(int tree[], int ss, int se, int i,
                    int diff, int si)
{
    // Base Case: If the input index lies outside the range
    // of this segment
    if (i < ss || i > se)
        return;

    // If the input index is in range of this node, then
    // update the value of the node and its children
    st[si] = st[si] + diff;
    if (se != ss)
    {
        int mid = getMid(ss, se);
        updateValueUtil(st, ss, mid, i, diff, 2*si + 1);
        updateValueUtil(st, mid+1, se, i, diff, 2*si + 2);
    }
}

```

What if there are updates on a range of array indexes?

For example add 10 to all values at indexes from 2 to 7 in array. The above update has to be called for every index from 2 to 7. We can avoid multiple calls by writing a function `updateRange()` that updates nodes accordingly.

```

/* Function to update segment tree for range update in input
array.
si -> index of current node in segment tree
ss and se -> Starting and ending indexes of elements for
        which current nodes stores sum.
us and eu -> starting and ending indexes of update query
ue -> ending index of update query
diff -> which we need to add in the range us to ue */
void updateRangeUtil(int si, int ss, int se, int us,
                    int ue, int diff)
{
    // out of range
    if (ss>se || ss>ue || se<us)
        return ;

    // Current node is a leaf node
    if (ss==se)
    {
        // Add the difference to current node
        tree[si] += diff;
        return;
    }

    // If not a leaf node, recur for children.
    int mid = (ss+se)/2;
    updateRangeUtil(si*2+1, ss, mid, us, ue, diff);
    updateRangeUtil(si*2+2, mid+1, se, us, ue, diff);

    // Use the result of children calls to update this
    // node
    tree[si] = tree[si*2+1] + tree[si*2+2];
}

```

Lazy Propagation – An optimization to make range updates faster

When there are many updates and updates are done on a range, we can postpone some updates (avoid recursive calls in update) and do those updates only when required.

Please remember that a node in segment tree stores or represents result of a query for a range of indexes. And if this node's range lies within the update operation range, then all descendants of the node must also be updated. For example consider the node with value 27 in above diagram, this node stores sum of values at indexes from 3 to 5. If our update query is for range 2 to 5, then we need to update this node and all descendants of this node. With Lazy propagation, we update only node with value 27 and postpone updates to its children by storing this update information in separate nodes called lazy nodes or values. We create an array `lazy[]` which represents lazy node. Size of `lazy[]` is same as array that represents segment tree, which is `tree[]` in below code.

The idea is to initialize all elements of `lazy[]` as 0. A value 0 in `lazy[i]` indicates that there are no pending updates on node `i` in segment tree. A non-zero value of `lazy[i]` means that this amount needs to be added to node `i` in segment tree before making any query to the node.

Below is modified update method.

```

// To update segment tree for change in array
// values at array indexes from us to ue.
updateRange(us, ue)
1) If current segment tree node has any pending
   update, then first add that pending update to
   current node.
2) If current node's range lies completely in
   update query range.
...a) Update current node
...b) Postpone updates to children by setting
    lazy value for children nodes.
3) If current node's range overlaps with update
   range, follow the same approach as above simple
   update.
...a) Recur for left and right children.
...b) Update current node using results of left
    and right calls.

```

Is there any change in Query Function also?

Since we have changed update to postpone its operations, there may be problems if a query is made to a node that is yet to be updated. So we need to update our query method also which is `getSumUtil` in [previous post](#). The `getSumUtil()` now first checks if there is a pending update and if there is, then updates the node. Once it makes sure that pending update is done, it works same as the previous `getSumUtil()`.

Below are programs to demonstrate working of Lazy Propagation.

C/C++

```

// Program to show segment tree to demonstrate lazy
// propagation
#include <stdio.h>
#include <math.h>
#define MAX 1000

// Ideally, we should not use global variables and large
// constant-sized arrays, we have done it here for simplicity.
int tree[MAX] = {0}; // To store segment tree
int lazy[MAX] = {0}; // To store pending updates

```

```

/* si -> index of current node in segment tree
ss and se -> Starting and ending indexes of elements for
which current nodes stores sum.
us and eu -> starting and ending indexes of update query
ue -> ending index of update query
diff -> which we need to add in the range us to ue */
void updateRangeUtil(int si, int ss, int se, int us,
int ue, int diff)
{
    // If lazy value is non-zero for current node of segment
    // tree, then there are some pending updates. So we need
    // to make sure that the pending updates are done before
    // making new updates. Because this value may be used by
    // parent after recursive calls (See last line of this
    // function)
    if (lazy[si] != 0)
    {
        // Make pending updates using value stored in lazy
        // nodes
        tree[si] += (se-ss+1)*lazy[si];

        // checking if it is not leaf node because if
        // it is leaf node then we cannot go further
        if (ss != se)
        {
            // We can postpone updating children we don't
            // need their new values now.
            // Since we are not yet updating children of si,
            // we need to set lazy flags for the children
            lazy[si*2 + 1] += lazy[si];
            lazy[si*2 + 2] += lazy[si];
        }

        // Set the lazy value for current node as 0 as it
        // has been updated
        lazy[si] = 0;
    }

    // out of range
    if (ss>se || ss>ue || se<us)
        return ;

    // Current segment is fully in range
    if (ss>=us && se<=ue)
    {
        // Add the difference to current node
        tree[si] += (se-ss+1)*diff;

        // same logic for checking leaf node or not
        if (ss != se)
        {
            // This is where we store values in lazy nodes,
            // rather than updating the segment tree itself
            // Since we don't need these updated values now
            // we postpone updates by storing values in lazy[]
            lazy[si*2 + 1] += diff;
            lazy[si*2 + 2] += diff;
        }
        return;
    }

    // If not completely in rang, but overlaps, recur for
    // children,
    int mid = (ss+se)/2;
    updateRangeUtil(si*2+1, ss, mid, us, ue, diff);
    updateRangeUtil(si*2+2, mid+1, se, us, ue, diff);

    // And use the result of children calls to update this
    // node
    tree[si] = tree[si*2+1] + tree[si*2+2];
}

// Function to update a range of values in segment
// tree
/* us and eu -> starting and ending indexes of update query
ue -> ending index of update query
diff -> which we need to add in the range us to ue */
void updateRange(int n, int us, int ue, int diff)
{
    updateRangeUtil(0, 0, n-1, us, ue, diff);
}

/* A recursive function to get the sum of values in given
range of the array. The following are parameters for
this function.
si -> Index of current node in the segment tree.
Initially 0 is passed as root is always at
index 0
ss & se -> Starting and ending indexes of the
segment represented by current node,
i.e., tree[si]
qs & qe -> Starting and ending indexes of query
range */
int getSumUtil(int ss, int se, int qs, int qe, int si)
{
    // If lazy flag is set for current node of segment tree,
    // then there are some pending updates. So we need to
    // make sure that the pending updates are done before
    // processing the sub sum query
    if (lazy[si] != 0)
    {
        // Make pending updates to this node. Note that this
        // node represents sum of elements in arr[ss..se] and
        // all these elements must be increased by lazy[si]
        tree[si] += (se-ss+1)*lazy[si];

        // checking if it is not leaf node because if
        // it is leaf node then we cannot go further
        if (ss != se)
        {
            // Since we are not yet updating children of si,
            // we need to set lazy values for the children
            lazy[si*2+1] += lazy[si];
            lazy[si*2+2] += lazy[si];
        }
    }
}

```

```

// unset the lazy value for current node as it has
// been updated
lazy[si] = 0;
}

// Out of range
if (ss>se || ss>qe || se<qe)
    return 0;

// At this point we are sure that pending lazy updates
// are done for current node. So we can return value
// (same as it was for query in our previous post)

// If this segment lies in range
if (ss==qs && se<=qe)
    return tree[si];

// If a part of this segment overlaps with the given
// range
int mid = (ss + se)/2;
return getSumUtil(ss, mid, qs, qe, 2*si+1) +
    getSumUtil(mid+1, se, qs, qe, 2*si+2);
}

// Return sum of elements in range from index qs (query
// start) to qe (query end). It mainly uses getSumUtil()
int getSum(int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n-1 || qs > qe)
    {
        printf("Invalid Input");
        return -1;
    }

    return getSumUtil(0, n-1, qs, qe, 0);
}

// A recursive function that constructs Segment Tree for
// array[ss..se]. si is index of current node in segment
// tree st.
void constructSTUtil(int arr[], int ss, int se, int si)
{
    // out of range as ss can never be greater than se
    if (ss > se)
        return ;

    // If there is one element in array, store it in
    // current node of segment tree and return
    if (ss == se)
    {
        tree[si] = arr[ss];
        return;
    }

    // If there are more than one elements, then recur
    // for left and right subtrees and store the sum
    // of values in this node
    int mid = (ss + se)/2;
    constructSTUtil(arr, ss, mid, si*2+1);
    constructSTUtil(arr, mid+1, se, si*2+2);

    tree[si] = tree[si*2 + 1] + tree[si*2 + 2];
}

/* Function to construct segment tree from given array.
This function allocates memory for segment tree and
calls constructSTUtil() to fill the allocated memory */
void constructST(int arr[], int n)
{
    // Fill the allocated memory st
    constructSTUtil(arr, 0, n-1, 0);
}

// Driver program to test above functions
int main()
{
    int arr[] = { 1, 3, 5, 7, 9, 11};
    int n = sizeof(arr)/sizeof(arr[0]);

    // Build segment tree from given array
    constructST(arr, n);

    // Print sum of values in array from index 1 to 3
    printf("Sum of values in given range = %d\n",
        getSum(n, 1, 3));

    // Add 10 to all nodes at indexes from 1 to 5.
    updateRange(n, 1, 5, 10);

    // Find sum after the value is updated
    printf("Updated sum of values in given range = %d\n",
        getSum(n, 1, 3));

    return 0;
}

```

Java

```

// Java program to demonstrate lazy propagation in segment tree
class LazySegmentTree
{
    final int MAX = 1000;    // Max tree size
    int tree[] = new int[MAX]; // To store segment tree
    int lazy[] = new int[MAX]; // To store pending updates

    /* si -> index of current node in segment tree
    ss and se -> Starting and ending indexes of elements for
    which current nodes stores sum.
    us and eu -> starting and ending indexes of update query
    ue -> ending index of update query

```



```

diff -> which we need to add in the range us to ue */
void updateRangeUtil(int si, int ss, int se, int us,
                    int ue, int diff)
{
    // If lazy value is non-zero for current node of segment
    // tree, then there are some pending updates. So we need
    // to make sure that the pending updates are done before
    // making new updates. Because this value may be used by
    // parent after recursive calls (See last line of this
    // function)
    // function)
    if (lazy[si] != 0)
    {
        // Make pending updates using value stored in lazy
        // nodes
        tree[si] += (se - ss + 1) * lazy[si];

        // checking if it is not leaf node because if
        // it is leaf node then we cannot go further
        if (ss != se)
        {
            // We can postpone updating children we don't
            // need their new values now.
            // Since we are not yet updating children of si,
            // we need to set lazy flags for the children
            lazy[si * 2 + 1] += lazy[si];
            lazy[si * 2 + 2] += lazy[si];
        }

        // Set the lazy value for current node as 0 as it
        // has been updated
        lazy[si] = 0;
    }

    // out of range
    if (ss > se || ss > ue || se < us)
        return;

    // Current segment is fully in range
    if (ss >= us && se <= ue)
    {
        // Add the difference to current node
        tree[si] += (se - ss + 1) * diff;

        // same logic for checking leaf node or not
        if (ss != se)
        {
            // This is where we store values in lazy nodes,
            // rather than updating the segment tree itself
            // Since we don't need these updated values now
            // we postpone updates by storing values in lazy[]
            lazy[si * 2 + 1] += diff;
            lazy[si * 2 + 2] += diff;
        }
        return;
    }

    // If not completely in range, but overlaps, recur for
    // children,
    int mid = (ss + se) / 2;
    updateRangeUtil(si * 2 + 1, ss, mid, us, ue, diff);
    updateRangeUtil(si * 2 + 2, mid + 1, se, us, ue, diff);

    // And use the result of children calls to update this
    // node
    tree[si] = tree[si * 2 + 1] + tree[si * 2 + 2];
}

// Function to update a range of values in segment
// tree
/* us and eu -> starting and ending indexes of update query
   ue -> ending index of update query
   diff -> which we need to add in the range us to ue */
void updateRange(int n, int us, int ue, int diff) {
    updateRangeUtil(0, 0, n - 1, us, ue, diff);
}

/* A recursive function to get the sum of values in given
range of the array. The following are parameters for
this function.
si -> Index of current node in the segment tree.
Initially 0 is passed as root is always at
index 0
ss & se -> Starting and ending indexes of the
segment represented by current node,
i.e., tree[si]
qs & qe -> Starting and ending indexes of query
range */
int getSumUtil(int ss, int se, int qs, int qe, int si)
{
    // If lazy flag is set for current node of segment tree,
    // then there are some pending updates. So we need to
    // make sure that the pending updates are done before
    // processing the sub sum query
    if (lazy[si] != 0)
    {
        // Make pending updates to this node. Note that this
        // node represents sum of elements in arr[ss..se] and
        // all these elements must be increased by lazy[si]
        tree[si] += (se - ss + 1) * lazy[si];

        // checking if it is not leaf node because if
        // it is leaf node then we cannot go further
        if (ss != se)
        {
            // Since we are not yet updating children of si,
            // we need to set lazy values for the children
            lazy[si * 2 + 1] += lazy[si];
            lazy[si * 2 + 2] += lazy[si];
        }

        // unset the lazy value for current node as it has
        // been updated
        lazy[si] = 0;
    }

    // Out of range

```

```

if (ss > se || ss > qe || se < qs)
    return 0;

// At this point sure, pending lazy updates are done
// for current node. So we can return value (same as
// was for query in our previous post)

// If this segment lies in range
if (ss >= qs && se <= qe)
    return tree[si];

// If a part of this segment overlaps with the given
// range
int mid = (ss + se) / 2;
return getSumUtil(ss, mid, qs, qe, 2 * si + 1) +
    getSumUtil(mid + 1, se, qs, qe, 2 * si + 2);
}

// Return sum of elements in range from index qs (query
// start) to qe (query end). It mainly uses getSumUtil()
int getSum(int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n - 1 || qs > qe)
    {
        System.out.println("Invalid Input");
        return -1;
    }

    return getSumUtil(0, n - 1, qs, qe, 0);
}

/* A recursive function that constructs Segment Tree for
array[ss..se]. si is index of current node in segment
tree st. */
void constructSTUtil(int arr[], int ss, int se, int si)
{
    // out of range as ss can never be greater than se
    if (ss > se)
        return;

    /* If there is one element in array, store it in
    current node of segment tree and return */
    if (ss == se)
    {
        tree[si] = arr[ss];
        return;
    }

    /* If there are more than one elements, then recur
    for left and right subtrees and store the sum
    of values in this node */
    int mid = (ss + se) / 2;
    constructSTUtil(arr, ss, mid, si * 2 + 1);
    constructSTUtil(arr, mid + 1, se, si * 2 + 2);

    tree[si] = tree[si * 2 + 1] + tree[si * 2 + 2];
}

/* Function to construct segment tree from given array.
This function allocates memory for segment tree and
calls constructSTUtil() to fill the allocated memory */
void constructST(int arr[], int n)
{
    // Fill the allocated memory st
    constructSTUtil(arr, 0, n - 1, 0);
}

// Driver program to test above functions
public static void main(String args[])
{
    int arr[] = { 1, 3, 5, 7, 9, 11 };
    int n = arr.length;
    LazySegmentTree tree = new LazySegmentTree();

    // Build segment tree from given array
    tree.constructST(arr, n);

    // Print sum of values in array from index 1 to 3
    System.out.println("Sum of values in given range = " +
        tree.getSum(n, 1, 3));

    // Add 10 to all nodes at indexes from 1 to 5.
    tree.updateRange(n, 1, 5, 10);

    // Find sum after the value is updated
    System.out.println("Updated sum of values in given range = " +
        tree.getSum(n, 1, 3));
}
// This Code is contributed by Ankur Narain Verma

```

Output:

```

Sum of values in given range = 15
Updated sum of values in given range = 45

```

This article is contributed by **Ankit Mittal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Advanced Data Structures
Segment-Tree

Persistent Segment Tree | Set 1 (Introduction)

Prerequisite : [Segment Tree](#)
[Persistency in Data Structure](#)

Segment Tree is itself a great data structure that comes into play in many cases , In this post we will introduce the concept of Persistency in this data structure. Persistency, simply means to retain the changes. But

obviously, retaining the changes cause extra memory consumption and hence affect the Time Complexity.

Our aim is to apply persistency in segment tree and also to ensure that it does not take more than **$O(\log n)$ time and space** for each change.

Let's think in terms of versions i.e. for each change in our segment tree we create a new version of it.

We will consider our initial version to be Version-0. Now, as we do any update in the segment tree we will create a new version for it and in similar fashion track the record for all versions.

But creating the whole tree for every version will take $O(n \log n)$ extra space and $O(n \log n)$ time. So, this idea runs out of time and memory for large number of versions.

Let's exploit the fact that for each new update(say point update for simplicity) in segment tree, At max $\log n$ nodes will be modified. So, our new version will only contain these $\log n$ new nodes and rest nodes will be the same as previous version. Therefore, it is quite clear that for each new version we only need to create these $\log n$ new nodes whereas the rest of nodes can be shared from the previous version.

Consider the below figure for better visualization(click on the image for better view) :-

[persistent segtree](#)



Consider the segment tree with green nodes . Lets call this segment tree as **version-0**. The left child for each node is connected with solid red edge where as the right child for each node is connected with solid purple edge. Clearly, this segment tree consists of 15 nodes.

Now consider we need to make change in the leaf node 13 of version-0.

So, the affected nodes will be – **node 13 , node 6 , node 3 , node 1**.

Therefore, for the new version (**Version-1**) we need to create only these **4 new nodes**.

Now, lets construct version-1 for this change in segment tree. We need a new node 1 as it is affected by change done in node 13. So , we will first create a new **node 1'**(yellow color) . The left child for node 1' will be the same for left child for node 1 in version-0. So, we connect the left child of node 1' with node 2 of version-0(red dashed line in figure). Let's now examine the right child for node 1' in version-1. We need to create a new node as it is affected . So we create a new node called node 3' and make it the right child for node 1'(solid purple edge connection).

In the similar fashion we will now examine for **node 3'**. The left child is affected , So we create a new node called **node 6'** and connect it with solid red edge with node 3' , where as the right child for node 3' will be the same as right child of node 3 in version-0. So, we will make the right child of node 3 in version-0 as the right child of node 3' in version-1(see the purple dash edge.)

Same procedure is done for node 6' and we see that the left child of node 6' will be the left child of node 6 in version-0(red dashed connection) and right child is newly created node called **node 13'**(solid purple dashed edge).

Each **yellow color node** is a newly created node and dashed edges are the inter-connection between the different versions of the segment tree.

Now, the Question arises : **How to keep track of all the versions?**

– We only need to keep track the first root node for all the versions and this will serve the purpose to track all the newly created nodes in the different versions. For this purpose we can maintain an array of pointers to the first node of segment trees for all versions.

Let's consider a very basic problem to see how to implement persistence in segment tree

Problem : Given an array $A[]$ and different point update operations.Considering each point operation to create a new version of the array. We need to answer the queries of type
Q v l r : output the sum of elements in range l to r just after the v-th update.

We will create all the versions of the segment tree and keep track of their root node.Then for each range sum query we will pass the required version's root node in our query function and output the required sum.

Below is the C++ implementation for the above problem:-

```
// C++ program to implement persistent segment
// tree.
#include "bits/stdc++.h"
using namespace std;

#define MAXN 100

/* data type for individual
 * node in the segment tree */
struct node
{
    // stores sum of the elements in node
    int val;

    // pointer to left and right children
    node* left, *right;

    // required constructors.....
    node() {}
    node(node* l, node* r, int v)
    {
        left = l;
        right = r;
        val = v;
    }
};
```

```

// input array
int arr[MAXN];

// root pointers for all versions
node* version[MAXN];

// Constructs Version-0
// Time Complexity : O(nlogn)
void build(node* n,int low,int high)
{
    if (low==high)
    {
        n->val = arr[low];
        return;
    }
    int mid = (low+high) / 2;
    n->left = new node(NULL, NULL, 0);
    n->right = new node(NULL, NULL, 0);
    build(n->left, low, mid);
    build(n->right, mid+1, high);
    n->val = n->left->val + n->right->val;
}

/**
 * Upgrades to new Version
 * @param prev : points to node of previous version
 * @param cur : points to node of current version
 * Time Complexity : O(logn)
 * Space Complexity : O(logn) */
void upgrade(node* prev, node* cur, int low, int high,
             int idx, int value)
{
    if (idx > high or idx < low or low > high)
        return;

    if (low == high)
    {
        // modification in new version
        cur->val = value;
        return;
    }
    int mid = (low+high) / 2;
    if (idx <= mid)
    {
        // link to right child of previous version
        cur->right = prev->right;

        // create new node in current version
        cur->left = new node(NULL, NULL, 0);

        upgrade(prev->left,cur->left, low, mid, idx, value);
    }
    else
    {
        // link to left child of previous version
        cur->left = prev->left;

        // create new node for current version
        cur->right = new node(NULL, NULL, 0);

        upgrade(prev->right, cur->right, mid+1, high, idx, value);
    }

    // calculating data for current version
    // by combining previous version and current
    // modification
    cur->val = cur->left->val + cur->right->val;
}

int query(node* n, int low, int high, int l, int r)
{
    if (l > high or r < low or low > high)
        return 0;
    if (l <= low and high <= r)
        return n->val;
    int mid = (low+high) / 2;
    int p1 = query(n->left,low,mid,l,r);
    int p2 = query(n->right,mid+1,high,l,r);
    return p1+p2;
}

int main(int argc, char const *argv[])
{
    int A[] = {1,2,3,4,5};
    int n = sizeof(A)/sizeof(int);

    for (int i=0; i<n; i++)
        arr[i] = A[i];

    // creating Version-0
    node* root = new node(NULL, NULL, 0);
    build(root, 0, n-1);

    // storing root node for version-0
    version[0] = root;

    // upgrading to version-1
    version[1] = new node(NULL, NULL, 0);
    upgrade(version[0], version[1], 0, n-1, 4, 1);

    // upgrading to version-2
    version[2] = new node(NULL, NULL, 0);
    upgrade(version[1],version[2], 0, n-1, 2, 10);

    cout << "In version 1 , query(0,4) : ";
    cout << query(version[1], 0, n-1, 0, 4) << endl;

    cout << "In version 2 , query(3,4) : ";
    cout << query(version[2], 0, n-1, 3, 4) << endl;

    cout << "In version 0 , query(0,3) : ";
    cout << query(version[0], 0, n-1, 0, 3) << endl;
    return 0;
}

```

Output:

```
In version 1 , query(0,4) : 11
In version 2 , query(3,4) : 5
In version 0 , query(0,3) : 10
```

Note : The above problem can also be solved by processing the queries offline by sorting it with respect to the version and answering the queries just after the corresponding update.

Time Complexity : The time complexity will be the same as the query and point update operation in the segment tree as we can consider the extra node creation step to be done in $O(1)$. Hence, the overall Time Complexity per query for new version creation and range sum query will be $O(\log n)$.

This article is contributed by **Nitish Kumar**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Trees
Segment-Tree

Trie | (Insert and Search)

Trie is an efficient information *retrieval* data structure. Using trie, search complexities can be brought to optimal limit (key length). If we store keys in binary search tree, a well balanced BST will need time proportional to $M \cdot \log N$, where M is maximum string length and N is number of keys in tree. Using trie, we can search the key in $O(M)$ time. However the penalty is on trie storage requirements.

Every node of trie consists of multiple branches. Each branch represents a possible character of keys. We need to mark the last node of every key as leaf node. A trie node field *value* will be used to distinguish the node as leaf node (there are other uses of the *value* field). A simple structure to represent nodes of English alphabet can be as following.

```
// Trie node
struct TrieNode
{
    struct TrieNode *children[ALPHABET_SIZE];

    // isLeaf is true if the node represents
    // end of a word
    bool isLeaf;
};
```

Inserting a key into trie is simple approach. Every character of input key is inserted as an individual trie node. Note that the *children* is an array of pointers to next level trie nodes. The key character acts as an index into the array *children*. If the input key is new or an extension of existing key, we need to construct non-existing nodes of the key, and mark leaf node. If the input key is prefix of existing key in trie, we simply mark the last node of key as leaf. The key length determines trie depth.

Searching for a key is similar to insert operation, however we only compare the characters and move down. The search can terminate due to end of string or lack of key in trie. In the former case, if the *value* field of last node is non-zero then the key exists in trie. In the second case, the search terminates without examining all the characters of key, since the key is not present in trie.

The following picture explains construction of trie using keys given in the example below.

```
      root
     /  |  \
    t  a  b
    |  |  |
    h  n  y
    |  |  |
    e  s  e
    /  |  |
   i  r  w
    |  |  |
   r  e  e
    |
    r
```

In the picture, every character is of type *trie_node_t*. For example, the *root* is of type *trie_node_t*, and it's children *a*, *b* and *t* are filled, all other nodes of root will be NULL. Similarly, "a" at the next level is having only one child ("n"), all other children are NULL. The leaf nodes are in blue.

Insert and search costs $O(\text{key_length})$, however the memory requirements of trie is $O(\text{ALPHABET_SIZE} \cdot \text{key_length} \cdot N)$ where N is number of keys in trie. There are efficient representation of trie nodes (e.g. compressed trie, *ternary search tree*, etc.) to minimize memory requirements of trie.

```
// C implementation of search and insert operations
// on Trie
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

#define ARRAY_SIZE(a) sizeof(a)/sizeof(a[0])

// Alphabet size (# of symbols)
#define ALPHABET_SIZE (26)

// Converts key current character into index
// use only 'a' through 'z' and lower case
#define CHAR_TO_INDEX(c) ((int)c - (int)'a')

// trie node
struct TrieNode
{
    struct TrieNode *children[ALPHABET_SIZE];

    // isLeaf is true if the node represents
    // end of a word
    bool isLeaf;
};

// Returns new trie node (initialized to NULLs)
struct TrieNode *getNode(void)
{
    struct TrieNode *pNode = NULL;

    pNode = (struct TrieNode *) malloc(sizeof(struct TrieNode));

    if (pNode)
    {
        int i;

        pNode->isLeaf = false;

        for (i = 0; i < ALPHABET_SIZE; i++)
```

```

        pNode->children[i] = NULL;
    }

    return pNode;
}

// If not present, inserts key into trie
// If the key is prefix of trie node, just marks leaf node
void insert(struct TrieNode *root, const char *key)
{
    int level;
    int length = strlen(key);
    int index;

    struct TrieNode *pCrawl = root;

    for (level = 0; level < length; level++)
    {
        index = CHAR_TO_INDEX(key[level]);
        if (!pCrawl->children[index])
            pCrawl->children[index] = getNode();

        pCrawl = pCrawl->children[index];
    }

    // mark last node as leaf
    pCrawl->isLeaf = true;
}

// Returns true if key presents in trie, else false
bool search(struct TrieNode *root, const char *key)
{
    int level;
    int length = strlen(key);
    int index;
    struct TrieNode *pCrawl = root;

    for (level = 0; level < length; level++)
    {
        index = CHAR_TO_INDEX(key[level]);

        if (!pCrawl->children[index])
            return false;

        pCrawl = pCrawl->children[index];
    }

    return (pCrawl != NULL && pCrawl->isLeaf);
}

// Driver
int main()
{
    // Input keys (use only 'a' through 'z' and lower case)
    char keys[][8] = {"the", "a", "there", "answer", "any",
                     "by", "bye", "their"};

    char output[][32] = {"Not present in trie", "Present in trie"};

    struct TrieNode *root = getNode();

    // Construct trie
    int i;
    for (i = 0; i < ARRAY_SIZE(keys); i++)
        insert(root, keys[i]);

    // Search for different keys
    printf("%s --- %s\n", "the", output[search(root, "the")]);
    printf("%s --- %s\n", "these", output[search(root, "these")]);
    printf("%s --- %s\n", "their", output[search(root, "their")]);
    printf("%s --- %s\n", "thaw", output[search(root, "thaw")]);

    return 0;
}

```

Output :

```

the --- Present in trie
these --- Not present in trie
their --- Present in trie
thaw --- Not present in trie

```

Next Article [Trie Delete](#)

This article is contributed by [Venki](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Trie | (Delete)

In the [previous post](#) on [trie](#) we have described how to insert and search a node in trie. Here is an algorithm how to delete a node from trie.

During delete operation we delete the key in bottom up manner using recursion. The following are possible conditions when deleting key from trie,

1. Key may not be there in trie. Delete operation should not modify trie.
2. Key present as unique key (no part of key contains another key (prefix), nor the key itself is prefix of another key in trie). Delete all the nodes.
3. Key is prefix key of another long key in trie. Unmark the leaf node.
4. Key present in trie, having atleast one other key as prefix key. Delete nodes from end of key until first leaf node of longest prefix key.

The highlighted code presents algorithm to implement above conditions. (One may be in dilemma how a pointer passed to delete helper is reflecting changes from deleteHelper to deleteKey. Note that we are holding trie as an ADT in trie_t node, which is passed by reference or pointer).

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define ARRAY_SIZE(a) sizeof(a)/sizeof(a[0])

// Alphabet size (# of symbols)

#define ALPHABET_SIZE (26)
#define INDEX(c) ((int)c - (int)'a')

#define FREE(p) \
    free(p); \
    p = NULL;

// forward declration
typedef struct trie_node trie_node_t;

// trie node
struct trie_node
{
    int value; // non zero if leaf
    trie_node_t *children[ALPHABET_SIZE];
};

// trie ADT
typedef struct trie trie_t;

struct trie
{
    trie_node_t *root;
    int count;
};

trie_node_t *getNode(void)
{
    trie_node_t *pNode = NULL;

    pNode = (trie_node_t *)malloc(sizeof(trie_node_t));

    if( pNode )
    {
        int i;

        pNode->value = 0;

        for(i = 0; i < ALPHABET_SIZE; i++)
        {
            pNode->children[i] = NULL;
        }
    }

    return pNode;
}

void initialize(trie_t *pTrie)
{
    pTrie->root = getNode();
    pTrie->count = 0;
}

void insert(trie_t *pTrie, char key[])
{
    int level;
    int length = strlen(key);
    int index;
    trie_node_t *pCrawl;

    pTrie->count++;
    pCrawl = pTrie->root;

    for( level = 0; level < length; level++ )
    {
        index = INDEX(key[level]);

        if( pCrawl->children[index] )
        {
            // Skip current node
            pCrawl = pCrawl->children[index];
        }
        else
        {
            // Add new node
            pCrawl->children[index] = getNode();
            pCrawl = pCrawl->children[index];
        }
    }

    // mark last node as leaf (non zero)
    pCrawl->value = pTrie->count;
}

int search(trie_t *pTrie, char key[])
{
    int level;
    int length = strlen(key);
    int index;
    trie_node_t *pCrawl;
```

```

pCrawl = pTrie->root;

for( level = 0; level < length; level++ )
{
    index = INDEX(key[level]);

    if( !pCrawl->children[index] )
    {
        return 0;
    }

    pCrawl = pCrawl->children[index];
}

return ( 0 != pCrawl && pCrawl->value );
}

int leafNode(trie_node_t *pNode)
{
    return ( pNode->value != 0 );
}

int isFreeNode(trie_node_t *pNode)
{
    int i;
    for( i = 0; i < ALPHABET_SIZE; i++ )
    {
        if( pNode->children[i] )
            return 0;
    }

    return 1;
}

bool deleteHelper(trie_node_t *pNode, char key[], int level, int len)
{
    if( pNode )
    {
        // Base case
        if( level == len )
        {
            if( pNode->value )
            {
                // Unmark leaf node
                pNode->value = 0;

                // If empty, node to be deleted
                if( isFreeNode(pNode) )
                {
                    return true;
                }

                return false;
            }
        }
        else // Recursive case
        {
            int index = INDEX(key[level]);

            if( deleteHelper(pNode->children[index], key, level+1, len) )
            {
                // last node marked, delete it
                FREE(pNode->children[index]);

                // recursively climb up, and delete eligible nodes
                return ( !leafNode(pNode) && isFreeNode(pNode) );
            }
        }
    }

    return false;
}

void deleteKey(trie_t *pTrie, char key[])
{
    int len = strlen(key);

    if( len > 0 )
    {
        deleteHelper(pTrie->root, key, 0, len);
    }
}

int main()
{
    char keys[][8] = {"she", "sells", "sea", "shore", "the", "by", "sheer"};
    trie_t trie;

    initialize(&trie);

    for( int i = 0; i < ARRAY_SIZE(keys); i++ )
    {
        insert(&trie, keys[i]);
    }

    deleteKey(&trie, keys[0]);

    printf("%s %s\n", "she", search(&trie, "she") ? "Present in trie" : "Not present in trie");

    return 0;
}

```


— Venki. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Advance Data Structures
TRIE
About Venki
Software Engineer
[View all posts by Venki](#) →

Longest prefix matching – A Trie based solution in Java

Given a dictionary of words and an input string, find the longest prefix of the string which is also a word in dictionary.

Examples:

Let the dictionary contains the following words:
{are, area, base, cat, cater, children, basement}

Below are some input/output examples:

| Input String | Output |
|--------------|-----------|
| caterer | cater |
| basemexy | base |
| child | < Empty > |

Solution

We build a Trie of all dictionary words. Once the Trie is built, traverse through it using characters of input string. If prefix matches a dictionary word, store current length and look for a longer match. Finally, return the longest match.

Following is Java implementation of the above solution based.

```
import java.util.HashMap;

// Trie Node, which stores a character and the children in a HashMap
class TrieNode {
    public TrieNode(char ch) {
        value = ch;
        children = new HashMap<>();
        blsEnd = false;
    }
    public HashMap<Character, TrieNode> getChildren() { return children; }
    public char getValue() { return value; }
    public void setlsEnd(boolean val) { blsEnd = val; }
    public boolean isEnd() { return blsEnd; }

    private char value;
    private HashMap<Character, TrieNode> children;
    private boolean blsEnd;
}

// Implements the actual Trie
class Trie {
    // Constructor
    public Trie() { root = new TrieNode((char)0); }

    // Method to insert a new word to Trie
    public void insert(String word) {

        // Find length of the given word
        int length = word.length();
        TrieNode crawl = root;

        // Traverse through all characters of given word
        for( int level = 0; level < length; level++)
        {
            HashMap<Character, TrieNode> child = crawl.getChildren();
            char ch = word.charAt(level);

            // If there is already a child for current character of given word
            if( child.containsKey(ch))
                crawl = child.get(ch);
            else // Else create a child
            {
                TrieNode temp = new TrieNode(ch);
                child.put( ch, temp );
                crawl = temp;
            }
        }

        // Set blsEnd true for last character
        crawl.setlsEnd(true);
    }

    // The main method that finds out the longest string 'input'
    public String getMatchingPrefix(String input) {
        String result = ""; // Initialize resultant string
        int length = input.length(); // Find length of the input string

        // Initialize reference to traverse through Trie
        TrieNode crawl = root;
```

```

// Iterate through all characters of input string 'str' and traverse
// down the Trie
int level, prevMatch = 0;
for( level = 0 ; level < length; level++ )
{
    // Find current character of str
    char ch = input.charAt(level);

    // HashMap of current Trie node to traverse down
    HashMap<Character,TrieNode> child = crawl.getChildren();

    // See if there is a Trie edge for the current character
    if( child.containsKey(ch) )
    {
        result += ch; //Update result
        crawl = child.get(ch); //Update crawl to move down in Trie

        // If this is end of a word, then update prevMatch
        if( crawl.isEnd() )
            prevMatch = level + 1;
    }
    else break;
}

// If the last processed character did not match end of a word,
// return the previously matching prefix
if( !crawl.isEnd() )
    return result.substring(0, prevMatch);

else return result;
}

private TrieNode root;
}

// Testing class
public class Test {
    public static void main(String[] args) {
        Trie dict = new Trie();
        dict.insert("are");
        dict.insert("area");
        dict.insert("base");
        dict.insert("cat");
        dict.insert("cater");
        dict.insert("basement");

        String input = "caterer";
        System.out.print(input + ": ");
        System.out.println(dict.getMatchingPrefix(input));

        input = "basement";
        System.out.print(input + ": ");
        System.out.println(dict.getMatchingPrefix(input));

        input = "are";
        System.out.print(input + ": ");
        System.out.println(dict.getMatchingPrefix(input));

        input = "arex";
        System.out.print(input + ": ");
        System.out.println(dict.getMatchingPrefix(input));

        input = "basemexz";
        System.out.print(input + ": ");
        System.out.println(dict.getMatchingPrefix(input));

        input = "xyz";
        System.out.print(input + ": ");
        System.out.println(dict.getMatchingPrefix(input));
    }
}

```

Output:

```

caterer: cater
basement: basement
are: are
arex: are
basemexz: base
xyz:

```

Time Complexity: Time complexity of finding the longest prefix is $O(n)$ where n is length of the input string. Refer [this](#) for time complexity of building the Trie.

This article is compiled by **Ravi Chandra Enaganti**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Advance Data Structures
Java
TRIE

Print unique rows in a given boolean matrix

Given a binary matrix, print all unique rows of the given matrix.

```

Input:
{0, 1, 0, 0, 1}
{1, 0, 1, 1, 0}
{0, 1, 0, 0, 1}
{1, 1, 1, 0, 0}

Output:
0 1 0 0 1
1 0 1 1 0
1 1 1 0 0

```

Method 1 (Simple)

A simple approach is to check each row with all processed rows. Print the first row. Now, starting from the second row, for each row, compare the row with already processed rows. If the row matches with any of the processed rows, don't print it. If the current row doesn't match with any row, print it.

Time complexity: $O(\text{ROW}^2 \times \text{COL})$

Auxiliary Space: $O(1)$

Method 2 (Use Binary Search Tree)

Find the decimal equivalent of each row and insert it into BST. Each node of the BST will contain two fields, one field for the decimal value, other for row number. Do not insert a node if it is duplicated. Finally, traverse the BST and print the corresponding rows.

Time complexity: $O(\text{ROW} \times \text{COL} + \text{ROW} \times \log(\text{ROW}))$

Auxiliary Space: $O(\text{ROW})$

This method will lead to Integer Overflow if number of columns is large.

Method 3 (Use Trie data structure)

Since the matrix is boolean, a variant of Trie data structure can be used where each node will be having two children one for 0 and other for 1. Insert each row in the Trie. If the row is already there, don't print the row. If row is not there in Trie, insert it in Trie and print it.

Below is C implementation of method 3.

```
// Given a binary matrix of M X N of integers, you need to return only unique rows of binary array
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define ROW 4
#define COL 5

// A Trie node
typedef struct Node
{
    bool isEndOfCol;
    struct Node *child[2]; // Only two children needed for 0 and 1
} Node;

// A utility function to allocate memory for a new Trie node
Node* newNode()
{
    Node* temp = (Node*)malloc(sizeof(Node));
    temp->isEndOfCol = 0;
    temp->child[0] = temp->child[1] = NULL;
    return temp;
}

// Inserts a new matrix row to Trie. If row is already
// present, then returns 0, otherwise inserts the row and
// return 1
bool insert(Node** root, int (*M)[COL], int row, int col)
{
    // base case
    if (*root == NULL)
        *root = newNode();

    // Recur if there are more entries in this row
    if (col < COL)
        return insert(&(*root->child[M[row][col]]), M, row, col+1);

    else // If all entries of this row are processed
    {
        // unique row found, return 1
        if (!(*root->isEndOfCol))
            return (*root->isEndOfCol = 1);

        // duplicate row found, return 0
        return 0;
    }
}

// A utility function to print a row
void printRow(int (*M)[COL], int row)
{
    int i;
    for (i = 0; i < COL; ++i)
        printf("%d ", M[row][i]);
    printf("\n");
}

// The main function that prints all unique rows in a
// given matrix.
void findUniqueRows(int (*M)[COL])
{
    Node* root = NULL; // create an empty Trie
    int i;

    // Iterate through all rows
    for (i = 0; i < ROW; ++i)
        // insert row to TRIE
        if (insert(&root, M, i, 0))
            // unique row found, print it
            printRow(M, i);
}

// Driver program to test above functions
int main()
{
    int M[ROW][COL] = {{0, 1, 0, 0, 1},
                       {1, 0, 1, 1, 0},
                       {0, 1, 0, 0, 1},
                       {1, 0, 1, 0, 0}};

    findUniqueRows(M);

    return 0;
}
```

Time complexity: $O(\text{ROW} \times \text{COL})$

Auxiliary Space: $O(\text{ROW} \times \text{COL})$

This method has better time complexity. Also, relative order of rows is maintained while printing.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Matrix
Advance Data Structures
Advanced Data Structures

How to Implement Reverse DNS Look Up Cache?

Reverse DNS look up is using an internet IP address to find a domain name. For example, if you type 74.125.200.106 in browser, it automatically redirects to google.in.

How to implement Reverse DNS Look Up cache? Following are the operations needed from cache.

- 1) Add a IP address to URL Mapping in cache.
- 2) Find URL for a given IP address.

One solution is to use [Hashing](#).

In this post, a [Trie](#) based solution is discussed. One advantage of Trie based solutions is, worst case upper bound is $O(1)$ for Trie, for hashing, the best possible average case time complexity is $O(1)$. Also, with Trie we can implement prefix search (finding all urls for a common prefix of IP addresses).

The general disadvantage of Trie is large amount of memory requirement, this is not a major problem here as the alphabet size is only 11 here. Ten characters are needed for digits from '0' to '9' and one for dot ('.').

The idea is to store IP addresses in Trie nodes and in the last node we store the corresponding domain name. Following is C style implementation in C++.

```
// C based program to implement reverse DNS lookup
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

// There are almost 11 different chars in a valid IP address
#define CHARS 11

// Maximum length of a valid IP address
#define MAX 50

// A utility function to find index of child for a given character 'c'
int getIndex(char c) { return (c == '.')? 10: (c - '0'); }

// A utility function to find character for a given child index.
char getCharFromIndex(int i) { return (i== 10)? '.' : ('0' + i); }

// Trie Node.
struct trieNode
{
    bool isLeaf;
    char *URL;
    struct trieNode *child[CHARS];
};

// Function to create a new trie node.
struct trieNode *newTrieNode(void)
{
    struct trieNode *newNode = new trieNode;
    newNode->isLeaf = false;
    newNode->URL = NULL;
    for (int i=0; i<CHARS; i++)
        newNode->child[i] = NULL;
    return newNode;
}

// This method inserts an ip address and the corresponding
// domain name in the trie. The last node in Trie contains the URL.
void insert(struct trieNode *root, char *ipAdd, char *URL)
{
    // Length of the ip address
    int len = strlen(ipAdd);
    struct trieNode *pCrawl = root;

    // Traversing over the length of the ip address.
    for (int level=0; level<len; level++)
    {
        // Get index of child node from current character
        // in ipAdd[]. Index must be from 0 to 10 where
        // 0 to 9 is used for digits and 10 for dot
        int index = getIndex(ipAdd[level]);

        // Create a new child if not exist already
        if (!pCrawl->child[index])
            pCrawl->child[index] = newTrieNode();

        // Move to the child
        pCrawl = pCrawl->child[index];
    }

    //Below needs to be carried out for the last node.
    //Save the corresponding URL of the ip address in the
    //last node of trie.
    pCrawl->isLeaf = true;
    pCrawl->URL = new char[strlen(URL) + 1];
    strcpy(pCrawl->URL, URL);
}
```

```
// This function returns URL if given IP address is present in DNS cache.
// Else returns NULL
char *searchDNSCache(struct trieNode *root, char *ipAdd)
{
    // Root node of trie.
    struct trieNode *pCrawl = root;
    int len = strlen(ipAdd);

    // Traversal over the length of ip address.
    for (int level=0; level<len; level++)
    {
        int index = getIndex(ipAdd[level]);
        if (!pCrawl->child[index])
            return NULL;
        pCrawl = pCrawl->child[index];
    }

    // If we find the last node for a given ip address, print the URL.
    if (pCrawl!=NULL && pCrawl->isLeaf)
        return pCrawl->URL;

    return NULL;
}

//Driver function.
int main()
{
    /* Change third ipAddress for validation */
    char ipAdd[MAX] = {"107.108.11.123", "107.109.123.255",
        "74.125.200.106"};
    char URL[50] = {"www.samsung.com", "www.samsung.net",
        "www.google.in"};
    int n = sizeof(ipAdd)/sizeof(ipAdd[0]);
    struct trieNode *root = newTrieNode();

    // Inserts all the ip address and their corresponding
    // domain name after ip address validation.
    for (int i=0; i<n; i++)
        insert(root, ipAdd[i], URL[i]);

    // If reverse DNS look up succeeds print the domain
    // name along with DNS resolved.
    char ip[] = "107.108.11.123";
    char *res_url = searchDNSCache(root, ip);
    if (res_url != NULL)
        printf("Reverse DNS look up resolved in cache:\n%s --> %s",
            ip, res_url);
    else
        printf("Reverse DNS look up not resolved in cache ");
    return 0;
}
```

Output:

```
Reverse DNS look up resolved in cache:
107.108.11.123 --> www.samsung.com
```

Note that the above implementation of Trie assumes that the given IP address does not contain characters other than {'0', '1', ..., '9', '.'}. What if a user gives an invalid IP address that contains some other characters? This problem can be resolved by [validating the input IP address](#) before inserting it into Trie. We can use the approach discussed [here](#) for IP address validation.

This article is contributed by **Kumar Gautam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Advance Data Structures
Advanced Data Structures

How to Implement Forward DNS Look Up Cache?

We have discussed [implementation of Reverse DNS Look Up Cache](#). Forward DNS look up is getting IP address for a given domain name typed in the web browser.

The cache should do the following operations :

1. Add a mapping from URL to IP address
2. Find IP address for a given URL

There are a few changes from [reverse DNS look up cache](#) that we need to incorporate.

1. Instead of [0-9] and (.) dot we need to take care of [A-Z], [a-z] and (.) dot. As most of the domain name contains only lowercase characters we can assume that there will be [a-z] and (.) 27 children for each trie node.
2. When we type [www.google.in](#) and [google.in](#) the browser takes us to the same page. So, we need to add a domain name into trie for the words after [www\(.\)](#). Similarly while searching for a domain name corresponding IP address remove the [www\(.\)](#) if the user has provided it.

This is left as an exercise and for simplicity we have taken care of [www.](#) also.

One solution is to use [Hashing](#). In this post, a [Trie](#) based solution is discussed. One advantage of Trie based solutions is, worst case upper bound is $O(1)$ for Trie, for hashing, the best possible average case time complexity is $O(1)$. Also, with Trie we can implement prefix search (finding all IPs for a common prefix of URLs). The general disadvantage of Trie is large amount of memory requirement. The idea is to store URLs in Trie nodes and store the corresponding IP address in last or leaf node.

Following is C style implementation in C++.

```
// C based program to implement reverse DNS lookup
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

// There are almost 27 different chars in a valid URL
// assuming URL consists [a-z] and (.)
#define CHARS 27

// Maximum length of a valid URL
#define MAX 100

// A utility function to find index of child for a given character 'c'
int getIndex(char c)
{
    return (c == '.') ? 26 : (c - 'a');
}

// A utility function to find character for a given child index.
char getCharFromIndex(int i)
{
    return (i == 26) ? '.' : ('a' + i);
}
```

```

// Trie Node.
struct trieNode
{
    bool isLeaf;
    char *ipAdd;
    struct trieNode *child[CHARS];
};

// Function to create a new trie node.
struct trieNode *newTrieNode(void)
{
    struct trieNode *newNode = new trieNode;
    newNode->isLeaf = false;
    newNode->ipAdd = NULL;
    for (int i = 0; i < CHARS; i++)
        newNode->child[i] = NULL;
    return newNode;
}

// This method inserts a URL and corresponding IP address
// in the trie. The last node in Trie contains the ip address.
void insert(struct trieNode *root, char *URL, char *ipAdd)
{
    // Length of the URL
    int len = strlen(URL);
    struct trieNode *pCrawl = root;

    // Traversing over the length of the URL.
    for (int level = 0; level < len; level++)
    {
        // Get index of child node from current character
        // in URL[] Index must be from 0 to 26 where
        // 0 to 25 is used for alphabets and 26 for dot
        int index = getIndex(URL[level]);

        // Create a new child if not exist already
        if (pCrawl->child[index])
            pCrawl->child[index] = newTrieNode();

        // Move to the child
        pCrawl = pCrawl->child[index];
    }

    // Below needs to be carried out for the last node.
    // Save the corresponding ip address of the URL in the
    // last node of trie.
    pCrawl->isLeaf = true;
    pCrawl->ipAdd = new char[strlen(ipAdd) + 1];
    strcpy(pCrawl->ipAdd, ipAdd);
}

// This function returns IP address if given URL is
// present in DNS cache. Else returns NULL
char *searchDNSCache(struct trieNode *root, char *URL)
{
    // Root node of trie.
    struct trieNode *pCrawl = root;
    int len = strlen(URL);

    // Traversal over the length of URL.
    for (int level = 0; level < len; level++)
    {
        int index = getIndex(URL[level]);
        if (pCrawl->child[index])
            return NULL;
        pCrawl = pCrawl->child[index];
    }

    // If we find the last node for a given ip address,
    // print the ip address.
    if (pCrawl != NULL && pCrawl->isLeaf)
        return pCrawl->ipAdd;

    return NULL;
}

// Driver function.
int main()
{
    char URL[50] = { "www.samsung.com", "www.samsung.net",
        "www.google.in"
    };
    char ipAdd[100] = { "107.108.11.123", "107.109.123.255",
        "74.125.200.106"
    };
    int n = sizeof(URL) / sizeof(URL[0]);
    struct trieNode *root = newTrieNode();

    // Inserts all the domain name and their corresponding
    // ip address
    for (int i = 0; i < n; i++)
        insert(root, URL[i], ipAdd[i]);

    // If forward DNS look up succeeds print the url along
    // with the resolved ip address.
    char url[] = "www.samsung.com";
    char *res_ip = searchDNSCache(root, url);
    if (res_ip != NULL)
        printf("Forward DNS look up resolved in cache:\n%s --> %s",
            url, res_ip);
    else
        printf("Forward DNS look up not resolved in cache ");

    return 0;
}

```

Output:

```

Forward DNS look up resolved in cache:
www.samsung.com --> 107.108.11.123

```

Binary Indexed Tree or Fenwick Tree

Let us consider the following problem to understand Binary Indexed Tree.

We have an array $arr[0 \dots n-1]$. We should be able to

- 1 Find the sum of first i elements.
- 2 Change value of a specified element of the array $arr[i] = x$ where $0 \leq i \leq n-1$.

A **simple solution** is to run a loop from 0 to $i-1$ and calculate sum of elements. To update a value, simply do $arr[i] = x$. The first operation takes $O(n)$ time and second operation takes $O(1)$ time. Another simple solution is to create another array and store sum from start to i at the i 'th index in this array. Sum of a given range can now be calculated in $O(1)$ time, but update operation takes $O(n)$ time now. This works well if the number of query operations are large and very few updates.

Can we perform both the operations in $O(\log n)$ time once given the array?

One Efficient Solution is to use **Segment Tree** that does both operations in $O(\log n)$ time.

Using **Binary Indexed Tree**, we can do both tasks in $O(\log n)$ time. The advantages of **Binary Indexed Tree** over **Segment** are, requires less space and very easy to implement..

Representation

Binary Indexed Tree is represented as an array. Let the array be $BITree[]$. Each node of Binary Indexed Tree stores sum of some elements of given array. Size of Binary Indexed Tree is equal to n where n is size of input array. In the below code, we have used size as $n+1$ for ease of implementation.

Construction

We construct the Binary Indexed Tree by first initializing all values in $BITree[]$ as 0. Then we call $update()$ operation for all indexes to store actual sums, update is discussed below.

Operations

```
getSum(index): Returns sum of arr[0..index]
// Returns sum of arr[0..index] using BITree[0..n]. It assumes that
// BITree[] is constructed for given array arr[0..n-1]
1) Initialize sum as 0 and index as index+1.
2) Do following while index is greater than 0.
...a) Add BITree[index] to sum
...b) Go to parent of BITree[index]. Parent can be obtained by removing
    the last set bit from index, i.e., index = index - (index & (-index))
3) Return sum.
```

BITSum



The above diagram demonstrates working of $getSum()$. Following are some important observations.

Node at index 0 is a dummy node.

A node at index y is parent of a node at index x , iff y can be obtained by removing last set bit from binary representation of x .

A child x of a node y stores sum of elements from of y (exclusive y) and of x (inclusive x).

```
update(index, val): Updates BIT for operation arr[index] += val
// Note that arr[] is not changed here. It changes
// only BI Tree for the already made change in arr[].
1) Initialize index as index+1.
2) Do following while index is smaller than or equal to n.
...a) Add value to BITree[index]
...b) Go to parent of BITree[index]. Parent can be obtained by removing
    the last set bit from index, i.e., index = index + (index & (-index))
```

BITUpdate1



The update process needs to make sure that all $BITree$ nodes that have $arr[i]$ as part of the section they cover must be updated. We get all such nodes of $BITree$ by repeatedly adding the decimal number corresponding to

the last set bit.

How does Binary Indexed Tree work?

The idea is based on the fact that all positive integers can be represented as sum of powers of 2. For example 19 can be represented as $16 + 2 + 1$. Every node of BI Tree stores sum of n elements where n is a power of 2. For example, in the above first diagram for `getSum()`, sum of first 12 elements can be obtained by sum of last 4 elements (from 9 to 12) plus sum of 8 elements (from 1 to 8). The number of set bits in binary representation of a number n is $O(\text{Log}n)$. Therefore, we traverse at-most $O(\text{Log}n)$ nodes in both `getSum()` and `update()` operations. Time complexity of construction is $O(n\text{Log}n)$ as it calls `update()` for all n elements.

Implementation:

Following are C++ and Python implementations of Binary Indexed Tree.

C++

```
// C++ code to demonstrate operations of Binary Index Tree
#include <iostream>
using namespace std;

/*      n --> No. of elements present in input array.
    BITree[0..n] --> Array that represents Binary Indexed Tree.
    arr[0..n-1] --> Input array for which prefix sum is evaluated. */

// Returns sum of arr[0..index]. This function assumes
// that the array is preprocessed and partial sums of
// array elements are stored in BITree[].
int getSum(int BITree[], int index)
{
    int sum = 0; // Initialize result

    // index in BITree[] is 1 more than the index in arr[]
    index = index + 1;

    // Traverse ancestors of BITree[index]
    while (index > 0)
    {
        // Add current element of BITree to sum
        sum += BITree[index];

        // Move index to parent node in getSum View
        index -= index & (-index);
    }
    return sum;
}

// Updates a node in Binary Index Tree (BITree) at given index
// in BITree. The given value 'val' is added to BITree[i] and
// all of its ancestors in tree.
void updateBIT(int BITree[], int n, int index, int val)
{
    // index in BITree[] is 1 more than the index in arr[]
    index = index + 1;

    // Traverse all ancestors and add 'val'
    while (index <= n)
    {
        // Add 'val' to current node of Bi Tree
        BITree[index] += val;

        // Update index to that of parent in update View
        index += index & (-index);
    }
}

// Constructs and returns a Binary Indexed Tree for given
// array of size n.
int *constructBITree(int arr[], int n)
{
    // Create and initialize BITree[] as 0
    int *BITree = new int[n+1];
    for (int i=1; i<=n; i++)
        BITree[i] = 0;

    // Store the actual values in BITree[] using update()
    for (int i=0; i<n; i++)
        updateBIT(BITree, n, i, arr[i]);

    // Uncomment below lines to see contents of BITree[]
    //for (int i=1; i<=n; i++)
    //    cout << BITree[i] << " ";

    return BITree;
}

// Driver program to test above functions
int main()
{
    int freq[] = {2, 1, 1, 3, 2, 3, 4, 5, 6, 7, 8, 9};
    int n = sizeof(freq)/sizeof(freq[0]);
    int *BITree = constructBITree(freq, n);
    cout << "Sum of elements in arr[0..5] is "
         << getSum(BITree, 5);

    // Let us test the update operation
    freq[3] += 6;
    updateBIT(BITree, n, 3, 6); //Update BIT for above change in arr[]

    cout << "\nSum of elements in arr[0..5] after update is "
         << getSum(BITree, 5);

    return 0;
}
```

Python

```
# Python implementation of Binary Indexed Tree

# Returns sum of arr[0..index]. This function assumes
# that the array is preprocessed and partial sums of
# array elements are stored in BITree[].
def getsum(BITree,i):
```



```

s = 0 #initialize result

# index in BITree[] is 1 more than the index in arr[]
i = i+1

# Traverse ancestors of BITree[index]
while i > 0:

    # Add current element of BITree to sum
    s += BITree[i]

    # Move index to parent node in getSum View
    i = i & (-i)
return s

# Updates a node in Binary Index Tree (BITree) at given index
# in BITree. The given value 'val' is added to BITree[i] and
# all of its ancestors in tree.
def updatebit(BITree , n , i ,v):

    # index in BITree[] is 1 more than the index in arr[]
    i += 1

    # Traverse all ancestors and add 'val'
    while i <= n:

        # Add 'val' to current node of Bi Tree
        BITree[i] += v

        # Update index to that of parent in update View
        i += i & (-i)

# Constructs and returns a Binary Indexed Tree for given
# array of size n.
def construct(arr, n):

    # Create and initialize BITree[] as 0
    BITree = [0]*(n+1)

    # Store the actual values in BITree[] using update()
    for i in range(n):
        updatebit(BITree, n, i, arr[i])

    # Uncomment below lines to see contents of BITree[]
    #for i in range(1,n+1):
    #    print BITree[i],
    return BITree

# Driver code to test above methods
freq = [2, 1, 1, 3, 2, 3, 4, 5, 6, 7, 8, 9]
BITree = construct(freq,len(freq))
print("Sum of elements in arr[0..5] is " + str(getsum(BITree,5)))
freq[3] += 6
updatebit(BITree, len(freq), 3, 6)
print("Sum of elements in arr[0..5] is " + str(getsum(BITree,5)))

# This code is contributed by Raju Varshney

```

Output:

```

Sum of elements in arr[0..5] is 12
Sum of elements in arr[0..5] after update is 18

```

Can we extend the Binary Indexed Tree for range Sum in Logn time?

This is simple to answer. The rangeSum(l, r) can be obtained as getSum(r) – getSum(l-1).

Applications:

Used to implement the arithmetic coding algorithm. Development of operations it supports were primarily motivated by use in that case. See [this](#) for more details.

Example Problems:

[Count inversions in an array | Set 3 \(Using BIT\)](#)

[Two Dimensional Binary Indexed Tree or Fenwick Tree](#)

[Counting Triangles in a Rectangular space using BIT](#)

References:

http://en.wikipedia.org/wiki/Fenwick_tree

<http://community.topcoder.com/to?module=Static&d1=tutorials&d2=binaryIndexedTrees>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Advance Data Structures
Advanced Data Structures
Binary-Indexed-Tree

Two Dimensional Binary Indexed Tree or Fenwick Tree

Prerequisite – [Fenwick Tree](#)

We know that to answer range sum queries on a 1-D array efficiently, binary indexed tree (or Fenwick Tree) is the best choice (even better than segment tree due to less memory requirements and a little faster than segment tree).

Can we answer sub-matrix sum queries efficiently using Binary Indexed Tree ?

The answer is **yes**. This is possible using a **2D BIT** which is nothing but an array of 1D BIT.

Algorithm:

We consider the below example. Suppose we have to find the sum of all numbers inside the highlighted area-



We assume the origin of the matrix at the bottom – O. Then a 2D BIT exploits the fact that-

Sum under the marked area = Sum(OB) - Sum(OD) -
Sum(OA) + Sum(OC)





In our program, we use the `getSum(x, y)` function which finds the sum of the matrix from (0, 0) to (x, y).
Hence the below formula :

Sum under the marked area = $\text{Sum}(\text{OB}) - \text{Sum}(\text{OD}) - \text{Sum}(\text{OA}) + \text{Sum}(\text{OC})$

The above formula gets reduced to,

Query(x_1, y_1, x_2, y_2) = $\text{getSum}(x_2, y_2) - \text{getSum}(x_2, y_1 - 1) - \text{getSum}(x_1 - 1, y_2) + \text{getSum}(x_1 - 1, y_1 - 1)$

where,

x1, y1 = x and y coordinates of C

x2, y2 = x and y coordinates of B

The `updateBIT(x, y, val)` function updates all the elements under the region – (x, y) to (N, M) where,

N = maximum X co-ordinate of the whole matrix.

M = maximum Y co-ordinate of the whole matrix.

The rest procedure is quite similar to that of 1D Binary Indexed Tree. Below is the C++ implementation of 2D indexed tree

/* C++ program to implement 2D Binary Indexed Tree

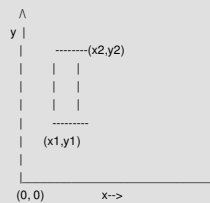
2D BIT is basically a BIT where each element is another BIT.
Updating by adding v on (x, y) means its effect will be found throughout the rectangle [(x, y), (max_x, max_y)],
and query for (x, y) gives you the result of the rectangle [(0, 0), (x, y)], assuming the total rectangle is [(0, 0), (max_x, max_y)]. So when you query and update on this BIT, you have to be careful about how many times you are subtracting a rectangle and adding it. Simple set union formula works here.

So if you want to get the result of a specific rectangle [(x1, y1), (x2, y2)], the following steps are necessary:

Query(x_1, y_1, x_2, y_2) = $\text{getSum}(x_2, y_2) - \text{getSum}(x_2, y_1 - 1) - \text{getSum}(x_1 - 1, y_2) + \text{getSum}(x_1 - 1, y_1 - 1)$

Here 'Query(x_1, y_1, x_2, y_2)' means the sum of elements enclosed in the rectangle with bottom-left corner's co-ordinates (x1, y1) and top-right corner's co-ordinates - (x2, y2)

Constraints -> $x_1 \leq x_2$ and $y_1 \leq y_2$



In this program we have assumed a square matrix. The program can be easily extended to a rectangular one. */

```
#include <bits/stdc++.h>
using namespace std;
```

```
#define N 4 // N->max_x and max_y
```

```
// A structure to hold the queries
struct Query
{
```

```
    int x1, y1; // x and y co-ordinates of bottom left
    int x2, y2; // x and y co-ordinates of top right
};
```

```
// A function to update the 2D BIT
void updateBIT(int BIT[][N+1], int x, int y, int val)
```

```
{
    for (; x <= N; x += (x & -x))
    {
        // This loop update all the 1D BIT inside the
        // array of 1D BIT = BIT[x]
        for (; y <= N; y += (y & -y))
            BIT[x][y] += val;
        }
    return;
}
```

```
// A function to get sum from (0, 0) to (x, y)
int getSum(int BIT[][N+1], int x, int y)
```

```
{
    int sum = 0;

    for (; x > 0; x -= (x & -x))
    {
        // This loop sum through all the 1D BIT
        // inside the array of 1D BIT = BIT[x]
        for (; y > 0; y -= (y & -y))
        {
            sum += BIT[x][y];
        }
    }
    return sum;
}
```

```
// A function to create an auxiliary matrix
// from the given input matrix
void constructAux(int mat[][N], int aux[][N+1])
{
    // Initialise Auxiliary array to 0
    for (int i=0; i<=N; i++)
        for (int j=0; j<=N; j++)
            aux[i][j] = 0;
}
```

```

// Construct the Auxiliary Matrix
for (int j=1; j<=N; j++)
    for (int i=1; i<=N; i++)
        aux[i][j] = mat[N-j][i-1];

return;
}

// A function to construct a 2D BIT
void construct2DBIT(int mat[][N], int BIT[][N+1])
{
    // Create an auxiliary matrix
    int aux[N+1][N+1];
    constructAux(mat, aux);

    // Initialise the BIT to 0
    for (int i=1; i<=N; i++)
        for (int j=1; j<=N; j++)
            BIT[i][j] = 0;

    for (int j=1; j<=N; j++)
    {
        for (int i=1; i<=N; i++)
        {
            // Creating a 2D-BIT using update function
            // everytime we encounter a value in the
            // input 2D-array
            int v1 = getSum(BIT, i, j);
            int v2 = getSum(BIT, i, j-1);
            int v3 = getSum(BIT, i-1, j-1);
            int v4 = getSum(BIT, i-1, j);

            // Assigning a value to a particular element
            // of 2D BIT
            updateBIT(BIT, i, j, aux[i][j]-(v1-v2-v4+v3));
        }
    }

    return;
}

// A function to answer the queries
void answerQueries(Query q[], int m, int BIT[][N+1])
{
    for (int i=0; i<m; i++)
    {
        int x1 = q[i].x1 + 1;
        int y1 = q[i].y1 + 1;
        int x2 = q[i].x2 + 1;
        int y2 = q[i].y2 + 1;

        int ans = getSum(BIT, x2, y2)-getSum(BIT, x2, y1-1)-
            getSum(BIT, x1-1, y2)+getSum(BIT, x1-1, y1-1);

        printf ("Query(%d, %d, %d, %d) = %d\n",
            q[i].x1, q[i].y1, q[i].x2, q[i].y2, ans);
    }
    return;
}

// Driver program
int main()
{
    int mat[N][N] = {{1, 2, 3, 4},
        {5, 3, 8, 1},
        {4, 6, 7, 5},
        {2, 4, 8, 9}};

    // Create a 2D Binary Indexed Tree
    int BIT[N+1][N+1];
    construct2DBIT(mat, BIT);

    /* Queries of the form - x1, y1, x2, y2
    For example the query- {1, 1, 3, 2} means the sub-matrix-
    y
    ^
    3 | 1 2 3 4   Sub-matrix
    2 | 5 3 8 1   {1,1,3,2}  --> 3 8 1
    1 | 4 6 7 5           6 7 5
    0 | 2 4 8 9
    |
    --|----- 0 1 2 3 ----> x
    |

    Hence sum of the sub-matrix = 3+8+1+6+7+5 = 30

    */

    Query q[] = {{1, 1, 3, 2}, {2, 3, 3, 3}, {1, 1, 1, 1}};
    int m = sizeof(q)/sizeof(q[0]);

    answerQueries(q, m, BIT);

    return(0);
}

```

Output:

```

Query(1, 1, 3, 2) = 30
Query(2, 3, 3, 3) = 7
Query(1, 1, 1, 1) = 6

```

Time Complexity:

- Both updateBIT(x, y, val) function and getSum(x, y) function takes $O(\log(NM))$ time.
- Building the 2D BIT takes $O(NM \log(NM))$.
- Since in each of the queries we are calling getSum(x, y) function so answering all the Q queries takes $O(Q \cdot \log(NM))$ time.

Hence the overall time complexity of the program is $O((NM+Q) \cdot \log(NM))$ where,

N = maximum X co-ordinate of the whole matrix.

M = maximum Y co-ordinate of the whole matrix.

Q = Number of queries.

Auxiliary Space: $O(NM)$ to store the BIT and the auxiliary array

References: <https://www.topcoder.com/community/data-science/data-science-tutorials/binary-indexed-trees/>

This article is contributed by **Rachit Belwariar**. If you like GeeksforGeeks and would like to contribute, you can also write an article using [contribute.geeksforgeeks.org](https://www.geeksforgeeks.org/contribute/) or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Binary-Indexed-Tree

Binary Indexed Tree : Range Updates and Point Queries

Given an array $arr[0..n-1]$. The following operations need to be performed.

1. **update(l, r, val)** : Add 'val' to all the elements in the array from [l, r].
2. **getElement(i)** : Find element in the array indexed at 'i'.

Initially all the elements in the array are 0. Queries can be in any order, i.e., there can be many updates before point query.

Example:

```
Input : arr = {0, 0, 0, 0, 0}
Queries: update : l = 0, r = 4, val = 2
        getElement : i = 3
        update : l = 3, r = 4, val = 3
        getElement : i = 3

Output: Element at 3 is 2
        Element at 3 is 5

Explanation : Array after first update becomes
              {2, 2, 2, 2, 2}
              Array after second update becomes
              {2, 2, 2, 5, 5}
```

Method 1 [update : $O(n)$, getElement() : $O(1)$]

1. **update(l, r, val)** : Iterate over the subarray from l to r and increase all the elements by val.
2. **getElement(i)** : To get the element at ith index, simply return $arr[i]$.

The time complexity in worst case is $O(q \cdot n)$ where q is number of queries and n is number of elements.

Method 2 [update : $O(1)$, getElement() : $O(n)$]

We can avoid updating all elements and can update only 2 indexes of the array!

1. **update(l, r, val)** : Add 'val' to the lth element and subtract 'val' from the $(r+1)$ th element, do this for all the update queries.

```
arr[l] = arr[l] + val
arr[r+1] = arr[r+1] - val
```

2. **getElement(i)** : To get ith element in the array find the sum of all integers in the array from 0 to i. (Prefix Sum).

Let's analyze the update query. **Why to add val to lth index?** Adding val to lth index means that all the elements after l are increased by val, since we will be computing the prefix sum for every element. **Why to subtract val from $(r+1)$ th index?** A range update was required from [l, r] but what we have updated is [l, n-1] so we need to remove val from all the elements after r i.e., subtract val from $(r+1)$ th index. Thus the val is added to range [l, r]. Below is C++ implementation of above approach.

```
// C++ program to demonstrate Range Update
// and Point Queries Without using BIT
#include <iostream>
using namespace std;

// Updates such that getElement() gets an increased
// value when queried from l to r.
void update(int arr[], int l, int r, int val)
{
    arr[l] += val;
    arr[r+1] -= val;
}

// Get the element indexed at i
int getElement(int arr[], int i)
{
    // To get ith element sum of all the elements
    // from 0 to i need to be computed
    int res = 0;
    for (int j = 0; j <= i; j++)
        res += arr[j];

    return res;
}

// Driver program to test above function
int main()
{
    int arr[] = {0, 0, 0, 0, 0};
    int n = sizeof(arr) / sizeof(arr[0]);

    int l = 2, r = 4, val = 2;
    update(arr, l, r, val);

    // Find the element at Index 4
    int index = 4;
    cout << "Element at index " << index << " is " <<
        getElement(arr, index) << endl;

    l = 0, r = 3, val = 4;
    update(arr, l, r, val);

    // Find the element at Index 3
    index = 3;
    cout << "Element at index " << index << " is " <<
        getElement(arr, index) << endl;

    return 0;
}
```

Output:

Element at index 4 is 2
Element at index 3 is 6

Time complexity : $O(q \cdot n)$ where q is number of queries.

Method 3 (Using Binary Indexed Tree)

In method 2, we have seen that the problem can be reduced to update and prefix sum queries. We have seen that BIT can be used to do update and prefix sum queries in $O(\log n)$ time.

Below is C++ implementation.

```
// C++ code to demonstrate Range Update and
// Point Queries on a Binary Index Tree
#include <iostream>
using namespace std;

// Updates a node in Binary Index Tree (BITree) at given index
// in BITree. The given value 'val' is added to BITree[i] and
// all of its ancestors in tree.
void updateBIT(int BITree[], int n, int index, int val)
{
    // index in BITree[] is 1 more than the index in arr[]
    index = index + 1;

    // Traverse all ancestors and add 'val'
    while (index <= n)
    {
        // Add 'val' to current node of BITree
        BITree[index] += val;

        // Update index to that of parent in update View
        index += index & (-index);
    }
}

// Constructs and returns a Binary Indexed Tree for given
// array of size n.
int *constructBITree(int arr[], int n)
{
    // Create and initialize BITree[] as 0
    int *BITree = new int[n+1];
    for (int i=1; i<=n; i++)
        BITree[i] = 0;

    // Store the actual values in BITree[] using update()
    for (int i=0; i<n; i++)
        updateBIT(BITree, n, i, arr[i]);

    // Uncomment below lines to see contents of BITree[]
    //for (int i=1; i<=n; i++)
    //    cout << BITree[i] << " ";

    return BITree;
}

// SERVES THE PURPOSE OF getElement()
// Returns sum of arr[0..index]. This function assumes
// that the array is preprocessed and partial sums of
// array elements are stored in BITree[]
int getSum(int BITree[], int index)
{
    int sum = 0; // Initialize result

    // index in BITree[] is 1 more than the index in arr[]
    index = index + 1;

    // Traverse ancestors of BITree[index]
    while (index>0)
    {
        // Add current element of BITree to sum
        sum += BITree[index];

        // Move index to parent node in getSum View
        index -= index & (-index);
    }
    return sum;
}

// Updates such that getElement() gets an increased
// value when queried from l to r.
void update(int BITree[], int l, int r, int n, int val)
{
    // Increase value at l by 'val'
    updateBIT(BITree, n, l, val);

    // Decrease value at r+1 by 'val'
    updateBIT(BITree, n, r+1, -val);
}

// Driver program to test above function
int main()
{
    int arr[] = {0, 0, 0, 0, 0};
    int n = sizeof(arr)/sizeof(arr[0]);
    int *BITree = constructBITree(arr, n);

    // Add 2 to all the element from [2,4]
    int l = 2, r = 4, val = 2;
    update(BITree, l, r, n, val);

    // Find the element at Index 4
    int index = 4;
    cout << "Element at index " << index << " is " <<
        getSum(BITree, index) << "\n";

    // Add 2 to all the element from [0,3]
    l = 0, r = 3, val = 4;
    update(BITree, l, r, n, val);

    // Find the element at Index 3
    index = 3;
    cout << "Element at index " << index << " is " <<
        getSum(BITree, index) << "\n";
}
```

```
    return 0;
}
```

Output:

```
Element at index 4 is 2
Element at index 3 is 6
```

Time Complexity : $O(q * \log n) + O(n * \log n)$ where q is number of queries.

Method 1 is efficient when most of the queries are `getElement()`, method 2 is efficient when most of the queries are `updates()` and method 3 is preferred when there is mix of both queries.

This article is contributed by **Chirag Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Trees
Binary-Indexed-Tree

Binary Indexed Tree : Range Update and Range Queries

Given an array `arr[0..n-1]`. The following operations need to be performed.

1. **update(l, r, val)** : Add 'val' to all the elements in the array from [l, r].
2. **getRangeSum(l, r)** : Find sum of all elements in array from [l, r].

Initially all the elements in the array are 0. Queries can be in any order, i.e., there can be many updates before range sum.

Example:

```
Input : n = 5 // {0, 0, 0, 0, 0}
Queries: update : l = 0, r = 4, val = 2
        update : l = 3, r = 4, val = 3
        getRangeSum : l = 2, r = 4
```

Output: Sum of elements of range [2, 4] is 12

```
Explanation : Array after first update becomes
{2, 2, 2, 2, 2}
Array after second update becomes
{2, 2, 2, 5, 5}
```

In the [previous post](#), we discussed range update and point query solutions using BIT.

`rangeUpdate(l, r, val)` : We add 'val' to element at index 'l'. We subtract 'val' from element at index 'r+1'.

`getElement(index)` [or `getSum()`]: We return sum of elements from 0 to index which can be quickly obtained using BIT.

We can compute `rangeSum()` using `getSum()` queries.

`rangeSum(l, r) = getSum(r) - getSum(l-1)`

A **Simple Solution** is to use solutions discussed in [previous post](#). Range update query is same. Range sum query can be achieved by doing get query for all elements in range.

An **Efficient Solution** is to make sure that both queries can be done in $O(\log n)$ time. We get range sum using prefix sums. How to make sure that update is done in a way so that prefix sum can be done quickly? Consider a situation where prefix sum `[0, k]` (where $0 \leq k < n$) is needed after range update on range `[l, r]`. Three cases arises as k can possibly lie in 3 regions.

Case 1: $0 < k < l$

The update query won't affect sum query.

Case 2: $l \leq k \leq r$

Consider an example:

```
Add 2 to range [2, 4], the resultant array would be:
0 0 2 2 2
If k = 3
Sum from [0, k] = 4
```

How to get this result?

Simply add the val from l^{th} index to k^{th} index. Sum is incremented by `val*(k) - val*(l-1)` after update query.

Case 3: $k > r$

For this case, we need to add "val" from l^{th} index to r^{th} index. Sum is incremented by `val*r - val*(l-1)` due to update query.

Observations :

Case 1: is simple as sum would remain same as it was before update.

Case 2: Sum was incremented by `val*k - val*(l-1)`. We can find "val", it is similar to finding the i^{th} element in [range update and point query article](#). So we maintain one BIT for Range Update and Point Queries, this BIT will be helpful in finding the value at k^{th} index. Now `val * k` is computed, how to handle extra term `val*(l-1)`?

In order to handle this extra term, we maintain another BIT (BIT2). Update `val * (l-1)` at l^{th} index, so when `getSum` query is performed on BIT2 will give result as `val*(l-1)`.

Case 3 : The sum in case 3 was incremented by `val*r - val*(l-1)`, the value of this term can be obtained using BIT2. Instead of adding, we subtract `val*(l-1) - val*r` as we can get this value from BIT2 by adding `val*(l-1)` as we did in case 2 and subtracting `val*r` in every update operation.

```
Update Query
Update(BITree1, l, val)
Update(BITree1, r+1, -val)
UpdateBIT2(BITree2, l, val*(l-1))
UpdateBIT2(BITree2, r+1, -val*r)

Range Sum
getSum(BITree1, k) - getSum(BITree2, k)
```

C++ Implementation of above idea

```
// C++ program to demonstrate Range Update
// and Range Queries using BIT
#include <iostream>
using namespace std;

// Returns sum of arr[0..index]. This function assumes
// that the array is preprocessed and partial sums of
// array elements are stored in BITree[]
int getSum(int BITree[], int index)
{
    int sum = 0; // Initialize result

    // index in BITree[] is 1 more than the index in arr[]
    index = index + 1;
```

```

// Traverse ancestors of BITree[index]
while (index>0)
{
    // Add current element of BITree to sum
    sum += BITree[index];

    // Move index to parent node in getSum View
    index -= index & (-index);
}
return sum;
}

// Updates a node in Binary Index Tree (BITree) at given
// index in BITree. The given value 'val' is added to
// BITree[i] and all of its ancestors in tree.
void updateBIT(int BITree[], int n, int index, int val)
{
    // index in BITree[] is 1 more than the index in arr[]
    index = index + 1;

    // Traverse all ancestors and add 'val'
    while (index <= n)
    {
        // Add 'val' to current node of BI Tree
        BITree[index] += val;

        // Update index to that of parent in update View
        index += index & (-index);
    }
}

// Returns the sum of array from [0, x]
int sum(int x, int BITree1[], int BITree2[])
{
    return (getSum(BITree1, x) * x) - getSum(BITree2, x);
}

void updateRange(int BITree1[], int BITree2[], int n,
                int val, int l, int r)
{
    // Update Both the Binary Index Trees
    // As discussed in the article

    // Update BIT1
    updateBIT(BITree1, n, l, val);
    updateBIT(BITree1, n, r+1, -val);

    // Update BIT2
    updateBIT(BITree2, n, l, val*(l-1));
    updateBIT(BITree2, n, r+1, -val*r);
}

int rangeSum(int l, int r, int BITree1[], int BITree2[])
{
    // Find sum from [0,r] then subtract sum
    // from [0,l-1] in order to find sum from
    // [l,r]
    return sum(r, BITree1, BITree2) -
           sum(l-1, BITree1, BITree2);
}

int *constructBITree(int n)
{
    // Create and initialize BITree[] as 0
    int *BITree = new int[n+1];
    for (int i=1; i<=n; i++)
        BITree[i] = 0;

    return BITree;
}

// Driver Program to test above function
int main()
{
    int n = 5;

    // Construct two BIT
    int *BITree1, *BITree2;

    // BIT1 to get element at any index
    // in the array
    BITree1 = constructBITree(n);

    // BIT 2 maintains the extra term
    // which needs to be subtracted
    BITree2 = constructBITree(n);

    // Add 5 to all the elements from [0,4]
    int l = 0, r = 4, val = 5;
    updateRange(BITree1, BITree2, n, val, l, r);

    // Add 2 to all the elements from [2,4]
    l = 2, r = 4, val = 10;
    updateRange(BITree1, BITree2, n, val, l, r);

    // Find sum of all the elements from
    // [1,4]
    l = 1, r = 4;
    cout << "Sum of elements from [" << l
    << " , " << r << "] is ";
    cout << rangeSum(l, r, BITree1, BITree2) << "n";

    return 0;
}

```

Output:

```
Sum of elements from [1,4] is 50
```

Time Complexity : $O(q \log(n))$ where q is number of queries.

This article is contributed by **Chirag Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org.

See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Binary-Indexed-Tree

Suffix Array | Set 1 (Introduction)

We strongly recommend to read following post on suffix trees as a pre-requisite for this post.

[Pattern Searching | Set 8 \(Suffix Tree Introduction\)](#)

A suffix array is a sorted array of all suffixes of a given string. The definition is similar to [Suffix Tree](#) which is compressed trie of all suffixes of the given text. Any suffix tree based algorithm can be replaced with an algorithm that uses a suffix array enhanced with additional information and solves the same problem in the same time complexity (Source [Wiki](#)).

A suffix array can be constructed from Suffix tree by doing a DFS traversal of the suffix tree. In fact Suffix array and suffix tree both can be constructed from each other in linear time.

Advantages of suffix arrays over suffix trees include improved space requirements, simpler linear time construction algorithms (e.g., compared to Ukkonen's algorithm) and improved cache locality (Source: [Wiki](#))

Example:

Let the given string be "banana".

```
0 banana          5 a
1 anana   Sort the Suffixes   3 ana
2 nana   ----->   1 anana
3 ana   alphabetically   0 banana
4 na          4 na
5 a          2 nana
```

So the suffix array for "banana" is [5, 3, 1, 0, 4, 2]

Naive method to build Suffix Array

A simple method to construct suffix array is to make an array of all suffixes and then sort the array. Following is implementation of simple method.

```
// Naive algorithm for building suffix array of a given text
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

// Structure to store information of a suffix
struct suffix
{
    int index;
    char *suff;
};

// A comparison function used by sort() to compare two suffixes
int cmp(struct suffix a, struct suffix b)
{
    return strcmp(a.suff, b.suff) < 0 ? 1 : 0;
}

// This is the main function that takes a string 'txt' of size n as an
// argument, builds and return the suffix array for the given string
int *buildSuffixArray(char *txt, int n)
{
    // A structure to store suffixes and their indexes
    struct suffix suffixes[n];

    // Store suffixes and their indexes in an array of structures.
    // The structure is needed to sort the suffixes alphabetically
    // and maintain their old indexes while sorting
    for (int i = 0; i < n; i++)
    {
        suffixes[i].index = i;
        suffixes[i].suff = (txt+i);
    }

    // Sort the suffixes using the comparison function
    // defined above.
    sort(suffixes, suffixes+n, cmp);

    // Store indexes of all sorted suffixes in the suffix array
    int *suffixArr = new int[n];
    for (int i = 0; i < n; i++)
        suffixArr[i] = suffixes[i].index;

    // Return the suffix array
    return suffixArr;
}

// A utility function to print an array of given size
void printArr(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver program to test above functions
int main()
{
    char txt[] = "banana";
    int n = strlen(txt);
    int *suffixArr = buildSuffixArray(txt, n);
    cout << "Following is suffix array for " << txt << endl;
    printArr(suffixArr, n);
    return 0;
}
```

Output:

```
Following is suffix array for banana
5 3 1 0 4 2
```

The time complexity of above method to build suffix array is $O(n^2 \log n)$ if we consider a $O(n \log n)$ algorithm used for sorting. The sorting step itself takes $O(n^2 \log n)$ time as every comparison is a comparison of two strings and the comparison takes $O(n)$ time.

There are many efficient algorithms to build suffix array. We will soon be covering them as separate posts.

Search a pattern using the built Suffix Array

To search a pattern in a text, we preprocess the text and build a suffix array of the text. Since we have a sorted array of all suffixes, [Binary Search](#) can be used to search. Following is the search function. Note that the function doesn't report all occurrences of pattern, it only report one of them.

```
// This code only contains search() and main. To make it a complete running
// above code or see http://code.geeksforgeeks.org/oY7OkD

// A suffix array based search function to search a given pattern
// 'pat' in given text 'txt' using suffix array suffArr[]
void search(char *pat, char *txt, int *suffArr, int n)
{
    int m = strlen(pat); // get length of pattern, needed for strcmp()

    // Do simple binary search for the pat in txt using the
    // built suffix array
    int l = 0, r = n-1; // Initialize left and right indexes
    while (l <= r)
    {
        // See if 'pat' is prefix of middle suffix in suffix array
        int mid = l + (r - l)/2;
        int res = strcmp(pat, txt+suffArr[mid], m);

        // If match found at the middle, print it and return
        if (res == 0)
        {
            cout << "Pattern found at index " << suffArr[mid];
            return;
        }

        // Move to left half if pattern is alphabetically less than
        // the mid suffix
        if (res < 0) r = mid - 1;

        // Otherwise move to right half
        else l = mid + 1;
    }

    // We reach here if return statement in loop is not executed
    cout << "Pattern not found";
}

// Driver program to test above function
int main()
{
    char txt[] = "banana"; // text
    char pat[] = "nan"; // pattern to be searched in text

    // Build suffix array
    int n = strlen(txt);
    int *suffArr = buildSuffixArray(txt, n);

    // search pat in txt using the built suffix array
    search(pat, txt, suffArr, n);

    return 0;
}
```

Output:

```
Pattern found at index 2
```

The time complexity of the above search function is $O(m \log n)$. There are more efficient algorithms to search pattern once the suffix array is built. In fact there is a $O(m)$ suffix array based algorithm to search a pattern. We will soon be discussing efficient algorithm for search.

Applications of Suffix Array

Suffix array is an extremely useful data structure, it can be used for a wide range of problems. Following are some famous problems where Suffix array can be used.

- 1) Pattern Searching
- 2) [Finding the longest repeated substring](#)
- 3) [Finding the longest common substring](#)
- 4) [Finding the longest palindrome in a string](#)

See [this](#) for more problems where Suffix arrays can be used.

This post is a simple introduction. There is a lot to cover in Suffix arrays. We have discussed a $O(n \log n)$ algorithm for Suffix Array construction [here](#). We will soon be discussing more efficient suffix array algorithms.

References:

<http://www.stanford.edu/class/cs97si/suffix-array.pdf>
http://en.wikipedia.org/wiki/Suffix_array

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

[Advanced Data Structure](#)
[Pattern Searching](#)
[Advance Data Structures](#)
[Advanced Data Structures](#)
[Pattern Searching](#)
[Suffix Array](#)

Suffix Array | Set 2 ($n \log n$ Algorithm)

A suffix array is a sorted array of all suffixes of a given string. The definition is similar to [Suffix Tree](#) which is compressed trie of all suffixes of the given text.

Let the given string be "banana".

```
0 banana          5 a
1 anana   Sort the Suffixes   3 ana
2 nana   ----->   1 anana
3 ana   alphabetically   0 banana
4 na           4 na
5 a           2 nana
```

The suffix array for "banana" is {5, 3, 1, 0, 4, 2}

We have discussed [Naive algorithm](#) for construction of suffix array. The Naive algorithm is to consider all suffixes, sort them using a $O(n \log n)$ sorting algorithm and while sorting, maintain original indexes. Time complexity of the Naive algorithm is $O(n^2 \log n)$ where n is the number of characters in the input string.

In this post, a $O(n \log n)$ algorithm for suffix array construction is discussed. Let us first discuss a $O(n * \log n * \log n)$ algorithm for simplicity. The idea is to use the fact that strings that are to be sorted are suffixes of a single string.

We first sort all suffixes according to first character, then according to first 2 characters, then first 4 characters and so on while the number of characters to be considered is smaller than $2n$. The important point is, if we

have sorted suffixes according to first 2^i characters, then we can sort suffixes according to first 2^{i+1} characters in $O(n \log n)$ time using a $n \log n$ sorting algorithm like Merge Sort. This is possible as two suffixes can be compared in $O(1)$ time (we need to compare only two values, see the below example and code).

The sort function is called $O(\log n)$ times (Note that we increase number of characters to be considered in powers of 2). Therefore overall time complexity becomes $O(n \log n \log n)$. See <http://www.stanford.edu/class/cs97si/suffix-array.pdf> for more details.

Let us build suffix array the example string "banana" using above algorithm.

Sort according to first two characters Assign a rank to all suffixes using ASCII value of first character. A simple way to assign rank is to do $\text{str}[i] - 'a'$ for i th suffix of $\text{str}[]$

| Index | Suffix | Rank |
|-------|--------|------|
| 0 | banana | 1 |
| 1 | anana | 0 |
| 2 | nana | 13 |
| 3 | ana | 0 |
| 4 | na | 13 |
| 5 | a | 0 |

For every character, we also store rank of next adjacent character, i.e., the rank of character at $\text{str}[i + 1]$ (This is needed to sort the suffixes according to first 2 characters). If a character is last character, we store next rank as -1

| Index | Suffix | Rank | Next Rank |
|-------|--------|------|-----------|
| 0 | banana | 1 | 0 |
| 1 | anana | 0 | 13 |
| 2 | nana | 13 | 0 |
| 3 | ana | 0 | 13 |
| 4 | na | 13 | 0 |
| 5 | a | 0 | -1 |

Sort all Suffixes according to rank and adjacent rank. Rank is considered as first digit or MSD, and adjacent rank is considered as second digit.

| Index | Suffix | Rank | Next Rank |
|-------|--------|------|-----------|
| 5 | a | 0 | -1 |
| 1 | anana | 0 | 13 |
| 3 | ana | 0 | 13 |
| 0 | banana | 1 | 0 |
| 2 | nana | 13 | 0 |
| 4 | na | 13 | 0 |

Sort according to first four character

Assign new ranks to all suffixes. To assign new ranks, we consider the sorted suffixes one by one. Assign 0 as new rank to first suffix. For assigning ranks to remaining suffixes, we consider rank pair of suffix just before the current suffix. If previous rank pair of a suffix is same as previous rank of suffix just before it, then assign it same rank. Otherwise assign rank of previous suffix plus one.

| Index | Suffix | Rank |
|-------|--------|--------------------------------------|
| 5 | a | 0 [Assign 0 to first] |
| 1 | anana | 1 (0, 13) is different from previous |
| 3 | ana | 1 (0, 13) is same as previous |
| 0 | banana | 2 (1, 0) is different from previous |
| 2 | nana | 3 (13, 0) is different from previous |
| 4 | na | 3 (13, 0) is same as previous |

For every suffix $\text{str}[i]$, also store rank of next suffix at $\text{str}[i + 2]$. If there is no next suffix at $i + 2$, we store next rank as -1

| Index | Suffix | Rank | Next Rank |
|-------|--------|------|-----------|
| 5 | a | 0 | -1 |
| 1 | anana | 1 | 1 |
| 3 | ana | 1 | 0 |
| 0 | banana | 2 | 3 |
| 2 | nana | 3 | 3 |
| 4 | na | 3 | -1 |

Sort all Suffixes according to rank and next rank.

| Index | Suffix | Rank | Next Rank |
|-------|--------|------|-----------|
| 5 | a | 0 | -1 |
| 3 | ana | 1 | 0 |
| 1 | anana | 1 | 1 |
| 0 | banana | 2 | 3 |
| 4 | na | 3 | -1 |
| 2 | nana | 3 | 3 |

```
// C++ program for building suffix array of a given text
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

// Structure to store information of a suffix
struct suffix
{
    int index; // To store original index
    int rank[2]; // To store ranks and next rank pair
};

// A comparison function used by sort() to compare two suffixes
// Compares two pairs, returns 1 if first pair is smaller
int cmp(struct suffix a, struct suffix b)
{
    return (a.rank[0] == b.rank[0]) ? (a.rank[1] < b.rank[1] ? 1 : 0) :
        (a.rank[0] < b.rank[0] ? 1 : 0);
}

// This is the main function that takes a string 'txt' of size n as an
// argument, builds and return the suffix array for the given string
int *buildSuffixArray(char *txt, int n)
{
    // A structure to store suffixes and their indexes
    struct suffix suffixes[n];

    // Store suffixes and their indexes in an array of structures.
    // The structure is needed to sort the suffixes alphabetically
    // and maintain their old indexes while sorting
    for (int i = 0; i < n; i++)
    {
        suffixes[i].index = i;
        suffixes[i].rank[0] = txt[i] - 'a';
        suffixes[i].rank[1] = ((i + 1) < n) ? (txt[i + 1] - 'a') : -1;
    }

    // Sort the suffixes using the comparison function
    // defined above.
    sort(suffixes, suffixes + n, cmp);
}
```

```

// At this point, all suffixes are sorted according to first
// 2 characters. Let us sort suffixes according to first 4
// characters, then first 8 and so on
int ind[n]; // This array is needed to get the index in suffixes[]
// from original index. This mapping is needed to get
// next suffix.
for (int k = 4; k < 2*n; k = k*2)
{
    // Assigning rank and index values to first suffix
    int rank = 0;
    int prev_rank = suffixes[0].rank[0];
    suffixes[0].rank[0] = rank;
    ind[suffixes[0].index] = 0;

    // Assigning rank to suffixes
    for (int i = 1; i < n; i++)
    {
        // If first rank and next ranks are same as that of previous
        // suffix in array, assign the same new rank to this suffix
        if (suffixes[i].rank[0] == prev_rank &&
            suffixes[i].rank[1] == suffixes[i-1].rank[1])
        {
            prev_rank = suffixes[i].rank[0];
            suffixes[i].rank[0] = rank;
        }
        else // Otherwise increment rank and assign
        {
            prev_rank = suffixes[i].rank[0];
            suffixes[i].rank[0] = ++rank;
        }
        ind[suffixes[i].index] = i;
    }

    // Assign next rank to every suffix
    for (int i = 0; i < n; i++)
    {
        int nextindex = suffixes[i].index + k/2;
        suffixes[i].rank[1] = (nextindex < n)?
            suffixes[nextindex].rank[0] : -1;
    }

    // Sort the suffixes according to first k characters
    sort(suffixes, suffixes+n, cmp);
}

// Store indexes of all sorted suffixes in the suffix array
int *suffixArr = new int[n];
for (int i = 0; i < n; i++)
    suffixArr[i] = suffixes[i].index;

// Return the suffix array
return suffixArr;
}

// A utility function to print an array of given size
void printArr(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver program to test above functions
int main()
{
    char txt[] = "banana";
    int n = strlen(txt);
    int *suffixArr = buildSuffixArray(txt, n);
    cout << "Following is suffix array for " << txt << endl;
    printArr(suffixArr, n);
    return 0;
}

```

Output:

```

Following is suffix array for banana
5 3 1 0 4 2

```

Note that the above algorithm uses standard sort function and therefore time complexity is $O(n \log n \log n)$. We can use [Radix Sort](#) here to reduce the time complexity to $O(n \log n)$.

Please note that suffix arrays can be constructed in $O(n)$ time also. We will soon be discussing $O(n)$ algorithms.

References:

<http://www.stanford.edu/class/cs97si/suffix-array.pdf>

<http://www.cbc.umd.edu/confcour/Fall2012/lec14b.pdf>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Pattern Searching
Advance Data Structures
Advanced Data Structures
Suffix Array

kasai's Algorithm for Construction of LCP array from Suffix Array

Background

Suffix Array : A suffix array is a sorted array of all suffixes of a given string.

Let the given string be "banana".

```

0 banana      5 a
1 anana  Sort the Suffixes  3 ana
2 nana  ----->  1 anana
3 ana  alphabetically  0 banana
4 na      4 na
5 a      2 nana

```

The suffix array for "banana" :

suffix[] = {5, 3, 1, 0, 4, 2}

We have discussed [Suffix Array](#) and its $O(n \log n)$ construction .

Once Suffix array is built, we can use it to efficiently search a pattern in a text. For example, we can use Binary Search to find a pattern (Complete code for the same is discussed [here](#))

LCP Array

The Binary Search based solution discussed [here](#) takes $O(m \log n)$ time where m is length of the pattern to be searched and n is length of the text. With the help of LCP array, we can search a pattern in $O(m + \log n)$ time. For example, if our task is to search "ana" in "banana", $m = 3$, $n = 5$.

LCP Array is an array of size n (like Suffix Array). A value $\text{lcp}[i]$ indicates length of the longest common prefix of the suffixes indexed by $\text{suffix}[i]$ and $\text{suffix}[i+1]$. $\text{suffix}[n-1]$ is not defined as there is no suffix after it.

```
txt[0..n-1] = "banana"
suffix[] = {5, 3, 1, 0, 4, 2}
lcp[]    = {1, 3, 0, 0, 2, 0}
```

Suffixes represented by suffix array in order are:
{ "a", "ana", "anana", "banana", "na", "nana" }

```
lcp[0] = Longest Common Prefix of "a" and "ana"    = 1
lcp[1] = Longest Common Prefix of "ana" and "anana" = 3
lcp[2] = Longest Common Prefix of "anana" and "banana" = 0
lcp[3] = Longest Common Prefix of "banana" and "na" = 0
lcp[4] = Longest Common Prefix of "na" and "nana" = 2
lcp[5] = Longest Common Prefix of "nana" and None = 0
```

How to construct LCP array?

LCP array construction is done two ways:

- 1) Compute the LCP array as a byproduct to the suffix array (Manber & Myers Algorithm)
- 2) Use an already constructed suffix array in order to compute the LCP values. (Kasai Algorithm).

There exist algorithms that can construct Suffix Array in $O(n)$ time and therefore we can always construct LCP array in $O(n)$ time. But in the below implementation, a $O(n \log n)$ algorithm is discussed.

Kasai's Algorithm

In this article Kasai's Algorithm is discussed. The algorithm constructs LCP array from suffix array and input text in $O(n)$ time. The idea is based on below fact:

Let lcp of suffix beginning at $\text{txt}[i]$ be k . If k is greater than 0, then lcp for suffix beginning at $\text{txt}[i+1]$ will be at least $k-1$. The reason is, relative order of characters remain same. If we delete the first character from both suffixes, we know that at least k characters will match. For example for substring "ana", lcp is 3, so for string "na" lcp will be at least 2. Refer [this](#) for proof.

Below is C++ implementation of Kasai's algorithm.

```
// C++ program for building LCP array for given text
#include <bits/stdc++.h>
using namespace std;

// Structure to store information of a suffix
struct suffix
{
    int index; // To store original index
    int rank[2]; // To store ranks and next rank pair
};

// A comparison function used by sort() to compare two suffixes
// Compares two pairs, returns 1 if first pair is smaller
int cmp(struct suffix a, struct suffix b)
{
    return (a.rank[0] == b.rank[0]) ? (a.rank[1] < b.rank[1] ? 1 : 0) :
        (a.rank[0] < b.rank[0] ? 1 : 0);
}

// This is the main function that takes a string 'txt' of size 'n' as an
// argument, builds and return the suffix array for the given string
vector<int> buildSuffixArray(string txt, int n)
{
    // A structure to store suffixes and their indexes
    struct suffix suffixes[n];

    // Store suffixes and their indexes in an array of structures.
    // The structure is needed to sort the suffixes alphabetically
    // and maintain their old indexes while sorting
    for (int i = 0; i < n; i++)
    {
        suffixes[i].index = i;
        suffixes[i].rank[0] = txt[i] - 'a';
        suffixes[i].rank[1] = ((i+1) < n) ? (txt[i+1] - 'a') : -1;
    }

    // Sort the suffixes using the comparison function
    // defined above.
    sort(suffixes, suffixes+n, cmp);

    // At this point, all suffixes are sorted according to first
    // 2 characters. Let us sort suffixes according to first 4
    // characters, then first 8 and so on
    int ind[n]; // This array is needed to get the index in suffixes[]
    // from original index. This mapping is needed to get
    // next suffix.
    for (int k = 4; k < 2*n; k = k*2)
    {
        // Assigning rank and index values to first suffix
        int rank = 0;
        int prev_rank = suffixes[0].rank[0];
        suffixes[0].rank[0] = rank;
        ind[suffixes[0].index] = 0;

        // Assigning rank to suffixes
        for (int i = 1; i < n; i++)
        {
            // If first rank and next ranks are same as that of previous
            // suffix in array, assign the same new rank to this suffix
            if (suffixes[i].rank[0] == prev_rank &&
                suffixes[i].rank[1] == suffixes[i-1].rank[1])
            {
                prev_rank = suffixes[i].rank[0];
                suffixes[i].rank[0] = rank;
            }
            else // Otherwise increment rank and assign
            {
                prev_rank = suffixes[i].rank[0];
                suffixes[i].rank[0] = ++rank;
            }
            ind[suffixes[i].index] = i;
        }

        // Assign next rank to every suffix
        for (int i = 0; i < n; i++)
```

```

{
    int nextindex = suffixes[i].index + k/2;
    suffixes[i].rank[1] = (nextindex < n)?
        suffixes[nextindex].rank[0] - 1;
}

// Sort the suffixes according to first k characters
sort(suffixes, suffixes+n, cmp);
}

// Store indexes of all sorted suffixes in the suffix array
vector<int> suffixArr;
for (int i = 0; i < n; i++)
    suffixArr.push_back(suffixes[i].index);

// Return the suffix array
return suffixArr;
}

/* To construct and return LCP */
vector<int> kasai(string txt, vector<int> suffixArr)
{
    int n = suffixArr.size();

    // To store LCP array
    vector<int> lcp(n, 0);

    // An auxiliary array to store inverse of suffix array
    // elements. For example if suffixArr[0] is 5, the
    // invSuff[5] would store 0. This is used to get next
    // suffix string from suffix array.
    vector<int> invSuff(n, 0);

    // Fill values in invSuff[]
    for (int i=0; i < n; i++)
        invSuff[suffixArr[i]] = i;

    // Initialize length of previous LCP
    int k = 0;

    // Process all suffixes one by one starting from
    // first suffix in txt[]
    for (int i=0; i<n; i++)
    {
        /* If the current suffix is at n-1, then we don't
        have next substring to consider. So lcp is not
        defined for this substring, we put zero. */
        if (invSuff[i] == n-1)
        {
            k = 0;
            continue;
        }

        /* j contains index of the next substring to
        be considered to compare with the present
        substring, i.e., next string in suffix array */
        int j = suffixArr[invSuff[i]+1];

        // Directly start matching from k'th index as
        // at-least k-1 characters will match
        while (i+k<n && j+k<n && txt[i+k]==txt[j+k])
            k++;

        lcp[invSuff[i]] = k; // lcp for the present suffix.

        // Deleting the starting character from the string.
        if (k>0)
            k--;
    }

    // return the constructed lcp array
    return lcp;
}

// Utility function to print an array
void printArr(vector<int> arr, int n)
{
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver program
int main()
{
    string str = "banana";

    vector<int> suffixArr = buildSuffixArray(str, str.length());
    int n = suffixArr.size();

    cout << "Suffix Array : \n";
    printArr(suffixArr, n);

    vector<int> lcp = kasai(str, suffixArr);

    cout << "\nLCP Array : \n";
    printArr(lcp, n);
    return 0;
}

```

Output:

Suffix Array :
5 3 1 0 4 2

LCP Array :
1 3 0 0 2 0

Illustration:

txt[] = "banana", suffix[] = {5, 3, 1, 0, 4, 2}

Suffix array represents
{ "a", "ana", "anana", "banana", "na", "nana" }

Inverse Suffix Array would be
 $\text{invSuff}[] = \{3, 2, 5, 1, 4, 0\}$

LCP values are evaluated in below order

We first compute LCP of first suffix in text which is **"banana"**. We need next suffix in suffix array to compute LCP (Remember lcp[i] is defined as Longest Common Prefix of suffix[i] and suffix[i+1]). **To find the next suffix in suffixArr[], we use Suffix[]**. The next suffix is "na". Since there is no common prefix between "banana" and "na", the value of LCP for "banana" is 0 and it is at index 3 in suffix array, so we fill lcp[3] as 0.

Next we compute LCP of second suffix which **"anana"**. Next suffix of "anana" in suffix array is "banana". Since there is no common prefix, the value of LCP for "anana" is 0 and it is at index 2 in suffix array, so we fill **lcp[2]** as 0.

Next we compute LCP of third suffix which "nana". Since there is no next suffix, the value of LCP for "nana" is not defined. We fill **lcp[5]** as 0.

Next suffix in text is "ana". Next suffix of "ana" in suffix array is "anana". Since there is a common prefix of length 3, the value of LCP for "ana" is 3. We fill `lcp[1]` as 3.

Now we lcp for next suffix in text which is **"na"**. This is where Kasai's algorithm uses the trick that LCP value must be at least 2 because previous LCP value was 3. Since there is no character after "na", final value of LCP is 2. We fill **lcp[4]** as 2.

Next suffix in text is "a". LCP value must be at least 1 because previous value was 2. Since there is no character after "a", final value of LCP is 1. We fill `lcp[0]` as 1.

We will soon be discussing implementation of search with the help of LCP array and how LCP array helps in reducing time complexity to $O(m + \log n)$.

References:

<http://web.stanford.edu/class/cs97si/suffix-array.pdf>

<http://www.mi.fu-berlin.de/wiki/pub/ABI/RnaSeqP4/suffix-array.pdf>

<http://codeforces.com/blog/entry/12796>

This article is contributed by **Prakhar Agrawal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Pattern Searching
Technical Scripter

Pattern Searching | Set 8 (Suffix Tree Introduction)

Given a text `txt[0..n-1]` and a pattern `pat[0..m-1]`, write a function `search(char pat[], char txt[])` that prints all occurrences of `pat[]` in `txt[]`. You may assume that $n > m$.

Preprocess Pattern or Preprocess Text?

We have discussed the following algorithms in the previous posts:

KMP Algorithm

Rabin Karp Algorithm

Finite Automata based Algorithm

Boyer Moore Algorithm

All of the above algorithms preprocess the pattern to make the pattern searching faster. The best time complexity that we could get by preprocessing pattern is $O(n)$ where n is length of the text. In this post, we will discuss an approach that preprocesses the text. A suffix tree is built of the text. After preprocessing text (building suffix tree of text), we can search any pattern in $O(m)$ time where m is length of the pattern.

Imagine you have stored complete work of **William Shakespeare** and preprocessed it. You can search any string in the complete work in time just proportional to length of the pattern. This is really a great improvement because length of pattern is generally much smaller than text.

Preprocessing of text may become costly if the text changes frequently. It is good for fixed text or less frequently changing text though.

A Suffix Tree for a given text is a compressed trie for all suffixes of the given text. We have discussed **Standard Trie**. Let us understand **Compressed Trie** with the following array of words.

{bear, bell, bid, bull, buy, sell, stock, stop}

Following is standard trie for the above input set of words.

Following is the compressed trie. Compress Trie is obtained from standard trie by joining chains of single nodes. The nodes of a compressed trie can be stored by storing index ranges at the nodes.

How to build a Suffix Tree for a given text?

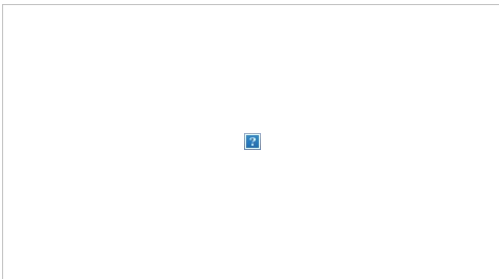
As discussed above, Suffix Tree is compressed trie of all suffixes, so following are very abstract steps to build a suffix tree from given text.

- 1) Generate all suffixes of given text.
- 2) Consider all suffixes as individual words and build a compressed trie.

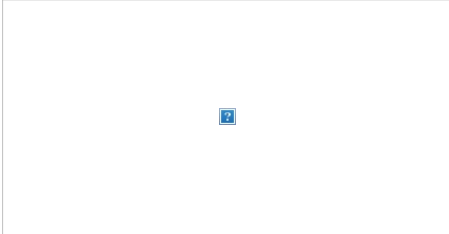
Let us consider an example text "banana\0" where "\0" is string termination character. Following are all suffixes of "banana\0"

```
banana\0
anana\0
nana\0
ana\0
na\0
a\0
\0
```

If we consider all of the above suffixes as individual words and build a trie, we get following.



If we join chains of single nodes, we get the following compressed trie, which is the Suffix Tree for given text "banana|0"



Please note that above steps are just to manually create a Suffix Tree. We will be discussing actual algorithm and implementation in a separate post.

How to search a pattern in the built suffix tree?

We have discussed above how to build a Suffix Tree which is needed as a preprocessing step in pattern searching. Following are abstract steps to search a pattern in the built Suffix Tree.

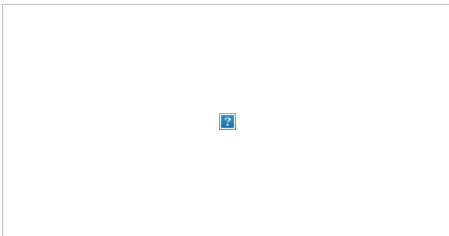
1) Starting from the first character of the pattern and root of Suffix Tree, do following for every character.

.....a) For the current character of pattern, if there is an edge from the current node of suffix tree, follow the edge.

.....b) If there is no edge, print "pattern doesn't exist in text" and return.

2) If all characters of pattern have been processed, i.e., there is a path from root for characters of the given pattern, then print "Pattern found".

Let us consider the example pattern as "nan" to see the searching process. Following diagram shows the path followed for searching "nan" or "nana".



How does this work?

Every pattern that is present in text (or we can say every substring of text) must be a prefix of one of all possible suffixes. The statement seems complicated, but it is a simple statement, we just need to take an example to check validity of it.

Applications of Suffix Tree

Suffix tree can be used for a wide range of problems. Following are some famous problems where Suffix Trees provide optimal time complexity solution.

- 1) Pattern Searching
- 2) Finding the longest repeated substring
- 3) Finding the longest common substring
- 4) Finding the longest palindrome in a string

There are many more applications. See this for more details.

Ukkonen's Suffix Tree Construction is discussed in following articles:

- Ukkonen's Suffix Tree Construction – Part 1
- Ukkonen's Suffix Tree Construction – Part 2
- Ukkonen's Suffix Tree Construction – Part 3
- Ukkonen's Suffix Tree Construction – Part 4
- Ukkonen's Suffix Tree Construction – Part 5
- Ukkonen's Suffix Tree Construction – Part 6

References:

- <http://fbim.fh-regensburg.de/~saj39122/sal/skript/progr/pr45102/Tries.pdf>
- <http://www.cs.ucf.edu/~shzhang/Combio12/lec3.pdf>
- <http://www.allisons.org/ll/AlgDS/Tree/Suffix/>

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Pattern Searching
Advance Data Structures
Advanced Data Structures
Pattern Searching
Suffix Tree

Ukkonen's Suffix Tree Construction – Part 1

Suffix Tree is very useful in numerous string processing and computational biology problems. Many books and e-resources talk about it theoretically and in few places, code implementation is discussed. But still, I felt something is missing and it's not easy to implement code to construct suffix tree and it's usage in many applications. This is an attempt to bridge the gap between theory and complete working code implementation. Here we will discuss Ukkonen's Suffix Tree Construction Algorithm. We will discuss it in step by step detailed way and in multiple parts from theory to implementation. We will start with brute force way and try to understand different concepts, tricks involved in Ukkonen's algorithm and in the last part, code implementation will be discussed.

Note: You may find some portion of the algorithm difficult to understand while 1st or 2nd reading and it's perfectly fine. With few more attempts and thought, you should be able to understand such portions.

Book [Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology](#) by **Dan Gusfield** explains the concepts very well.

A suffix tree **T** for a m-character string **S** is a rooted directed tree with exactly **m** leaves numbered 1 to **m**. (Given that last string character is unique in string)

- Root can have zero, one or more children.
- Each internal node, other than the root, has at least two children.
- Each edge is labelled with a nonempty substring of **S**.
- No two edges coming out of same node can have edge-labels beginning with the same character.

Concatenation of the edge-labels on the path from the root to leaf **i** gives the suffix of **S** that starts at position **i**, i.e. **S[i...m]**.

Note: Position starts with 1 (it's not zero indexed, but later, while code implementation, we will use zero indexed position)

For string $S = \text{xabxac}$ with $m = 6$, suffix tree will look like following:

Ukkonen's Suffix Tree Construction



It has one root node and two internal nodes and 6 leaf nodes.

String Depth of **red** path is 1 and it represents suffix c starting at position 6

String Depth of **blue** path is 4 and it represents suffix bxca starting at position 3

String Depth of **green** path is 2 and it represents suffix ac starting at position 5

String Depth of **orange** path is 6 and it represents suffix xabxac starting at position 1

Edges with labels **a** (**green**) and **xa** (**orange**) are non-leaf edge (which ends at an internal node). All other edges are leaf edge (ends at a leaf)

If one suffix of S matches a prefix of another suffix of S (when last character in not unique in string), then path for the first suffix would not end at a leaf.

For String $S = \text{xabxa}$, with $m = 5$, following is the suffix tree:

Ukkonen's Suffix Tree Construction



Here we will have 5 suffixes: xabxa , abxa , bxa , xa and a .

Path for suffixes ' xa ' and ' a ' do not end at a leaf. A tree like above (Figure 2) is called implicit suffix tree as some suffixes (' xa ' and ' a ') are not seen explicitly in tree.

To avoid this problem, we add a character which is not present in string already. We normally use $\$$, $\#$ etc as termination characters.

Following is the suffix tree for string $S = \text{xabxa\$}$ with $m = 6$ and now all 6 suffixes end at leaf.

Ukkonen's Suffix Tree Construction



A naive algorithm to build a suffix tree

Given a string S of length m , enter a single edge for suffix $S[1..m]$ (the entire string) into the tree, then successively enter suffix $S[i..m]$ into the growing tree, for i increasing from 2 to m . Let N_i denote the intermediate tree that encodes all the suffixes from 1 to i .

So N_{i+1} is constructed from N_i as follows:

- Start at the root of N_i
- Find the longest path from the root which matches a prefix of $S[i+1..m]$
- Match ends either at the node (say w) or in the middle of an edge [say (u, v)].
- If it is in the middle of an edge (u, v) , break the edge (u, v) into two edges by inserting a new node w just after the last character on the edge that matched a character in $S[i+1..m]$ and just before the first character on the edge that mismatched. The new edge (u, w) is labelled with the part of the (u, v) label that matched with $S[i+1..m]$, and the new edge (w, v) is labelled with the remaining part of the (u, v) label.
- Create a new edge $(w, i+1)$ from w to a new leaf labelled $i+1$ and it labels the new edge with the unmatched part of suffix $S[i+1..m]$

This takes $O(m^2)$ to build the suffix tree for the string S of length m .

Following are few steps to build suffix tree based for string " $\text{xabxa\$}$ " based on above algorithm:

Ukkonen's Suffix Tree Construction



Ukkonen's Suffix Tree Construction



Ukkonen's Suffix Tree Construction



Ukkonen's Suffix Tree Construction



Implicit suffix tree

While generating suffix tree using Ukkonen's algorithm, we will see implicit suffix tree in intermediate steps few times depending on characters in string S . In implicit suffix trees, there will be no edge with $\$$ (or $\#$ or any other termination character) label and no internal node with only one edge going out of it.

To get implicit suffix tree from a suffix tree SS ,

- Remove all terminal symbol $\$$ from the edge labels of the tree,
- Remove any edge that has no label
- Remove any node that has only one edge going out of it and merge the edges.

Ukkonen's Suffix Tree Construction



High Level Description of Ukkonen's algorithm

Ukkonen's algorithm constructs an implicit suffix tree T_i for each prefix $S[1..i]$ of S (of length m).

It first builds T_1 using 1st character, then T_2 using 2nd character, then T_3 using 3rd character, ..., T_m using m^{th} character.

Implicit suffix tree T_{i+1} is built on top of implicit suffix tree T_i .

The true suffix tree for S is built from T_m by adding $\$$.

At any time, Ukkonen's algorithm builds the suffix tree for the characters seen so far and so it has **on-line** property that may be useful in some situations.

Time taken is $O(m)$.

Ukkonen's algorithm is divided into m phases (one phase for each character in the string with length m)

In phase $i+1$, tree T_{i+1} is built from tree T_i .

Each phase $i+1$ is further divided into $i+1$ extensions, one for each of the $i+1$ suffixes of $S[1..i+1]$

In extension j of phase $i+1$, the algorithm first finds the end of the path from the root labelled with substring $S[j..i]$.

It then extends the substring by adding the character $S[i+1]$ to its end (if it is not there already).

In extension 1 of phase $i+1$, we put string $S[1..i+1]$ in the tree. Here $S[1..i]$ will already be present in tree due to previous phase i . We just need to add $S[i+1]$ th character in tree (if not there already).

In extension 2 of phase $i+1$, we put string $S[2..i+1]$ in the tree. Here $S[2..i]$ will already be present in tree due to previous phase i . We just need to add $S[i+1]$ th character in tree (if not there already).

In extension 3 of phase $i+1$, we put string $S[3..i+1]$ in the tree. Here $S[3..i]$ will already be present in tree due to previous phase i . We just need to add $S[i+1]$ th character in tree (if not there already).

.

In extension $i+1$ of phase $i+1$, we put string $S[i+1..i+1]$ in the tree. This is just one character which may not be in tree (if character is seen first time so far). If so, we just add a new leaf edge with label $S[i+1]$.

High Level Ukkonen's algorithm

Construct tree T_1

For i from 1 to $m-1$ do

begin {phase $i+1$ }

For j from 1 to $i+1$

begin {extension j }

Find the end of the path from the root labelled $S[j..i]$ in the current tree.

Extend that path by adding character $S[i+1]$ if it is not there already

end;

end;

Suffix extension is all about adding the next character into the suffix tree built so far.

In extension j of phase $i+1$, algorithm finds the end of $S[j..i]$ (which is already in the tree due to previous phase i) and then it extends $S[j..i]$ to be sure the suffix $S[j..i+1]$ is in the tree.

There are 3 extension rules:

Rule 1: If the path from the root labelled $S[j..i]$ ends at leaf edge (i.e. $S[i]$ is last character on leaf edge) then character $S[i+1]$ is just added to the end of the label on that leaf edge.

Rule 2: If the path from the root labelled $S[j..i]$ ends at non-leaf edge (i.e. there are more characters after $S[i]$ on path) and next character is not $s[i+1]$, then a new leaf edge with label $s[i+1]$ and number j is created starting from character $S[i+1]$.

A new internal node will also be created if $s[1..i]$ ends inside (in-between) a non-leaf edge.

Rule 3: If the path from the root labelled $S[j..i]$ ends at non-leaf edge (i.e. there are more characters after $S[i]$ on path) and next character is $s[i+1]$ (already in tree), do nothing.

One important point to note here is that from a given node (root or internal), there will be one and only one edge starting from one character. There will not be more than one edges going out of any node, starting with same character.

Following is a step by step suffix tree construction of string xabxac using Ukkonen's algorithm:

Ukkonen's Suffix Tree Construction



Ukkonen's Suffix Tree Construction



Ukkonen's Suffix Tree Construction



Ukkonen's Suffix Tree Construction



Ukkonen's Suffix Tree Construction



Ukkonen's Suffix Tree Construction



In next parts ([Part 2](#), [Part 3](#), [Part 4](#) and [Part 5](#)), we will discuss suffix links, active points, few tricks and finally code implementations ([Part 6](#)).

References:

<http://web.stanford.edu/~mjkay/gusfield.pdf>

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Pattern Searching
Pattern Searching
Suffix Tree

Ukkonen's Suffix Tree Construction – Part 2

In [Ukkonen's Suffix Tree Construction – Part 1](#), we have seen high level Ukkonen's Algorithm. This 2nd part is continuation of [Part 1](#).

Please go through [Part 1](#), before looking at current article.

In Suffix Tree Construction of string S of length m , there are m phases and for a phase j ($1 \leq j \leq m$), we add j^{th} character in tree built so far and this is done through j extensions. All extensions follow one of the three extension rules (discussed in [Part 1](#)).

To do j^{th} extension of phase $i+1$ (adding character $S[i+1]$), we first need to find end of the path from the root labelled $S[1..i]$ in the current tree. One way is start from root and traverse the edges matching $S[1..i]$ string. This will take $O(m^3)$ time to build the suffix tree. Using few observations and implementation tricks, it can be done in $O(m)$ which we will see now.

Suffix links

For an internal node v with path-label xA , where x denotes a single character and A denotes a (possibly empty) substring, if there is another node $s(v)$ with path-label A , then a pointer from v to $s(v)$ is called a suffix link.

If A is empty string, suffix link from internal node will go to root node.

There will not be any suffix link from root node (As it's not considered as internal node).

Ukkonen's Suffix Tree Construction



In extension j of some phase i , if a new internal node v with path-label xA is added, then in extension $j+1$ in the same phase i :

- Either the path labelled A already ends at an internal node (or root node if A is empty)
- OR a new internal node at the end of string A will be created

In extension $j+1$ of same phase i , we will create a suffix link from the internal node created in j^{th} extension to the node with path labelled A .

So in a given phase, any newly created internal node (with path-label xA) will have a suffix link from it (pointing to another node with path-label A) by the end of the next extension.

In any implicit suffix tree T_i after phase i , if internal node v has path-label xA , then there is a node $s(v)$ in T_i with path-label A and node v will point to node $s(v)$ using suffix link.

At any time, all internal nodes in the changing tree will have suffix links from them to another internal node (or root) except for the most recently added internal node, which will receive its suffix link by the end of the next extension.

How suffix links are used to speed up the implementation?

In extension j of phase $i+1$, we need to find the end of the path from the root labelled $S[j..i]$ in the current tree. One way is start from root and traverse the edges matching $S[j..i]$ string. Suffix links provide a short cut to find end of the path.

Ukkonen's Suffix Tree Construction



Ukkonen's Suffix Tree Construction



So we can see that, to find end of path $S[j..i]$, we need not traverse from root. We can start from the end of path $S[j-1..i]$, walk up one edge to node v (i.e. go to parent node), follow the suffix link to $s(v)$, then walk down the path y (which is $abcd$ here in Figure 17).

This shows the use of suffix link is an improvement over the process.

Note: In the next part 3, we will introduce activePoint which will help to avoid "walk up". We can directly go to node $s(v)$ from node v .

When there is a suffix link from node v to node $s(v)$, then if there is a path labelled with string y from node v to a leaf, then there must be a path labelled with string y from node $s(v)$ to a leaf. In Figure 17, there is a path label "abcd" from node v to a leaf, then there is a path with same label "abcd" from node $s(v)$ to a leaf.

This fact can be used to improve the walk from $s(v)$ to leaf along the path y . This is called "skip/count" trick.

Skip/Count Trick

When walking down from node $s(v)$ to leaf, instead of matching path character by character as we travel, we can directly skip to the next node if number of characters on the edge is less than the number of characters we need to travel. If number of characters on the edge is more than the number of characters we need to travel, we directly skip to the last character on that edge.

If implementation is such a way that number of characters on any edge, character at a given position in string S should be obtained in constant time, then skip/count trick will do the walk down in proportional to the number of nodes on it rather than the number of characters on it.

Ukkonen's Suffix Tree Construction



Using suffix link along with skip/count trick, suffix tree can be built in $O(m^2)$ as there are m phases and each phase takes $O(m)$.

Edge-label compression

So far, path labels are represented as characters in string. Such a suffix tree will take $O(m^2)$ space to store the path labels. To avoid this, we can use two pair of indices (start, end) on each edge for path labels, instead of substring itself. The indices start and end tells the path label start and end position in string S . With this, suffix tree needs $O(m)$ space.

Ukkonen's Suffix Tree Construction



There are two observations about the way extension rules interact in successive extensions and phases. These two observations lead to two more implementation tricks (first trick "skip/count" is seen already while walk down).

Observation 1: Rule 3 is show stopper

In a phase i , there are i extensions (1 to i) to be done.

When rule 3 applies in any extension j of phase $i+1$ (i.e. path labelled $S[j..i]$ continues with character $S[i+1]$), then it will also apply in all further extensions of same phase (i.e. extensions $j+1$ to $i+1$ in phase $i+1$). That's because if path labelled $S[j..i]$ continues with character $S[i+1]$, then path labelled $S[j+1..i]$, $S[j+2..i]$, $S[j+3..i]$, ..., $S[i..i]$ will also continue with character $S[i+1]$.

Consider Figure 11, Figure12 and Figure 13 in Part 1 where Rule 3 is applied.

In Figure 11, "xab" is added in tree and in Figure 12 (Phase 4), we add next character "x". In this, 3 extensions are done (which adds 3 suffixes). Last suffix "x" is already present in tree.

In Figure 13, we add character "a" in tree (Phase 5). First 3 suffixes are added in tree and last two suffixes "xa" and "a" are already present in tree. This shows that if suffix $S[j..i]$ present in tree, then ALL the remaining suffixes $S[j+1..i]$, $S[j+2..i]$, $S[j+3..i]$, ..., $S[i..i]$ will also be there in tree and no work needed to add those remaining suffixes.

So no more work needed to be done in any phase as soon as rule 3 applies in any extension in that phase. If a new internal node v gets created in extension j and rule 3 applies in next extension $j+1$, then we need to add suffix link from node v to current node (if we are on internal node) or root node. **ActiveNode**, which will be discussed in part 3, will help while setting suffix links.

Trick 2

Stop the processing of any phase as soon as rule 3 applies. All further extensions are already present in tree implicitly.

Observation 2: Once a leaf, always a leaf

Once a leaf is created and labelled j (for suffix starting at position j in string S), then this leaf will always be a leaf in successive phases and extensions. Once a leaf is labelled as j , extension rule 1 will always apply to extension j in all successive phases.

Consider Figure 9 to Figure 14 in [Part 1](#).

In Figure 10 (Phase 2), Rule 1 is applied on leaf labelled 1. After this, in all successive phases, rule 1 is always applied on this leaf.

In Figure 11 (Phase 3), Rule 1 is applied on leaf labelled 2. After this, in all successive phases, rule 1 is always applied on this leaf.

In Figure 12 (Phase 4), Rule 1 is applied on leaf labelled 3. After this, in all successive phases, rule 1 is always applied on this leaf.

In any phase i , there is an initial sequence of consecutive extensions where rule 1 or rule 2 are applied and then as soon as rule 3 is applied, phase i ends.

Also rule 2 creates a new leaf always (and internal node sometimes).

If J_i represents the last extension in phase i when rule 1 or 2 was applied (i.e after i^{th} phase, there will be J_i leaves labelled 1, 2, 3, ..., J_i), then $J_i \leq J_{i+1}$

J_i will be equal to J_{i+1} when there are no new leaf created in phase $i+1$ (i.e rule 3 is applied in J_{i+1} extension)

In Figure 11 (Phase 3), Rule 1 is applied in 1st two extensions and Rule 2 is applied in 3rd extension, so here $J_3 = 3$

In Figure 12 (Phase 4), no new leaf created (Rule 1 is applied in 1st 3 extensions and then rule 3 is applied in 4th extension which ends the phase). Here $J_4 = 3 = J_3$

In Figure 13 (Phase 5), no new leaf created (Rule 1 is applied in 1st 3 extensions and then rule 3 is applied in 4th extension which ends the phase). Here $J_5 = 3 = J_4$

J_i will be less than J_{i+1} when few new leaves are created in phase $i+1$.

In Figure 14 (Phase 6), new leaf created (Rule 1 is applied in 1st 3 extensions and then rule 2 is applied in last 3 extension which ends the phase). Here $J_6 = 6 > J_5$

So we can see that in phase $i+1$, only rule 1 will apply in extensions 1 to J_i (which really doesn't need much work, can be done in constant time and that's the trick 3), extension J_{i+1} onwards, rule 2 may apply to zero or more extensions and then finally rule 3, which ends the phase.

Now edge labels are represented using two indices (start, end), for any leaf edge, end will always be equal to phase number i.e. for phase i , end = i for leaf edges, for phase $i+1$, end = $i+1$ for leaf edges.

Trick 3

In any phase i , leaf edges may look like (p, i) , (q, i) , (r, i) , where p, q, r are starting position of different edges and i is end position of all. Then in phase $i+1$, these leaf edges will look like $(p, i+1)$, $(q, i+1)$, $(r, i+1)$, This way, in each phase, end position has to be incremented in all leaf edges. For this, we need to traverse through all leaf edges and increment end position for them. To do same thing in constant time, maintain a global index e and e will be equal to phase number. So now leaf edges will look like (p, e) , (q, e) , (r, e) .. In any phase, just increment e and extension on all leaf edges will be done. Figure 19 shows this.

So using suffix links and tricks 1, 2 and 3, a suffix tree can be built in linear time.

Tree T_m could be implicit tree if a suffix is prefix of another. So we can add a \$ terminal symbol first and then run algorithm to get a true suffix tree (A true suffix tree contains all suffixes explicitly). To label each leaf with corresponding suffix starting position (all leaves are labelled as global index e), a linear time traversal can be done on tree.

At this point, we have gone through most of the things we needed to know to create suffix tree using Ukkonen's algorithm. In next [Part 3](#), we will take string $S = \text{"abcabxabcd"}$ as an example and go through all the things step by step and create the tree. While building the tree, we will discuss few more implementation issues which will be addressed by **ActivePoints**.

We will continue to discuss the algorithm in [Part 4](#) and [Part 5](#). Code implementation will be discussed in [Part 6](#).

References:

<http://web.stanford.edu/~mjkay/gusfield.pdf>

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Pattern Searching
Suffix Tree

Ukkonen's Suffix Tree Construction – Part 3

This article is continuation of following two articles:

[Ukkonen's Suffix Tree Construction – Part 1](#)

[Ukkonen's Suffix Tree Construction – Part 2](#)

Please go through [Part 1](#) and [Part 2](#), before looking at current article, where we have seen few basics on suffix tree, high level ukkonen's algorithm, suffix link and three implementation tricks.

Here we will take string $S = \text{"abcabxabcd"}$ as an example and go through all the things step by step and create the tree.

We will add \$ (discussed in [Part 1](#) why we do this) so string S would be $\text{"abcabxabcd\$"}$.

While building suffix tree for string S of length m :

- There will be m phases 1 to m (one phase for each character)
- In our current example, m is 11, so there will be 11 phases.
- First phase will add first character 'a' in the tree, second phase will add second character 'b' in tree, third phase will add third character 'c' in tree,, m^{th} phase will add m^{th} character in tree (This makes Ukkonen's algorithm an online algorithm)
- Each phase i will go through at-most i extensions (from 1 to i). If current character being added in tree is not seen so far, all i extensions will be completed (Extension Rule 3 will not apply in this phase). If current character being added in tree is seen before, then phase i will complete early (as soon as Extension Rule 3 applies) without going through all i extensions
- There are three extension rules (1, 2 and 3) and each extension j (from 1 to i) of any phase i will adhere to one of these three rules.
- Rule 1 adds a new character on existing leaf edge
- Rule 2 creates a new leaf edge (And may also create new internal node, if the path label ends in between an edge)
- Rule 3 ends the current phase (when current character is found in current edge being traversed)
- Phase 1 will read first character from the string, will go through 1 extension.

(In figures, we are showing characters on edge labels just for explanation, while writing code, we will only use start and end indices – The Edge-label compression discussed in [Part 2](#))

Extension 1 will add suffix "a" in tree. We start from root and traverse path with label 'a'. There is no path from root, going out with label 'a', so create a leaf edge (Rule 2).

Ukkonen's Suffix Tree Construction



Phase 1 completes with the completion of extension 1 (As a phase i has at most i extensions)

For any string, Phase 1 will have only one extension and it will always follow Rule 2.

- Phase 2 will read second character, will go through at least 1 and at most 2 extensions.
- In our example, phase 2 will read second character 'b'. Suffixes to be added are "ab" and "b".
- Extension 1 adds suffix "ab" in tree.
- Path for label 'a' ends at leaf edge, so add 'b' at the end of this edge.
- Extension 1 just increments the end index by 1 (from 1 to 2) on first edge (Rule 1).

Ukkonen's Suffix Tree Construction



Extension 2 adds suffix "b" in tree. There is no path from root, going out with label 'b', so creates a leaf edge (Rule 2).

Ukkonen's Suffix Tree Construction



Phase 2 completes with the completion of extension 2.

Phase 2 went through two extensions here. Rule 1 applied in 1st Extension and Rule 2 applied in 2nd Extension.

- Phase 3 will read third character, will go through at least 1 and at most 3 extensions.
In our example, phase 3 will read third character 'c'. Suffixes to be added are "abc", "bc" and "c".

Extension 1 adds suffix "abc" in tree.

Path for label 'ab' ends at leaf edge, so add 'c' at the end of this edge.

Extension 1 just increments the end index by 1 (from 2 to 3) on this edge (Rule 1).

Ukkonen's Suffix Tree Construction



Extension 2 adds suffix "bc" in tree.

Path for label 'b' ends at leaf edge, so add 'c' at the end of this edge.

Extension 2 just increments the end index by 1 (from 2 to 3) on this edge (Rule 1).

Ukkonen's Suffix Tree Construction



Extension 3 adds suffix "c" in tree. There is no path from root, going out with label 'c', so creates a leaf edge (Rule 2).

Ukkonen's Suffix Tree Construction



Phase 3 completes with the completion of extension 3.

Phase 3 went through three extensions here. Rule 1 applied in first two Extensions and Rule 2 applied in 3rd Extension.

- Phase 4 will read fourth character, will go to at least 1 and at most 4 extensions.
In our example, phase 4 will read fourth character 'a'. Suffixes to be added are "abca", "bca", "ca" and "a".

Extension 1 adds suffix "abca" in tree.

Path for label 'abc' ends at leaf edge, so add 'a' at the end of this edge.

Extension 1 just increments the end index by 1 (from 3 to 4) on this edge (Rule 1).

Ukkonen's Suffix Tree Construction



Extension 2 adds suffix "bca" in tree.

Path for label 'bc' ends at leaf edge, so add 'a' at the end of this edge.

Extension 2 just increments the end index by 1 (from 3 to 4) on this edge (Rule 1).

Ukkonen's Suffix Tree Construction



Extension 3 adds suffix "ca" in tree.

Path for label 'c' ends at leaf edge, so add 'a' at the end of this edge.

Extension 3 just increments the end index by 1 (from 3 to 4) on this edge (Rule 1).

Ukkonen's Suffix Tree Construction



Extension 4 adds suffix "a" in tree.

Path for label 'a' exists in the tree. No more work needed and Phase 4 ends here (Rule 3 and Trick 2). This is an example of implicit suffix tree. Here suffix "a" is not seen explicitly (because it doesn't end at a leaf edge) but it is in the tree implicitly. So there is no change in tree structure after extension 4. It will remain as above in Figure 28.

Phase 4 completes as soon as Rule 3 is applied while Extension 4.

Phase 4 went through four extensions here. Rule 1 applied in first three Extensions and Rule 3 applied in 4th Extension.

Now we will see few observations and how to implement those.

- At the end of any phase i , there are at most i leaf edges (if i^{th} character is not seen so far, there will be i leaf edges, else there will be less than i leaf edges).
e.g. After phases 1, 2 and 3 in our example, there are 1, 2 and 3 leaf edges respectively, but after phase 4, there are 3 leaf edges only (not 4).
- After completing phase i , "end" indices of all leaf edges are i . How do we implement this in code? Do we need to iterate through all those extensions, find leaf edges by traversing from root to leaf and increment the "end" index? Answer is "NO".

For this, we will maintain a global variable (say "END") and we will just increment this global variable "END" and all leaf edge end indices will point to this global variable. So this way, if we have j leaf edges after phase i , then in phase $i+1$, first j extensions (1 to j) will be done by just incrementing variable "END" by 1 (END will be $i+1$ at the point).

Here we just implemented the trick 3 – **Once a leaf, always a leaf**. This trick processes all the j leaf edges (i.e. extension 1 to j) using rule 1 in a constant time in any phase. Rule 1 will not apply to subsequent extensions in the same phase. This can be verified in the four phases we discussed above. If at all Rule 1 applies in any phase, it only applies in initial few phases continuously (say 1 to j). Rule 1 never applies later in a given phase once Rule 2 or Rule 3 is applied in that phase.

- In the example explained so far, in each extension (where trick 3 is not applied) of any phase to add a suffix in tree, we are traversing from root by matching path labels against the suffix being added. If there are j leaf edges after phase i , then in phase $i+1$, first j extensions will follow Rule 1 and will be done in constant time using trick 3. There are $i+1-j$ extensions yet to be performed. For these extensions, which node (root or some other internal node) to start from and which path to go? Answer to this depends on how previous phase i is completed.

If previous phase i went through all the i extensions (when i^{th} character is unique so far), then in next phase $i+1$, trick 3 will take care of first i suffixes (the i leaf edges) and then extension $i+1$ will start from root node and it will insert just one character $[(i+1)^{\text{th}}]$ suffix in tree by creating a leaf edge using Rule 2.

If previous phase i completes early (and this will happen if and only if rule 3 applies – when i^{th} character is already seen before), say at j^{th} extension (i.e. rule 3 is applied at j^{th} extension), then there are $j-1$ leaf edges so far.

We will state few more facts (which may be a repeat, but we want to make sure it's clear to you at this point) here based on discussion so far:

- Phase 1 starts with Rule 2, all other phases start with Rule 1
- Any phase ends with either Rule 2 or Rule 3
- Any phase i may go through a series of j extensions ($1 \leq j \leq i$). In these j extensions, first p ($0 \leq p < i$) extensions will follow Rule 1, next q ($0 \leq q \leq i-p$) extensions will follow Rule 2 and next r ($0 \leq r \leq 1$) extensions will follow Rule 3. The order in which Rule 1, Rule 2 and Rule 3 apply, is never intermixed in a phase. They apply in order of their number (if at all applied), i.e. in a phase, Rule 1 applies 1st, then Rule 2 and then Rule 3
- In a phase i , $p + q + r \leq i$
- At the end of any phase i , there will be $p+q$ leaf edges and next phase $i+1$ will go through Rule 1 for first $p+q$ extensions

In the next phase $i+1$, trick 3 (Rule 1) will take care of first $j-1$ suffixes (the $j-1$ leaf edges), then extension j will start where we will add j^{th} suffix in tree. For this, we need to find the best possible matching edge and then add new character at the end of that edge. How to find the end of best matching edge? Do we need to traverse from root node and match tree edges against the j^{th} suffix being added character by character? This will take time and overall algorithm will not be linear. activePoint comes to the rescue here.

In previous phase i , while j^{th} extension, path traversal ended at a point (which could be an internal node or some point in the middle of an edge) where i^{th} character being added was found in tree already and Rule 3 applied, j^{th} extension of phase $i+1$ will start exactly from the same point and we start matching path against $(i+1)^{\text{th}}$ character. activePoint helps to avoid unnecessary path traversal from root in any extension based on the knowledge gained in traversals done in previous extension. There is no traversal needed in 1st p extensions where Rule 1 is applied. Traversal is done where Rule 2 or Rule 3 gets applied and that's where activePoint tells the starting point for traversal where we match the path against the current character being added in tree. Implementation is done in such a way that, in any extension where we need a traversal, activePoint is set to right location already (with one exception case **APCFALZ** discussed below) and at the end of current extension, we reset activePoint as appropriate so that next extension (of same phase or next phase) where a traversal is required, activePoint points to the right place already.

activePoint: This could be root node, any internal node or any point in the middle of an edge. This is the point where traversal starts in any extension. For the 1st extension of phase 1, activePoint is set to root. Other extension will get activePoint set correctly by previous extension (with one exception case **APCFALZ** discussed below) and it is the responsibility of current extension to reset activePoint appropriately at the end, to be used in next extension where Rule 2 or Rule 3 is applied (of same or next phase).

To accomplish this, we need a way to store activePoint. We will store this using three variables: **activeNode**, **activeEdge**, **activeLength**.

activeNode: This could be root node or an internal node.

activeEdge: When we are on root node or internal node and we need to walk down, we need to know which edge to choose. activeEdge will store that information. In case, activeNode itself is the point from where traversal starts, then activeEdge will be set to next character being processed in next phase.

activeLength: This tells how many characters we need to walk down (on the path represented by activeEdge) from activeNode to reach the activePoint where traversal starts. In case, activeNode itself is the point from where traversal starts, then activeLength will be ZERO.

(click on below image to see it clearly)

Ukkonen's Suffix Tree Construction



After phase i , if there are j leaf edges then in phase $i+1$, first j extensions will be done by trick 3. activePoint will be needed for the extensions from $j+1$ to $i+1$ and activePoint may or may not change between two extensions depending on the point where previous extension ends.

activePoint change for extension rule 3 (APCFER3): When rule 3 applies in any phase i , then before we move on to next phase $i+1$, we increment activeLength by 1. There is no change in activeNode and activeEdge. Why? Because in case of rule 3, the current character from string S is matched on the same path represented by current activePoint, so for next activePoint, activeNode and activeEdge remain the same, only activeLength is increased by 1 (because of matched character in current phase). This new activePoint (same node, same edge and incremented length) will be used in phase $i+1$.

activePoint change for walk down (APCFWD): activePoint may change at the end of an extension based on extension rule applied. activePoint may also change during the extension when we do walk down. Let's consider an activePoint is (A, s, 11) in the above activePoint example figure. If this is the activePoint at the start of some extension, then while walk down from activeNode A, other internal nodes will be seen. Anytime if we encounter an internal node while walk down, that node will become activeNode (it will change activeEdge and activeLength as appropriate so that new activePoint represents the same point as earlier). In this walk down, below is the sequence of changes in activePoint:

(A, s, 11) —>>> (B, w, 7) —>>> (C, a, 3)

All above three activePoints refer to same point 'c'

Let's take another example.

If activePoint is (D, a, 11) at the start of an extension, then while walk down, below is the sequence of changes in activePoint:

(D, a, 10) —>>> (E, d, 7) —>>> (F, f, 5) —>> (G, j, 1)

All above activePoints refer to same point 'k'.

If activePoints are (A, s, 3), (A, t, 5), (B, w, 1), (D, a, 2) etc when no internal node comes in the way while walk down, then there will be no change in activePoint for APCFWD.

The idea is that, at any time, the closest internal node from the point, where we want to reach, should be the activePoint. Why? This will minimize the length of traversal in the next extension.

activePoint change for Active Length ZERO (APCFALZ): Let's consider an activePoint (A, s, 0) in the above activePoint example figure. And let's say current character being processed from string S is 'x' (or any other character). At the start of extension, when activeLength is ZERO, activeEdge is set to the current character being processed, i.e. 'x', because there is no walk down needed here (as activeLength is ZERO) and so next character we look for is current character being processed.

- While code implementation, we will loop through all the characters of string S one by one. Each loop for i^{th} character will do processing for phase i . Loop will run one or more time depending on how many extensions

are left to be performed (Please note that in a phase $i+1$, we don't really have to perform all $i+1$ extensions explicitly, as trick 3 will take care of j extensions for all j leaf edges coming from previous phase i). We will use a variable **remainingSuffixCount**, to track how many extensions are yet to be performed explicitly in any phase (after trick 3 is performed). Also, at the end of any phase, if remainingSuffixCount is ZERO, this tells that all suffixes supposed to be added in tree, are added explicitly and present in tree. If remainingSuffixCount is non-zero at the end of any phase, that tells that suffixes of that many count are not added in tree explicitly (because of rule 3, we stopped early), but they are in tree implicitly though (Such trees are called implicit suffix tree). These implicit suffixes will be added explicitly in subsequent phases when a unique character comes in the way.

We will continue our discussion in [Part 4](#) and [Part 5](#). Code implementation will be discussed in [Part 6](#).

References:

<http://web.stanford.edu/~mjkay/gusfield.pdf>

[Ukkonen's suffix tree algorithm in plain English](#)

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Pattern Searching
Suffix-Tree

Ukkonen's Suffix Tree Construction – Part 4

This article is continuation of following three articles:

[Ukkonen's Suffix Tree Construction – Part 1](#)

[Ukkonen's Suffix Tree Construction – Part 2](#)

[Ukkonen's Suffix Tree Construction – Part 3](#)

Please go through [Part 1](#), [Part 2](#) and [Part 3](#), before looking at current article, where we have seen few basics on suffix tree, high level ukkonen's algorithm, suffix link and three implementation tricks and some details on activePoint along with an example string "abcabxabcd" where we went through four phases of building suffix tree.

Let's revisit those four phases we have seen already in [Part 3](#), in terms of trick 2, trick 3 and activePoint.

- activePoint is initialized to (root, NULL, 0), i.e. activeNode is root, activeEdge is NULL (for easy understanding, we are giving character value to activeEdge, but in code implementation, it will be index of the character) and activeLength is ZERO.
- The global variable END and remainingSuffixCount are initialized to ZERO

*****Phase 1*****

In Phase 1, we read 1st character (a) from string S

- Set END to 1
- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 1 here, i.e. there is 1 extension left to be performed)
- Run a loop remainingSuffixCount times (i.e. one time) as below:
 - If activeLength is ZERO, set activeEdge to the current character (here activeEdge will be 'a'). This is **APCFALZ**.
 - Check if there is an edge going out from activeNode (which is root in this phase 1) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, leaf edge gets created (Rule 2).
 - Once extension is performed, decrement the remainingSuffixCount by 1
 - At this point, activePoint is (root, a, 0)

At the end of phase 1, remainingSuffixCount is ZERO (All suffixes are added explicitly).

Figure 20 in [Part 3](#) is the resulting tree after phase 1.

*****Phase 2*****

In Phase 2, we read 2nd character (b) from string S

- Set END to 2 (This will do extension 1)
- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 1 here, i.e. there is 1 extension left to be performed)
- Run a loop remainingSuffixCount times (i.e. one time) as below:
 - If activeLength is ZERO, set activeEdge to the current character (here activeEdge will be 'b'). This is **APCFALZ**.
 - Check if there is an edge going out from activeNode (which is root in this phase 2) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, leaf edge gets created.
 - Once extension is performed, decrement the remainingSuffixCount by 1
 - At this point, activePoint is (root, b, 0)

At the end of phase 2, remainingSuffixCount is ZERO (All suffixes are added explicitly).

Figure 22 in [Part 3](#) is the resulting tree after phase 2.

*****Phase 3*****

In Phase 3, we read 3rd character (c) from string S

- Set END to 3 (This will do extensions 1 and 2)
- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 1 here, i.e. there is 1 extension left to be performed)
- Run a loop remainingSuffixCount times (i.e. one time) as below:
 - If activeLength is ZERO, set activeEdge to the current character (here activeEdge will be 'c'). This is **APCFALZ**.
 - Check if there is an edge going out from activeNode (which is root in this phase 3) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, leaf edge gets created.
 - Once extension is performed, decrement the remainingSuffixCount by 1
 - At this point, activePoint is (root, c, 0)

At the end of phase 3, remainingSuffixCount is ZERO (All suffixes are added explicitly).

Figure 25 in [Part 3](#) is the resulting tree after phase 3.

*****Phase 4*****

In Phase 4, we read 4th character (a) from string S

- Set END to 4 (This will do extensions 1, 2 and 3)
- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 1 here, i.e. there is 1 extension left to be performed)
- Run a loop remainingSuffixCount times (i.e. one time) as below:
 - If activeLength is ZERO, set activeEdge to the current character (here activeEdge will be 'a'). This is **APCFALZ**.
 - Check if there is an edge going out from activeNode (which is root in this phase 3) for the activeEdge. If not, create a leaf edge. If present, walk down (The trick 1 – skip/count). In our example, edge 'a' is present going out of activeNode (i.e. root). No walk down needed as activeLength < edgeLength. We increment activeLength from zero to 1 (**APCFER3**) and stop any further processing (Rule 3).
 - At this point, activePoint is (root, a, 1) and remainingSuffixCount remains set to 1 (no change there)

At the end of phase 4, remainingSuffixCount is 1 (One suffix 'a', the last one, is not added explicitly in tree, but it is there in tree implicitly).

Figure 28 in [Part 3](#) is the resulting tree after phase 4.

Revisiting completed for 1st four phases, we will continue building the tree and see how it goes.

*****Phase 5*****

In phase 5, we read 5th character (b) from string S

- Set END to 5 (This will do extensions 1, 2 and 3). See Figure 29 shown below.
- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 2 here, i.e. there are 2 extension left to be performed, which are extensions 4 and 5. Extension 4 is supposed to add suffix "ab" and extension 5 is supposed to add suffix "b" in tree)
- Run a loop remainingSuffixCount times (i.e. two times) as below:
 - Check if there is an edge going out from activeNode (which is root in this phase 3) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, edge 'a' is present going out of activeNode (i.e.

root).

- Do a walk down (The trick 1 – skip/count) if necessary. In current phase 5, no walk down needed as activeLength < edgeLength. Here activePoint is (root, a, 1) for extension 4 (remainingSuffixCount = 2)
- Check if current character of string S (which is 'b') is already present after the activePoint. If yes, no more processing (rule 3). Same is the case in our example, so we increment activeLength from 1 to 2 (**APCFER3**) and we stop here (Rule 3).
- At this point, activePoint is (root, a, 2) and remainingSuffixCount remains set to 2 (no change in remainingSuffixCount)

Ukkonen's Suffix Tree Construction



At the end of phase 5, remainingSuffixCount is 2 (Two suffixes, 'ab' and 'b', the last two, are not added explicitly in tree, but they are in tree implicitly).

*****Phase 6*****

In phase 6, we read 6th character (x) from string S

- Set END to 6 (This will do extensions 1, 2 and 3)

Ukkonen's Suffix Tree Construction



- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 3 here, i.e. there are 3 extension left to be performed, which are extensions 4, 5 and 6 for suffixes "abx", "bx" and "x" respectively)
- Run a loop remainingSuffixCount times (i.e. three times) as below:
- While extension 4, the activePoint is (root, a, 2) which points to 'b' on edge starting with 'a'.
- In extension 4, current character 'x' from string S doesn't match with the next character on the edge after activePoint, so this is the case of extension rule 2. So a leaf edge is created here with edge label x. Also here traversal ends in middle of an edge, so a new internal node also gets created at the end of activePoint.
- Decrement the remainingSuffixCount by 1 (from 3 to 2) as suffix "abx" added in tree.

Ukkonen's Suffix Tree Construction



Now activePoint will change after applying rule 2. Three other cases, (**APCFER3**, **APCFWD** and **APCFALZ**) where activePoint changes, are already discussed in [Part 3](#).

activePoint change for extension rule 2 (APCFER2):

Case 1 (APCFER2C1): If activeNode is root and activeLength is greater than ZERO, then decrement the activeLength by 1 and activeEdge will be set " $S[i - \text{remainingSuffixCount} + 1]$ " where i is current phase number. Can you see why this change in activePoint? Look at current extension we just discussed above for phase 6 (i=6) again where we added suffix "abx". There activeLength is 2 and activeEdge is 'a'. Now in next extension, we need to add suffix "bx" in the tree, i.e. path label in next extension should start with 'b'. So 'b' (the 5th character in string S) should be active edge for next extension and index of b will be " $i - \text{remainingSuffixCount} + 1$ " ($6 - 2 + 1 = 5$). activeLength is decremented by 1 because activePoint gets closer to root by length 1 after every extension.

What will happen If activeNode is root and activeLength is ZERO? This case is already taken care by **APCFALZ**.

Case 2 (APCFER2C2): If activeNode is not root, then follow the suffix link from current activeNode. The new node (which can be root node or another internal node) pointed by suffix link will be the activeNode for next extension. No change in activeLength and activeEdge. Can you see why this change in activePoint? This is because: If two nodes are connected by a suffix link, then labels on all paths going down from those two nodes, starting with same character, will be exactly same and so for two corresponding similar point on those paths, activeEdge and activeLength will be same and the two nodes will be the activeNode. Look at Figure 18 in [Part 2](#). Let's say in phase i and extension j, suffix 'xAabcdedg' was added in tree. At that point, let's say activePoint was (Node-V, a, 7), i.e. point 'g'. So for next extension j+1, we would add suffix 'Aabcdedfg' and for that we need to traverse 2nd path shown in Figure 18. This can be done by following suffix link from current activeNode v. Suffix link takes us to the path to be traversed somewhere in between [Node s(v)] below which the path is exactly same as how it was below the previous activeNode v. As said earlier, "activePoint gets closer to root by length 1 after every extension", this reduction in length will happen above the node s(v) but below s(v), no change at all. So when activeNode is not root in current extension, then for next extension, only activeNode changes (No change in activeEdge and activeLength).

- At this point in extension 4, current activePoint is (root, a, 2) and based on **APCFER2C1**, new activePoint for next extension 5 will be (root, b, 1)
- Next suffix to be added is 'bx' (with remainingSuffixCount 2).
- Current character 'x' from string S doesn't match with the next character on the edge after activePoint, so this is the case of extension rule 2. So a leaf edge is created here with edge label x. Also here traversal ends in middle of an edge, so a new internal node also gets created at the end of activePoint.
Suffix link is also created from previous internal node (of extension 4) to the new internal node created in current extension 5.
- Decrement the remainingSuffixCount by 1 (from 2 to 1) as suffix "bx" added in tree.

Ukkonen's Suffix Tree Construction



- At this point in extension 5, current activePoint is (root, b, 1) and based on **APCFER2C1** new activePoint for next extension 6 will be (root, x, 0)
- Next suffix to be added is 'x' (with remainingSuffixCount 1).
- In the next extension 6, character x will not match to any existing edge from root, so a new edge with label x will be created from root node. Also suffix link from previous extension's internal node goes to root (as no new internal node created in current extension 6).
- Decrement the remainingSuffixCount by 1 (from 1 to 0) as suffix "x" added in tree



This completes the phase 6.

Note that phase 6 has completed all its 6 extensions (Why? Because the current character c was not seen in string so far, so rule 3, which stops further extensions never got chance to get applied in phase 6) and so the tree generated after phase 6 is a true suffix tree (i.e. not an implicit tree) for the characters 'abcabx' read so far and it has all suffixes explicitly in the tree.

While building the tree above, following facts were noticed:

- A newly created internal node in extension i, points to another internal node or root (if activeNode is root in extension i+1) by the end of extension i+1 via suffix link (Every internal node MUST have a suffix link pointing to another internal node or root)
- Suffix link provides short cut while searching path label end of next suffix
- With proper tracking of activePoints between extensions/phases, unnecessary walkdown from root can be avoided.

We will go through rest of the phases (7 to 11) in [Part 5](#) and build the tree completely and after that, we will see the code for the algorithm in [Part 6](#).

References:

<http://web.stanford.edu/~mjkay/gusfield.pdf>

[Ukkonen's suffix tree algorithm in plain English](#)

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Pattern Searching
Pattern Searching
Suffix-Tree

Ukkonen's Suffix Tree Construction – Part 5

This article is continuation of following four articles:

[Ukkonen's Suffix Tree Construction – Part 1](#)

[Ukkonen's Suffix Tree Construction – Part 2](#)

[Ukkonen's Suffix Tree Construction – Part 3](#)

[Ukkonen's Suffix Tree Construction – Part 4](#)

Please go through [Part 1](#), [Part 2](#), [Part 3](#) and [Part 4](#), before looking at current article, where we have seen few basics on suffix tree, high level ukkonen's algorithm, suffix link and three implementation tricks and some details on activePoint along with an example string "abcabxabc" where we went through six phases of building suffix tree.

Here, we will go through rest of the phases (7 to 11) and build the tree completely.

*****Phase 7*****

In phase 7, we read 7th character (a) from string S

- Set END to 7 (This will do extensions 1, 2, 3, 4, 5 and 6) – because we have 6 leaf edges so far by the end of previous phase 6.



- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 1 here, i.e. there is only 1 extension left to be performed, which is extensions 7 for suffix 'a')
- Run a loop remainingSuffixCount times (i.e. one time) as below:
- If activeLength is ZERO [activePoint in previous phase was (root, x, 0)], set activeEdge to the current character (here activeEdge will be 'a'). This is **APCFALZ**. Now activePoint becomes (root, 'a', 0).
- Check if there is an edge going out from activeNode (which is root in this phase 7) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, edge 'a' is present going out of activeNode (i.e. root), here we increment activeLength from zero to 1 (**APCFER3**) and stop any further processing.
- At this point, activePoint is (root, a, 1) and remainingSuffixCount remains set to 1 (no change there)

At the end of phase 7, remainingSuffixCount is 1 (One suffix 'a', the last one, is not added explicitly in tree, but it is there in tree implicitly).

Above Figure 33 is the resulting tree after phase 7.

*****Phase 8*****

In phase 8, we read 8th character (b) from string S

- Set END to 8 (This will do extensions 1, 2, 3, 4, 5 and 6) – because we have 6 leaf edges so far by the end of previous phase 7 (Figure 34).



- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 2 here, i.e. there are two extensions left to be performed, which are extensions 7 and 8 for suffixes 'ab' and 'b' respectively)
- Run a loop remainingSuffixCount times (i.e. two times) as below:
- Check if there is an edge going out from activeNode (which is root in this phase 8) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, edge 'a' is present going out of activeNode (i.e. root).
- Do a walk down (The trick 1 – skip/count) if necessary. In current phase 8, no walk down needed as activeLength < edgeLength. Here activePoint is (root, a, 1) for extension 7 (remainingSuffixCount = 2)
- Check if current character of string S (which is 'b') is already present after the activePoint. If yes, no more processing (rule 3). Same is the case in our example, so we increment activeLength from 1 to 2 (**APCFER3**) and we stop here (Rule 3).
- At this point, activePoint is (root, a, 2) and remainingSuffixCount remains set to 2 (no change in remainingSuffixCount)

At the end of phase 8, remainingSuffixCount is 2 (Two suffixes, 'ab' and 'b', the last two, are not added explicitly in tree explicitly, but they are in tree implicitly).

*****Phase 9*****

In phase 9, we read 9th character (c) from string S

- Set END to 9 (This will do extensions 1, 2, 3, 4, 5 and 6) – because we have 6 leaf edges so far by the end of previous phase 8.

Ukkonen's Suffix Tree Construction



- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 3 here, i.e. there are three extensions left to be performed, which are extensions 7, 8 and 9 for suffixes 'abc', 'bc' and 'c' respectively)
- Run a loop remainingSuffixCount times (i.e. three times) as below:
- Check if there is an edge going out from activeNode (which is root in this phase 9) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, edge 'a' is present going out of activeNode (i.e. root).
- Do a walk down (The trick 1 – skip/count) if necessary. In current phase 9, walk down needed as activeLength(2) >= edgeLength(2). While walk down, activePoint changes to (Node A, c, 0) based on **APCFWD** (This is first time **APCFWD** is being applied in our example).
- Check if current character of string S (which is 'c') is already present after the activePoint. If yes, no more processing (rule 3). Same is the case in our example, so we increment activeLength from 0 to 1 (**APCFER3**) and we stop here (Rule 3).
- At this point, activePoint is (Node A, c, 1) and remainingSuffixCount remains set to 3 (no change in remainingSuffixCount)

At the end of phase 9, remainingSuffixCount is 3 (Three suffixes, 'abc', 'bc' and 'c', the last three, are not added explicitly in tree explicitly, but they are in tree implicitly).

*****Phase 10*****

In phase 10, we read 10th character (d) from string S

- Set END to 10 (This will do extensions 1, 2, 3, 4, 5 and 6) – because we have 6 leaf edges so far by the end of previous phase 9.

Ukkonen's Suffix Tree Construction



- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 4 here, i.e. there are four extensions left to be performed, which are extensions 7, 8, 9 and 10 for suffixes 'abcd', 'bcd', 'cd' and 'd' respectively)
- Run a loop remainingSuffixCount times (i.e. four times) as below:

*****Extension 7*****

- Check if there is an edge going out from activeNode (Node A) for the activeEdge(c). If not, create a leaf edge. If present, walk down. In our example, edge 'c' is present going out of activeNode (Node A).
- Do a walk down (The trick 1 – skip/count) if necessary. In current Extension 7, no walk down needed as activeLength < edgeLength.
- Check if current character of string S (which is 'd') is already present after the activePoint. If not, rule 2 will apply. In our example, there is no path starting with 'd' going out of activePoint, so we create a leaf edge with label 'd'. Since activePoint ends in the middle of an edge, we will create a new internal node just after the activePoint (Rule 2)

Ukkonen's Suffix Tree Construction



The newly created internal node c (in above Figure) in current extension 7, will get its suffix link set in next extension 8 (see Figure 38 below).

- Decrement the remainingSuffixCount by 1 (from 4 to 3) as suffix "abcd" added in tree.
- Now activePoint will change for next extension 8. Current activeNode is an internal node (Node A), so there must be a suffix link from there and we will follow that to get new activeNode and that's going to be 'Node B'. There is no change in activeEdge and activeLength (This is **APCFER2C2**). So new activePoint is (Node B, c, 1).

*****Extension 8*****

- Now in extension 8 (here we will add suffix 'bcd'), while adding character 'd' after the current activePoint, exactly same logic will apply as previous extension 7. In previous extension 7, we added character 'd' at activePoint (Node A, c, 1) and in current extension 8, we are going to add same character 'd' at activePoint (Node B, c, 1). So logic will be same and here we a new leaf edge with label 'd' and a new internal node will be created. And the new internal node (C) of previous extension will point to the new node (D) of current extension via suffix link.

Ukkonen's Suffix Tree Construction



Please note the node C from previous extension (see Figure 37 above) got its suffix link set here and node D created in current extension will get its suffix link set in next extension. What happens if no new node created in next extensions? We have seen this before in Phase 6 (Part 4) and will see again in last extension of this Phase 10. Stay Tuned.

- Decrement the remainingSuffixCount by 1 (from 3 to 2) as suffix "bcd" added in tree.
- Now activePoint will change for next extension 9. Current activeNode is an internal node (Node B), so there must be a suffix link from there and we will follow that to get new activeNode and that is 'Root Node'. There is no change in activeEdge and activeLength (This is **APCFER2C2**). So new activePoint is (root, c, 1).

*****Extension 9*****

- Now in extension 9 (here we will add suffix 'cd'), while adding character 'd' after the current activePoint, exactly same logic will apply as previous extensions 7 and 8. Note that internal node D created in previous extension 8, now points to internal node E (created in current extension) via suffix link.

Ukkonen's Suffix Tree Construction



- Decrement the remainingSuffixCount by 1 (from 2 to 1) as suffix "cd" added in tree.
- Now activePoint will change for next extension 10. Current activeNode is root and activeLength is 1, based on **APCFER2C1**, activeNode will remain 'root', activeLength will be decremented by 1 (from 1 to ZERO) and activeEdge will be 'd'. So new activePoint is (root, d, 0).

*****Extension 10*****

- Now in extension 10 (here we will add suffix 'd'), while adding character 'd' after the current activePoint, there is no edge starting with d going out of activeNode root, so a new leaf edge with label d is created (Rule 2). Note that internal node E created in previous extension 9, now points to root node via suffix link (as no new internal node created in this extension).

Ukkonen's Suffix Tree Construction



Internal Node created in previous extension, waiting for suffix link to be set in next extension, points to root if no internal node created in next extension. In code implementation, as soon as a new internal node (Say A) gets created in an extension j, we will set its suffix link to root node and in next extension j+1, if Rule 2 applies on an existing or newly created node (Say B) or Rule 3 applies with some active node (Say B), then suffix link of node A will change to the new node B, else node A will keep pointing to root

- Decrement the remainingSuffixCount by 1 (from 1 to 0) as suffix "d" added in tree. That means no more suffix is there to add and so the phase 10 ends here. Note that this tree is an explicit tree as all suffixes are added in tree explicitly (Why ?? because character d was not seen before in string S so far)
- activePoint for next phase 11 is (root, d, 0).

We see following facts in Phase 10:

- Internal Nodes connected via suffix links have exactly same tree below them, e.g. In above Figure 40, A and B have same tree below them, similarly C, D and E have same tree below them.
- Due to above fact, in any extension, when current activeNode is derived via suffix link from previous extension's activeNode, then exactly same extension logic apply in current extension as previous extension. (In Phase 10, same extension logic is applied in extensions 7, 8 and 9)
- If a new internal node gets created in extension j of any phase i, then this newly created internal node will get its suffix link set by the end of next extension j+1 of same phase i. e.g. node C got created in extension 7 of phase 10 (Figure 37) and it got its suffix link set to node D in extension 8 of same phase 10 (Figure 38). Similarly node D got created in extension 8 of phase 10 (Figure 38) and it got its suffix link set to node E in extension 9 of same phase 10 (Figure 39). Similarly node E got created in extension 9 of phase 10 (Figure 39) and it got its suffix link set to root in extension 10 of same phase 10 (Figure 40).
- Based on above fact, every internal node will have a suffix link to some other internal node or root. Root is not an internal node and it will not have suffix link.

*****Phase 11*****

In phase 11, we read 11th character (\$) from string S

- Set END to 11 (This will do extensions 1 to 10) – because we have 10 leaf edges so far by the end of previous phase 10.

Ukkonen's Suffix Tree Construction



- Increment remainingSuffixCount by 1 (from 0 to 1), i.e. there is only one suffix "\$" to be added in tree.
- Since activeLength is ZERO, activeEdge will change to current character '\$' of string S being processed (**APCFALZ**).
- There is no edge going out from activeNode root, so a leaf edge with label '\$' will be created (Rule 2).

Ukkonen's Suffix Tree Construction



- Decrement the remainingSuffixCount by 1 (from 1 to 0) as suffix "\$" added in tree. That means no more suffix is there to add and so the phase 11 ends here. Note that this tree is an explicit tree as all suffixes are added in tree explicitly (Why ?? because character \$ was not seen before in string S so far)

Now we have added all suffixes of string 'abcabxabcd\$' in suffix tree. There are 11 leaf ends in this tree and labels on the path from root to leaf end represents one suffix. Now the only one thing left is to assign a number (suffix index) to each leaf end and that number would be the suffix starting position in the string S. This can be done by a DFS traversal on tree. While DFS traversal, keep track of label length and when a leaf end is found, set the suffix index as "stringSize - labelSize + 1". Indexed suffix tree will look like below:

Ukkonen's Suffix Tree Construction



In above Figure, suffix indices are shown as character position starting with 1 (It's not zero indexed). In code implementation, suffix index will be set as zero indexed, i.e. where we see suffix index j (1 to m for string of length m) in above figure, in code implementation, it will be j-1 (0 to m-1)

And we are done !!!!

Data Structure to represent suffix tree

How to represent the suffix tree?? There are nodes, edges, labels and suffix links and indices.

Below are some of the operations/query we will be doing while building suffix tree and later on while using the suffix tree in different applications/usages:

- What length of path label on some edge?
- What is the path label on some edge?
- Check if there is an outgoing edge for a given character from a node.
- What is the character value on an edge at some given distance from a node?
- Where an internal node is pointing via suffix link?
- What is the suffix index on a path from root to leaf?
- Check if a given string present in suffix tree (as substring, suffix or prefix)?

We may think of different data structures which can fulfil these requirements.

In the next [Part 6](#), we will discuss the data structure we will use in our code implementation and the code as well.

References:

<http://web.stanford.edu/~mjkay/gusfield.pdf>

[Ukkonen's suffix tree algorithm in plain English](#)

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Pattern Searching
Pattern Searching
Suffix-Tree

Ukkonen's Suffix Tree Construction – Part 6

This article is continuation of following five articles:

[Ukkonen's Suffix Tree Construction – Part 1](#)

[Ukkonen's Suffix Tree Construction – Part 2](#)

[Ukkonen's Suffix Tree Construction – Part 3](#)

[Ukkonen's Suffix Tree Construction – Part 4](#)

[Ukkonen's Suffix Tree Construction – Part 5](#)

Please go through [Part 1](#), [Part 2](#), [Part 3](#), [Part 4](#) and [Part 5](#), before looking at current article, where we have seen few basics on suffix tree, high level ukkonen's algorithm, suffix link and three implementation tricks and activePoints along with an example string "abcabxabcd" where we went through all phases of building suffix tree.

Here, we will see the data structure used to represent suffix tree and the code implementation.

At that end of [Part 5](#) article, we have discussed some of the operations we will be doing while building suffix tree and later when we use suffix tree in different applications.

There could be different possible data structures we may think of to fulfill the requirements where some data structure may be slow on some operations and some fast. Here we will use following in our implementation:

We will have SuffixTreeNode structure to represent each node in tree. SuffixTreeNode structure will have following members:

- **children** – This will be an array of alphabet size. This will store all the children nodes of current node on different edges starting with different characters.
- **suffixLink** – This will point to other node where current node should point via suffix link.
- **start, end** – These two will store the edge label details from parent node to current node. (start, end) interval specifies the edge, by which the node is connected to its parent node. Each edge will connect two nodes, one parent and one child, and (start, end) interval of a given edge will be stored in the child node. Lets say there are two nodes A (parent) and B (Child) connected by an edge with indices (5, 8) then this indices (5, 8) will be stored in node B.
- **suffixIndex** – This will be non-negative for leaves and will give index of suffix for the path from root to this leaf. For non-leaf node, it will be -1.

This data structure will answer to the required queries quickly as below:

- How to check if a node is root ? — Root is a special node, with no parent and so it's start and end will be -1, for all other nodes, start and end indices will be non-negative.
- How to check if a node is internal or leaf node ? — suffixIndex will help here. It will be -1 for internal node and non-negative for leaf nodes.
- What is the length of path label on some edge ? — Each edge will have start and end indices and length of path label will be end-start+1
- What is the path label on some edge ? — If string is S, then path label will be substring of S from start index to end index inclusive, [start, end].
- How to check if there is an outgoing edge for a given character c from a node A ? — If A->children[c] is not NULL, there is a path, if NULL, no path.
- What is the character value on an edge at some given distance d from a node A ? — Character at distance d from node A will be S[A->start + d], where S is the string.
- Where an internal node is pointing via suffix link ? — Node A will point to A->suffixLink
- What is the suffix index on a path from root to leaf ? — If leaf node is A on the path, then suffix index on that path will be A->suffixIndex

Following is C implementation of Ukkonen's Suffix Tree Construction. The code may look a bit lengthy, probably because of a good amount of comments.

```
// A C program to implement Ukkonen's Suffix Tree Construction
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];

    //pointer to other node via suffix link
    struct SuffixTreeNode *suffixLink;

    /*(start, end) interval specifies the edge, by which the
    node is connected to its parent node. Each edge will
    connect two nodes, one parent and one child, and
    (start, end) interval of a given edge will be stored
    in the child node. Lets say there are two nodes A and B
    connected by an edge with indices (5, 8) then this
    indices (5, 8) will be stored in node B. */
    int start;
    int *end;

    /*for leaf nodes, it stores the index of suffix for
    the path from root to leaf*/
    int suffixIndex;
};

typedef struct SuffixTreeNode Node;

char text[100]; //Input string
Node *root = NULL; //Pointer to root node

/*lastNewNode will point to newly created internal node,
waiting for it's suffix link to be set, which might get
a new suffix link (other than root) in next extension of
same phase. lastNewNode will be set to NULL when last
newly created internal node (if there is any) got it's
suffix link reset to new internal node created in next
extension of same phase. */
```

```

Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represented as input string character
index (not the character itself)*/
int activeEdge = -1;
int activeLength = 0;

// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string

Node *newNode(int start, int *end)
{
    Node *node =(Node*) malloc(sizeof(Node));
    int i;
    for (i = 0; i < MAX_CHAR; i++)
        node->children[i] = NULL;

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

    /*suffixIndex will be set to -1 by default and
    actual suffix index will be set later for leaves
    at the end of all phases*/
    node->suffixIndex = -1;
    return node;
}

int edgeLength(Node *n) {
    return *(n->end) - (n->start) + 1;
}

int walkDown(Node *currNode)
{
    /*activePoint change for walk down (APCFWD) using
    Skip/Count Trick (Trick 1). If activeLength is greater
    than current edge length, set next internal node as
    activeNode and adjust activeEdge and activeLength
    accordingly to represent same activePoint*/
    if (activeLength >= edgeLength(currNode))
    {
        activeEdge += edgeLength(currNode);
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
        return 1;
    }
    return 0;
}

void extendSuffixTree(int pos)
{
    /*Extension Rule 1, this takes care of extending all
    leaves created so far in tree*/
    leafEnd = pos;

    /*Increment remainingSuffixCount indicating that a
    new suffix added to the list of suffixes yet to be
    added in tree*/
    remainingSuffixCount++;

    /*set lastNewNode to NULL while starting a new phase,
    indicating there is no internal node waiting for
    it's suffix link reset in current phase*/
    lastNewNode = NULL;

    //Add all suffixes (yet to be added) one by one in tree
    while(remainingSuffixCount > 0) {

        if (activeLength == 0)
            activeEdge = pos; //APCFALZ

        // There is no outgoing edge starting with
        // activeEdge from activeNode
        if (activeNode->children[text[activeEdge]] == NULL)
        {
            /*Extension Rule 2 (A new leaf edge gets created)
            activeNode->children[text[activeEdge]] =
                newNode(pos, &leafEnd);

            /*A new leaf edge is created in above line starting
            from an existing node (the current activeNode), and
            if there is any internal node waiting for it's suffix
            link get reset, point the suffix link from that last
            internal node to current activeNode. Then set lastNewNode
            to NULL indicating no more node waiting for suffix link
            reset*/
            if (lastNewNode != NULL)
            {
                lastNewNode->suffixLink = activeNode;
                lastNewNode = NULL;
            }
        }
        // There is an outgoing edge starting with activeEdge
        // from activeNode
        else
        {
            // Get the next node at the end of edge starting
            // with activeEdge
            Node *next = activeNode->children[text[activeEdge]];
            if (walkDown(next))//Do walkdown
            {
                //Start from next node (the new activeNode)
                continue;
            }
            /*Extension Rule 3 (current character being processed
            is already on the edge)*/

```

```

    if (text[next->start + activeLength] == text[pos])
    {
        //If a newly created node waiting for it's
        //suffix link to be set, then set suffix link
        //of that waiting node to current active node
        if (lastNewNode != NULL && activeNode != root)
        {
            lastNewNode->suffixLink = activeNode;
            lastNewNode = NULL;
        }

        //APCFER3
        activeLength++;
        /*STOP all further processing in this phase
        and move on to next phase*/
        break;
    }

    /*We will be here when activePoint is in middle of
    the edge being traversed and current character
    being processed is not on the edge (we fall off
    the tree). In this case, we add a new internal node
    and a new leaf edge going out of that new node. This
    is Extension Rule 2, where a new leaf edge and a new
    internal node get created*/
    splitEnd = (int*) malloc(sizeof(int));
    *splitEnd = next->start + activeLength - 1;

    //New internal node
    Node *split = newNode(next->start, splitEnd);
    activeNode->children[text[activeEdge]] = split;

    //New leaf coming out of new internal node
    split->children[text[pos]] = newNode(pos, &leafEnd);
    next->start += activeLength;
    split->children[text[next->start]] = next;

    /*We got a new internal node here. If there is any
    internal node created in last extensions of same
    phase which is still waiting for it's suffix link
    reset, do it now.*/
    if (lastNewNode != NULL)
    {
        /*suffixLink of lastNewNode points to current newly
        created internal node*/
        lastNewNode->suffixLink = split;
    }

    /*Make the current newly created internal node waiting
    for it's suffix link reset (which is pointing to root
    at present). If we come across any other internal node
    (existing or newly created) in next extension of same
    phase, when a new leaf edge gets added (i.e. when
    Extension Rule 2 applies is any of the next extension
    of same phase) at that point, suffixLink of this node
    will point to that internal node.*/
    lastNewNode = split;
}

/* One suffix got added in tree, decrement the count of
suffixes yet to be added.*/
remainingSuffixCount--;
if (activeNode == root && activeLength > 0) //APCFER2C1
{
    activeLength--;
    activeEdge = pos - remainingSuffixCount + 1;
}
else if (activeNode != root) //APCFER2C2
{
    activeNode = activeNode->suffixLink;
}
}
}

void print(int i, int j)
{
    int k;
    for (k=i; k<=j; k++)
        printf("%c", text[k]);
}

//Print the suffix tree as well along with setting suffix index
//So tree will be printed in DFS manner
//Each edge along with it's suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL) return;

    if (n->start != -1) //A non-root node
    {
        //Print the label on edge from parent to current node
        print(n->start, "(n->end)");
    }
    int leaf = 1;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            if (leaf == 1 && n->start != -1)
                printf("[%d]n", n->suffixIndex);

            //Current node is not a leaf as it has outgoing
            //edges from it.
            leaf = 0;
            setSuffixIndexByDFS(n->children[i], labelHeight +
                edgeLength(n->children[i]));
        }
    }
    if (leaf == 1)
    {
        n->suffixIndex = size - labelHeight;
        printf("[%d]n", n->suffixIndex);
    }
}

```

```

void freeSuffixTreeByPostOrder(Node *n)
{
    if (n == NULL)
        return;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
    if (n->suffixIndex == -1)
        free(n->end);
    free(n);
}

/*Build the suffix tree and print the edge labels along with
suffixIndex. suffixIndex for leaf edges will be >= 0 and
for non-leaf edges will be -1*/
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;

    /*Root is a special node with start and end indices as -1,
    as it has no parent from where an edge comes to root*/
    root = newNode(-1, rootEnd);

    activeNode = root; /*First activeNode will be root
    for (i=0; i<size; i++)
        extendSuffixTree(i);
    int labelHeight = 0;
    setSuffixIndexByDFS(root, labelHeight);

    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
}

// driver program to test above functions
int main(int argc, char *argv[])
{
    // strcpy(text, "abc"); buildSuffixTree();
    // strcpy(text, "abxac#"); buildSuffixTree();
    // strcpy(text, "abxa"); buildSuffixTree();
    // strcpy(text, "abxa$"); buildSuffixTree();
    // strcpy(text, "abcabxcd$"); buildSuffixTree();
    // strcpy(text, "geeksforgeeks$"); buildSuffixTree();
    // strcpy(text, "THIS IS A TEST TEXT$"); buildSuffixTree();
    // strcpy(text, "AABAACAADAABAAABAA$"); buildSuffixTree();
    return 0;
}

```

Output (Each edge of Tree, along with suffix index of child node on edge, is printed in DFS order. To understand the output better, match it with the last figure no 43 in previous [Part 5](#) article):

```

$ {10}
ab [-1]
c [-1]
abxabcd$ {0}
d$ {6}
xabcd$ {3}
b [-1]
c [-1]
abxabcd$ {1}
d$ {7}
xabcd$ {4}
c [-1]
abxabcd$ {2}
d$ {8}
d$ {9}
xabcd$ {5}

```

Now we are able to build suffix tree in linear time, we can solve many string problem in efficient way:

- Check if a given pattern P is substring of text T (Useful when text is fixed and pattern changes, [KMP](#) otherwise)
- Find all occurrences of a given pattern P present in text T
- Find longest repeated substring
- [Linear Time Suffix Array Creation](#)

The above basic problems can be solved by DFS traversal on suffix tree.

We will soon post articles on above problems and others like below:

- Build [Generalized suffix tree](#)
- Linear Time [Longest common substring problem](#)
- Linear Time [Longest palindromic substring](#)

And [More](#).

Test you understanding?

1. Draw suffix tree (with proper suffix link, suffix indices) for string "AABAACAADAABAAABAA\$" on paper and see if that matches with code output.
2. Every extension must follow one of the three rules: Rule 1, Rule 2 and Rule 3.
Following are the rules applied on five consecutive extensions in some Phase i (i > 5), which ones are valid:
A) Rule 1, Rule 2, Rule 2, Rule 3, Rule 3
B) Rule 1, Rule 2, Rule 2, Rule 3, Rule 2
C) Rule 2, Rule 1, Rule 1, Rule 3, Rule 3
D) Rule 1, Rule 1, Rule 1, Rule 1, Rule 1
E) Rule 2, Rule 2, Rule 2, Rule 2, Rule 2
F) Rule 3, Rule 3, Rule 3, Rule 3, Rule 3
3. What are the valid sequences in above for Phase 5
4. Every internal node MUST have it's suffix link set to another node (internal or root). Can a newly created node point to already existing internal node or not ? Can it happen that a new node created in extension j, may not get it's right suffix link in next extension j+1 and get the right one in later extensions like j+2, j+3 etc ?
5. Try solving the basic problems discussed above.

We have published following articles on suffix tree applications:

- [Suffix Tree Application 1 – Substring Check](#)
- [Suffix Tree Application 2 – Searching All Patterns](#)

- [Suffix Tree Application 3 – Longest Repeated Substring](#)
- [Suffix Tree Application 4 – Build Linear Time Suffix Array](#)
- [Generalized Suffix Tree 1](#)
- [Suffix Tree Application 5 – Longest Common Substring](#)
- [Suffix Tree Application 6 – Longest Palindromic Substring](#)

References:

<http://web.stanford.edu/~mjkay/gusfield.pdf>

[Ukkonen's suffix tree algorithm in plain English](#)

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Pattern Searching
Pattern Searching
Suffix Tree

Generalized Suffix Tree 1

In earlier suffix tree articles, we created suffix tree for one string and then we queried that tree for [substring check](#), [searching all patterns](#), [longest repeated substring](#) and [built suffix array](#) (All linear time operations).

There are lots of other problems where multiple strings are involved.

e.g. pattern searching in a text file or dictionary, spell checker, phone book, [Autocomplete](#), [Longest common substring problem](#), [Longest palindromic substring](#) and [More](#).

For such operations, all the involved strings need to be indexed for faster search and retrieval. One way to do this is using suffix trie or suffix tree. We will discuss suffix tree here.

A suffix tree made of a set of strings is known as [Generalized Suffix Tree](#).

We will discuss a simple way to build [Generalized Suffix Tree](#) here for **two strings only**.

Later, we will discuss another approach to build [Generalized Suffix Tree](#) for **two or more strings**.

Here we will use the [suffix tree implementation](#) for one string discussed already and modify that a bit to build [generalized suffix tree](#).

Lets consider two strings X and Y for which we want to build generalized suffix tree. For this we will make a new string X#Y\$ where # and \$ both are terminal symbols (must be unique). Then we will build suffix tree for X#Y\$ which will be the generalized suffix tree for X and Y. Same logic will apply for more than two strings (i.e. concatenate all strings using unique terminal symbols and then build suffix tree for concatenated string).

Lets say X = xabxa, and Y = babxba, then

X#Y\$ = xabxa#babxba\$

If we run the code implemented at [Ukkonen's Suffix Tree Construction – Part 6](#) for string xabxa#babxba\$, we get following output:

(Click to see it clearly)
[Generalized Suffix Tree](#)



We can use this tree to solve some of the problems, but we can refine it a bit by removing unwanted substrings on a path label. A path label should have substring from only one input string, so if there are path labels having substrings from multiple input strings, we can keep only the initial portion corresponding to one string and remove all the later portion. For example, for path labels #babxba\$, a#babxba\$ and bxa#babxba\$, we can remove babxba\$ (belongs to 2nd input string) and then new path labels will be #, a# and bxa# respectively. With this change, above diagram will look like below:

(Click to see it clearly)
[Generalized Suffix Tree](#)



Below implementation is built on top of [original implementation](#). Here we are removing unwanted characters on path labels. If a path label has “#” character in it, then we are trimming all characters after the “#” in that path label.

Note: This implementation builds generalized suffix tree for only two strings X and Y which are concatenated as X#Y\$

```
// A C program to implement Ukkonen's Suffix Tree Construction
// And then build generalized suffix tree
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];

    //pointer to other node via suffix link
    struct SuffixTreeNode *suffixLink;

    /*(start, end) interval specifies the edge, by which the
    node is connected to its parent node. Each edge will
    connect two nodes, one parent and one child, and
    (start, end) interval of a given edge will be stored
    in the child node. Lets say there are two nodes A and B
    connected by an edge with indices (5, 8) then this
    indices (5, 8) will be stored in node B. */
    int start;
    int *end;

    /*for leaf nodes, it stores the index of suffix for
    the path from root to leaf*/
    int suffixIndex;
};

typedef struct SuffixTreeNode Node;

char text[100]; //Input string
Node *root = NULL; //Pointer to root node

/*lastNewNode will point to newly created internal node,
waiting for it's suffix link to be set, which might get
a new suffix link (other than root) in next extension of
same phase. lastNewNode will be set to NULL when last
newly created internal node (if there is any) got it's
suffix link reset to new internal node created in next
extension of same phase. */
Node *lastNewNode = NULL;
Node *activeNode = NULL;
```



```

/*activeEdge is represented as input string character
index (not the character itself)*/
int activeEdge = -1;
int activeLength = 0;

// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string

Node *newNode(int start, int *end)
{
    Node *node =(Node*) malloc(sizeof(Node));
    int i;
    for (i = 0; i < MAX_CHAR; i++)
        node->children[i] = NULL;

    /*For root node, suffixLink will be set to NULL.
    For internal nodes, suffixLink will be set to root
    by default in current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

    /*suffixIndex will be set to -1 by default and
    actual suffix index will be set later for leaves
    at the end of all phases*/
    node->suffixIndex = -1;
    return node;
}

int edgeLength(Node *n) {
    if(n == root)
        return 0;
    return *(n->end) - (n->start) + 1;
}

int walkDown(Node *currNode)
{
    /*activePoint change for walk down (APCFWD) using
    Skip/Count Trick (Trick 1). If activeLength is greater
    than current edge length, set next internal node as
    activeNode and adjust activeEdge and activeLength
    accordingly to represent same activePoint*/
    if (activeLength >= edgeLength(currNode))
    {
        activeEdge += edgeLength(currNode);
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
        return 1;
    }
    return 0;
}

void extendSuffixTree(int pos)
{
    /*Extension Rule 1, this takes care of extending all
    leaves created so far in tree*/
    leafEnd = pos;

    /*Increment remainingSuffixCount indicating that a
    new suffix added to the list of suffixes yet to be
    added in tree*/
    remainingSuffixCount++;

    /*set lastNewNode to NULL while starting a new phase,
    indicating there is no internal node waiting for
    it's suffix link reset in current phase*/
    lastNewNode = NULL;

    //Add all suffixes (yet to be added) one by one in tree
    while(remainingSuffixCount > 0) {

        if (activeLength == 0)
            activeEdge = pos; //APCFALZ

        // There is no outgoing edge starting with
        // activeEdge from activeNode
        if (activeNode->children[text[activeEdge]] == NULL)
        {
            /*Extension Rule 2 (A new leaf edge gets created)
            activeNode->children[text[activeEdge]] =
                newNode(pos, &leafEnd);

            /*A new leaf edge is created in above line starting
            from an existing node (the current activeNode), and
            if there is any internal node waiting for it's suffix
            link get reset, point the suffix link from that last
            internal node to current activeNode. Then set lastNewNode
            to NULL indicating no more node waiting for suffix link
            reset.*/
            if (lastNewNode != NULL)
            {
                lastNewNode->suffixLink = activeNode;
                lastNewNode = NULL;
            }
        }

        // There is an outgoing edge starting with activeEdge
        // from activeNode
        else
        {
            // Get the next node at the end of edge starting
            // with activeEdge
            Node *next = activeNode->children[text[activeEdge]];
            if (walkDown(next))//Do walkdown
            {
                //Start from next node (the new activeNode)
                continue;
            }

            /*Extension Rule 3 (current character being processed
            is already on the edge)*/
            if (text[next->start + activeLength] == text[pos])

```

```

{
    //If a newly created node waiting for it's
    //suffix link to be set, then set suffix link
    //of that waiting node to current active node
    if(lastNewNode != NULL && activeNode != root)
    {
        lastNewNode->suffixLink = activeNode;
        lastNewNode = NULL;
    }

    //APCFER3
    activeLength++;
    /*STOP all further processing in this phase
    and move on to next phase*/
    break;
}

/*We will be here when activePoint is in middle of
the edge being traversed and current character
being processed is not on the edge (we fall off
the tree). In this case, we add a new internal node
and a new leaf edge going out of that new node. This
is Extension Rule 2, where a new leaf edge and a new
internal node get created*/
splitEnd = (int*) malloc(sizeof(int));
*splitEnd = next->start + activeLength - 1;

//New internal node
Node *split = newNode(next->start, splitEnd);
activeNode->children[text[activeEdge]] = split;

//New leaf coming out of new internal node
split->children[text[pos]] = newNode(pos, &leafEnd);
next->start += activeLength;
split->children[text[next->start]] = next;

/*We got a new internal node here. If there is any
internal node created in last extensions of same
phase which is still waiting for it's suffix link
reset, do it now.*/
if (lastNewNode != NULL)
{
    /*suffixLink of lastNewNode points to current newly
    created internal node*/
    lastNewNode->suffixLink = split;
}

/*Make the current newly created internal node waiting
for it's suffix link reset (which is pointing to root
at present). If we come across any other internal node
(existing or newly created) in next extension of same
phase, when a new leaf edge gets added (i.e. when
Extension Rule 2 applies is any of the next extension
of same phase) at that point, suffixLink of this node
will point to that internal node.*/
lastNewNode = split;
}

/* One suffix got added in tree, decrement the count of
suffixes yet to be added.*/
remainingSuffixCount--;
if (activeNode == root && activeLength > 0) //APCFER2C1
{
    activeLength--;
    activeEdge = pos - remainingSuffixCount + 1;
}
else if (activeNode != root) //APCFER2C2
{
    activeNode = activeNode->suffixLink;
}
}
}

void print(int i, int j)
{
    int k;
    for (k=i; k<=j && text[k] != '#'; k++)
        printf("%c", text[k]);
    if(k<=j)
        printf("#");
}

//Print the suffix tree as well along with setting suffix index
//So tree will be printed in DFS manner
//Each edge along with it's suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL) return;

    if (n->start != -1) //A non-root node
    {
        //Print the label on edge from parent to current node
        print(n->start, "(n->end)");
    }
    int leaf = 1;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            if ((leaf == 1 && n->start != -1)
                printf("%d]n", n->suffixIndex);

            //Current node is not a leaf as it has outgoing
            //edges from it.
            leaf = 0;
            setSuffixIndexByDFS(n->children[i], labelHeight +
                                edgeLength(n->children[i]));
        }
    }
    if (leaf == 1)
    {
        for(i = n->start; i <= "(n->end)"; i++)
        {
            if(text[i] == '#') //Trim unwanted characters
            {

```

```

n->end = (int*) malloc(sizeof(int));
*(n->end) = i;
}
}
n->suffixIndex = size - labelHeight;
printf( "[%d]\n", n->suffixIndex);
}
}

void freeSuffixTreeByPostOrder(Node *n)
{
    if (n == NULL)
        return;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
    if (n->suffixIndex == -1)
        free(n->end);
    free(n);
}

/*Build the suffix tree and print the edge labels along with
suffixIndex. suffixIndex for leaf edges will be >= 0 and
for non-leaf edges will be -1*/
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;

    /*Root is a special node with start and end indices as -1.
    as it has no parent from where an edge comes to root*/
    root = newNode(-1, rootEnd);

    activeNode = root; //First activeNode will be root
    for (i=0; i<size; i++)
        extendSuffixTree(i);
    int labelHeight = 0;
    setSuffixIndexByDFS(root, labelHeight);

    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
}

// driver program to test above functions
int main(int argc, char *argv[])
{
    // strcpy(text, "xabxac#abcabxabcd$"); buildSuffixTree();
    strcpy(text, "xabxa#babxba$"); buildSuffixTree();
    return 0;
}

```

Output: (You can see that below output corresponds to the 2nd Figure shown above)

```

# [5]
$ [12]
a [-1]
# [4]
$ [11]
bx [-1]
a# [1]
ba$ [7]
b [-1]
a [-1]
$ [10]
bxba$ [6]
x [-1]
a# [2]
ba$ [8]
x [-1]
a [-1]
# [3]
bxa# [0]
ba$ [9]

```

If two strings are of size M and N, this implementation will take $O(M+N)$ time and space.

If input strings are not concatenated already, then it will take $2(M+N)$ space in total, $M+N$ space to store the generalized suffix tree and another $M+N$ space to store concatenated string.

Followup:

Extend above implementation for more than two strings (i.e. concatenate all strings using unique terminal symbols and then build suffix tree for concatenated string)

One problem with this approach is the need of unique terminal symbol for each input string. This will work for few strings but if there is too many input strings, we may not be able to find that many unique terminal symbols.

We will discuss another approach to build generalized suffix tree soon where we will need only one unique terminal symbol and that will resolve the above problem and can be used to build generalized suffix tree for any number of input strings.

We have published following more articles on suffix tree applications:

- [Suffix Tree Application 1 – Substring Check](#)
- [Suffix Tree Application 2 – Searching All Patterns](#)
- [Suffix Tree Application 3 – Longest Repeated Substring](#)
- [Suffix Tree Application 4 – Build Linear Time Suffix Array](#)
- [Suffix Tree Application 5 – Longest Common Substring](#)
- [Suffix Tree Application 6 – Longest Palindromic Substring](#)

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Pattern Searching
Pattern Searching
Suffix-Tree

Suffix Tree Application 4 – Build Linear Time Suffix Array

Given a string, build it's [Suffix Array](#)

We have already discussed following two ways of building suffix array:

- Naive $O(n^2 \log n)$ algorithm
- Enhanced $O(n \log n)$ algorithm

Please go through these to have the basic understanding.

Here we will see how to build suffix array in linear time using suffix tree.

As a prerequisite, we must know how to build a suffix tree in one or the other way.

Here we will build suffix tree using Ukkonen's Algorithm, discussed already as below:

[Ukkonen's Suffix Tree Construction – Part 1](#)

[Ukkonen's Suffix Tree Construction – Part 2](#)

[Ukkonen's Suffix Tree Construction – Part 3](#)

[Ukkonen's Suffix Tree Construction – Part 4](#)

[Ukkonen's Suffix Tree Construction – Part 5](#)

[Ukkonen's Suffix Tree Construction – Part 6](#)

Lets consider string abcabxabcd.

It's suffix array would be:

0 6 3 1 7 4 2 8 9 5

Lets look at following figure:

Suffix Tree Application



This is suffix tree for String "abcabxabcd\$"

If we do a DFS traversal, visiting edges in lexicographic order (we have been doing the same traversal in other Suffix Tree Application articles as well) and print suffix indices on leaves, we will get following:

10 0 6 3 1 7 4 2 8 9 5

"\$" is lexicographically lesser than [a-zA-Z].

The suffix index 10 corresponds to edge with "\$" label.

Except this 1st suffix index, the sequence of all other numbers gives the suffix array of the string.

So if we have a suffix tree of the string, then to get it's suffix array, we just need to do a lexicographic order DFS traversal and store all the suffix indices in resultant suffix array, except the very 1st suffix index.

```
// A C program to implement Ukkonen's Suffix Tree Construction
// And then create suffix array in linear time
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];

    //pointer to other node via suffix link
    struct SuffixTreeNode *suffixLink;

    /*(start, end) interval specifies the edge, by which the
    node is connected to its parent node. Each edge will
    connect two nodes, one parent and one child, and
    (start, end) interval of a given edge will be stored
    in the child node. Lets say there are two nodes A and B
    connected by an edge with indices (5, 8) then this
    indices (5, 8) will be stored in node B. */
    int start;
    int *end;

    /*for leaf nodes, it stores the index of suffix for
    the path from root to leaf*/
    int suffixIndex;
};

typedef struct SuffixTreeNode Node;

char text[100]; //Input string
Node *root = NULL; //Pointer to root node

/*lastNewNode will point to newly created internal node,
waiting for it's suffix link to be set, which might get
a new suffix link (other than root) in next extension of
same phase. lastNewNode will be set to NULL when last
newly created internal node (if there is any) got it's
suffix link reset to new internal node created in next
extension of same phase. */
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represented as input string character
index (not the character itself)*/
int activeEdge = -1;
int activeLength = 0;

// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string

Node *newNode(int start, int *end)
{
    Node *node = (Node*) malloc(sizeof(Node));
    int i;
```

```

for (i = 0; i < MAX_CHAR; i++)
    node->children[i] = NULL;

/*For root node, suffixLink will be set to NULL
For internal nodes, suffixLink will be set to root
by default in current extension and may change in
next extension*/
node->suffixLink = root;
node->start = start;
node->end = end;

/*suffixIndex will be set to -1 by default and
actual suffix index will be set later for leaves
at the end of all phases*/
node->suffixIndex = -1;
return node;
}

int edgeLength(Node *n) {
if(n == root)
return 0;
return *(n->end) - (n->start) + 1;
}

int walkDown(Node *currNode)
{
/*activePoint change for walk down (APCFWD) using
Skip/Count Trick (Trick 1). If activeLength is greater
than current edge length, set next internal node as
activeNode and adjust activeEdge and activeLength
accordingly to represent same activePoint*/
if (activeLength >= edgeLength(currNode))
{
activeEdge += edgeLength(currNode);
activeLength -= edgeLength(currNode);
activeNode = currNode;
return 1;
}
return 0;
}

void extendSuffixTree(int pos)
{
/*Extension Rule 1, this takes care of extending all
leaves created so far in tree*/
leafEnd = pos;

/*Increment remainingSuffixCount indicating that a
new suffix added to the list of suffixes yet to be
added in tree*/
remainingSuffixCount++;

/*set lastNewNode to NULL while starting a new phase,
indicating there is no internal node waiting for
it's suffix link reset in current phase*/
lastNewNode = NULL;

//Add all suffixes (yet to be added) one by one in tree
while(remainingSuffixCount > 0) {

if (activeLength == 0)
activeEdge = pos; //APCFALZ

// There is no outgoing edge starting with
// activeEdge from activeNode
if (activeNode->children[text[activeEdge]] == NULL)
{
/*Extension Rule 2 (A new leaf edge gets created)
activeNode->children[text[activeEdge]] =
newNode(pos, &leafEnd);

/*A new leaf edge is created in above line starting
from an existing node (the current activeNode), and
if there is any internal node waiting for it's suffix
link get reset, point the suffix link from that last
internal node to current activeNode. Then set lastNewNode
to NULL indicating no more node waiting for suffix link
reset.*/
if (lastNewNode != NULL)
{
lastNewNode->suffixLink = activeNode;
lastNewNode = NULL;
}
}

// There is an outgoing edge starting with activeEdge
// from activeNode
else
{
// Get the next node at the end of edge starting
// with activeEdge
Node *next = activeNode->children[text[activeEdge]];
if (walkDown(next))//Do walkdown
{
//Start from next node (the new activeNode)
continue;
}

/*Extension Rule 3 (current character being processed
is already on the edge)*/
if (text[next->start + activeLength] == text[pos])
{
//If a newly created node waiting for it's
suffix link to be set, then set suffix link
//of that waiting node to current active node
if(lastNewNode != NULL && activeNode != root)
{
lastNewNode->suffixLink = activeNode;
lastNewNode = NULL;
}

//APCFER3
activeLength++;
/*STOP all further processing in this phase
and move on to next phase*/
break;
}
}
}

```

```

/*We will be here when activePoint is in middle of
the edge being traversed and current character
being processed is not on the edge (we fall off
the tree). In this case, we add a new internal node
and a new leaf edge going out of that new node. This
is Extension Rule 2, where a new leaf edge and a new
internal node get created*/
splitEnd = (int*) malloc(sizeof(int));
*splitEnd = next->start + activeLength - 1;

//New internal node
Node *split = newNode(next->start, splitEnd);
activeNode->children[text[activeEdge]] = split;

//New leaf coming out of new internal node
split->children[text[pos]] = newNode(pos, &leafEnd);
next->start += activeLength;
split->children[text[next->start]] = next;

/*We got a new internal node here. If there is any
internal node created in last extensions of same
phase which is still waiting for its suffix link
reset, do it now.*/
if (lastNewNode != NULL)
{
/*suffixLink of lastNewNode points to current newly
created internal node*/
lastNewNode->suffixLink = split;
}

/*Make the current newly created internal node waiting
for its suffix link reset (which is pointing to root
at present). If we come across any other internal node
(existing or newly created) in next extension of same
phase, when a new leaf edge gets added (i.e. when
Extension Rule 2 applies is any of the next extension
of same phase) at that point, suffixLink of this node
will point to that internal node.*/
lastNewNode = split;
}

/* One suffix got added in tree, decrement the count of
suffixes yet to be added.*/
remainingSuffixCount--;
if (activeNode == root && activeLength > 0) //APCFER2C1
{
activeLength--;
activeEdge = pos - remainingSuffixCount + 1;
}
else if (activeNode != root) //APCFER2C2
{
activeNode = activeNode->suffixLink;
}
}
}

void print(int i, int j)
{
int k;
for (k=i; k<=j; k++)
printf("%c", text[k]);
}

//Print the suffix tree as well along with setting suffix index
//So tree will be printed in DFS manner
//Each edge along with its suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
if (n == NULL) return;

if (n->start != -1) //A non-root node
{
//Print the label on edge from parent to current node
//Uncomment below line to print suffix tree
// printf(n->start, "(n->end)");
}
int leaf = 1;
int i;
for (i = 0; i < MAX_CHAR; i++)
{
if (n->children[i] != NULL)
{
//Uncomment below two lines to print suffix index
// if (leaf == 1 && n->start != -1)
// printf(" [%d]n", n->suffixIndex);

//Current node is not a leaf as it has outgoing
//edges from it.
leaf = 0;
setSuffixIndexByDFS(n->children[i], labelHeight +
edgeLength(n->children[i]));
}
}
if (leaf == 1)
{
n->suffixIndex = size - labelHeight;
//Uncomment below line to print suffix index
//printf(" [%d]n", n->suffixIndex);
}
}

void freeSuffixTreeByPostOrder(Node *n)
{
if (n == NULL)
return;
int i;
for (i = 0; i < MAX_CHAR; i++)
{
if (n->children[i] != NULL)
{
freeSuffixTreeByPostOrder(n->children[i]);
}
}
if (n->suffixIndex == -1)
free(n->end);
free(n);
}

```

```

}

/*Build the suffix tree and print the edge labels along with
suffixIndex. suffixIndex for leaf edges will be >= 0 and
for non-leaf edges will be -1*/
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;

    /*Root is a special node with start and end indices as -1,
    as it has no parent from where an edge comes to root*/
    root = newNode(-1, rootEnd);

    activeNode = root; //First activeNode will be root
    for (i=0; i<size; i++)
        extendSuffixTree(i);
    int labelHeight = 0;
    setSuffixIndexByDFS(root, labelHeight);
}

void doTraversal(Node *n, int suffixArray[], int *idx)
{
    if(n == NULL)
    {
        return;
    }
    int i=0;
    if(n->suffixIndex == -1) //If it is internal node
    {
        for (i = 0; i < MAX_CHAR; i++)
        {
            if(n->children[i] != NULL)
            {
                doTraversal(n->children[i], suffixArray, idx);
            }
        }
    }
    //If it is Leaf node other than "$" label
    else if(n->suffixIndex > -1 && n->suffixIndex < size)
    {
        suffixArray[*idx++] = n->suffixIndex;
    }
}

void buildSuffixArray(int suffixArray[])
{
    int i = 0;
    for(i=0; i< size; i++)
        suffixArray[i] = -1;
    int idx = 0;
    doTraversal(root, suffixArray, &idx);
    printf("Suffix Array for String ");
    for(i=0; i<size; i++)
        printf("%c", text[i]);
    printf(" is: ");
    for(i=0; i<size; i++)
        printf("%d ", suffixArray[i]);
    printf("\n");
}

// driver program to test above functions
int main(int argc, char *argv[])
{
    strcpy(text, "banana$");
    buildSuffixTree();
    size--;
    int *suffixArray =(int*) malloc(sizeof(int) * size);
    buildSuffixArray(suffixArray);
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    free(suffixArray);

    strcpy(text, "GEEKSFORGEEKS$");
    buildSuffixTree();
    size--;
    suffixArray =(int*) malloc(sizeof(int) * size);
    buildSuffixArray(suffixArray);
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    free(suffixArray);

    strcpy(text, "AAAAAAAAA$");
    buildSuffixTree();
    size--;
    suffixArray =(int*) malloc(sizeof(int) * size);
    buildSuffixArray(suffixArray);
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    free(suffixArray);

    strcpy(text, "ABCDEFGG$");
    buildSuffixTree();
    size--;
    suffixArray =(int*) malloc(sizeof(int) * size);
    buildSuffixArray(suffixArray);
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    free(suffixArray);

    strcpy(text, "ABABABA$");
    buildSuffixTree();
    size--;
    suffixArray =(int*) malloc(sizeof(int) * size);
    buildSuffixArray(suffixArray);
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    free(suffixArray);

    strcpy(text, "abcabxabcd$");
    buildSuffixTree();
    size--;
    suffixArray =(int*) malloc(sizeof(int) * size);
    buildSuffixArray(suffixArray);

```

```
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);
free(suffixArray);

strcpy(text, "CCAAACCCGATT$");
buildSuffixTree();
size--;
suffixArray =(int*) malloc(sizeof(int) * size);
buildSuffixArray(suffixArray);
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);
free(suffixArray);

return 0;
}
```

Output:

```
Suffix Array for String banana is: 5 3 1 0 4 2
Suffix Array for String GEEKSFORGEES is: 9 1 10 2 5 8 0 11 3 6 7 12 4
Suffix Array for String AAAAAAAAA is: 9 8 7 6 5 4 3 2 1 0
Suffix Array for String ABCDEFG is: 0 1 2 3 4 5 6
Suffix Array for String ABABABA is: 6 4 2 0 5 3 1
Suffix Array for String abcabxabc is: 0 6 3 1 7 4 2 8 9 5
Suffix Array for String CCAAACCCGATTA is: 12 2 3 4 9 1 0 5 6 7 8 11 10
```

Ukkonen's Suffix Tree Construction takes $O(N)$ time and space to build suffix tree for a string of length N and after that, traversal of tree take $O(N)$ to build suffix array.

So overall, it's linear in time and space.

Can you see why traversal is $O(N)$?? Because a suffix tree of string of length N will have at most $N-1$ internal nodes and N leaves. Traversal of these nodes can be done in $O(N)$.

We have published following more articles on suffix tree applications:

- [Suffix Tree Application 1 – Substring Check](#)
- [Suffix Tree Application 2 – Searching All Patterns](#)
- [Suffix Tree Application 3 – Longest Repeated Substring](#)
- [Generalized Suffix Tree 1](#)
- [Suffix Tree Application 5 – Longest Common Substring](#)
- [Suffix Tree Application 6 – Longest Palindromic Substring](#)

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Pattern Searching
Suffix Tree

Suffix Tree Application 1 – Substring Check

Given a text string and a pattern string, check if pattern exists in text or not.

Few pattern searching algorithms (KMP, Rabin-Karp, Naive Algorithm, Finite Automata) are already discussed, which can be used for this check.

Here we will discuss suffix tree based algorithm.

As a prerequisite, we must know how to build a suffix tree in one or the other way.

Once we have a suffix tree built for given text, we need to traverse the tree from root to leaf against the characters in pattern. If we do not fall off the tree (i.e. there is a path from root to leaf or somewhere in middle) while traversal, then pattern exists in text as a substring.

Suffix Tree Application



Here we will build suffix tree using Ukkonen's Algorithm, discussed already as below:

[Ukkonen's Suffix Tree Construction – Part 1](#)

[Ukkonen's Suffix Tree Construction – Part 2](#)

[Ukkonen's Suffix Tree Construction – Part 3](#)

[Ukkonen's Suffix Tree Construction – Part 4](#)

[Ukkonen's Suffix Tree Construction – Part 5](#)

[Ukkonen's Suffix Tree Construction – Part 6](#)

The core traversal implementation for substring check, can be modified accordingly for suffix trees built by other algorithms.

```
// A C program for substring check using Ukkonen's Suffix Tree Construction
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];

    //pointer to other node via suffix link
    struct SuffixTreeNode *suffixLink;

    /*(start, end) interval specifies the edge, by which the
    node is connected to its parent node. Each edge will
    connect two nodes, one parent and one child, and
    (start, end) interval of a given edge will be stored
    in the child node. Lets say there are two nodes A and B
    connected by an edge with indices (5, 8) then this
    indices (5, 8) will be stored in node B. */
    int start;
    int *end;

    /*for leaf nodes, it stores the index of suffix for
    the path from root to leaf*/
    int suffixIndex;
};

typedef struct SuffixTreeNode Node;

char text[100]; //Input string
Node *root = NULL; //Pointer to root node
```



```

/*lastNewNode will point to newly created internal node,
waiting for it's suffix link to be set, which might get
a new suffix link (other than root) in next extension of
same phase. lastNewNode will be set to NULL when last
newly created internal node (if there is any) got it's
suffix link reset to new internal node created in next
extension of same phase. */
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represented as input string character
index (not the character itself)*/
int activeEdge = -1;
int activeLength = 0;

// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string

Node *newNode(int start, int *end)
{
    Node *node =(Node*) malloc(sizeof(Node));
    int i;
    for (i = 0; i < MAX_CHAR; i++)
        node->children[i] = NULL;

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

    /*suffixIndex will be set to -1 by default and
    actual suffix index will be set later for leaves
    at the end of all phases*/
    node->suffixIndex = -1;
    return node;
}

int edgeLength(Node *n) {
    if(n == root)
        return 0;
    return *(n->end) - (n->start) + 1;
}

int walkDown(Node *currNode)
{
    /*activePoint change for walk down (APCFWD) using
    Skip/Count Trick (Trick 1). If activeLength is greater
    than current edge length, set next internal node as
    activeNode and adjust activeEdge and activeLength
    accordingly to represent same activePoint*/
    if (activeLength >= edgeLength(currNode))
    {
        activeEdge += edgeLength(currNode);
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
        return 1;
    }
    return 0;
}

void extendSuffixTree(int pos)
{
    /*Extension Rule 1, this takes care of extending all
    leaves created so far in tree*/
    leafEnd = pos;

    /*Increment remainingSuffixCount indicating that a
    new suffix added to the list of suffixes yet to be
    added in tree*/
    remainingSuffixCount++;

    /*set lastNewNode to NULL while starting a new phase,
    indicating there is no internal node waiting for
    it's suffix link reset in current phase*/
    lastNewNode = NULL;

    //Add all suffixes (yet to be added) one by one in tree
    while(remainingSuffixCount > 0) {

        if (activeLength == 0)
            activeEdge = pos; //APCFALZ

        // There is no outgoing edge starting with
        // activeEdge from activeNode
        if (activeNode->children[text[activeEdge]] == NULL)
        {
            /*Extension Rule 2 (A new leaf edge gets created)
            activeNode->children[text[activeEdge]] =
            newNode(pos, &leafEnd);

            /*A new leaf edge is created in above line starting
            from an existing node (the current activeNode), and
            if there is any internal node waiting for it's suffix
            link get reset, point the suffix link from that last
            internal node to current activeNode. Then set lastNewNode
            to NULL indicating no more node waiting for suffix link
            reset*/
            if (lastNewNode != NULL)
            {
                lastNewNode->suffixLink = activeNode;
                lastNewNode = NULL;
            }
        }

        // There is an outgoing edge starting with activeEdge
        // from activeNode
        else
        {
            // Get the next node at the end of edge starting

```

```

// with activeEdge
Node *next = activeNode->children[text[activeEdge]];
if (walkDown(next))//Do walkdown
{
    //Start from next node (the new activeNode)
    continue;
}
/*Extension Rule 3 (current character being processed
is already on the edge)*/
if (text[next->start + activeLength] == text[pos])
{
    //If a newly created node waiting for it's
    //suffix link to be set, then set suffix link
    //of that waiting node to current active node
    if (lastNewNode != NULL && activeNode != root)
    {
        lastNewNode->suffixLink = activeNode;
        lastNewNode = NULL;
    }

    //APCFER3
    activeLength++;
    /*STOP all further processing in this phase
    and move on to next phase*/
    break;
}

/*We will be here when activePoint is in middle of
the edge being traversed and current character
being processed is not on the edge (we fall off
the tree). In this case, we add a new internal node
and a new leaf edge going out of that new node. This
is Extension Rule 2, where a new leaf edge and a new
internal node get created*/
splitEnd = (int*) malloc(sizeof(int));
*splitEnd = next->start + activeLength - 1;

//New internal node
Node *split = newNode(next->start, splitEnd);
activeNode->children[text[activeEdge]] = split;

//New leaf coming out of new internal node
split->children[text[pos]] = newNode(pos, &leafEnd);
next->start += activeLength;
split->children[text[next->start]] = next;

/*We got a new internal node here. If there is any
internal node created in last extensions of same
phase which is still waiting for it's suffix link
reset, do it now.*/
if (lastNewNode != NULL)
{
    /*suffixLink of lastNewNode points to current newly
    created internal node*/
    lastNewNode->suffixLink = split;
}

/*Make the current newly created internal node waiting
for it's suffix link reset (which is pointing to root
at present). If we come across any other internal node
(existing or newly created) in next extension of same
phase, when a new leaf edge gets added (i.e. when
Extension Rule 2 applies is any of the next extension
of same phase) at that point, suffixLink of this node
will point to that internal node.*/
lastNewNode = split;
}

/* One suffix got added in tree, decrement the count of
suffixes yet to be added.*/
remainingSuffixCount--;
if (activeNode == root && activeLength > 0) //APCFER2C1
{
    activeLength--;
    activeEdge = pos - remainingSuffixCount + 1;
}
else if (activeNode != root) //APCFER2C2
{
    activeNode = activeNode->suffixLink;
}
}
}

void print(int i, int j)
{
    int k;
    for (k=i; k<=j; k++)
        printf("%c", text[k]);
}

//Print the suffix tree as well along with setting suffix index
//So tree will be printed in DFS manner
//Each edge along with it's suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL) return;

    if (n->start != -1) //A non-root node
    {
        //Print the label on edge from parent to current node
        //Uncomment below line to print suffix tree
        // print(n->start, "(n->end)");
    }
    int leaf = 1;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            //Uncomment below two lines to print suffix index
            // if (leaf == 1 && n->start != -1)
            //     printf("%d\n", n->suffixIndex);

            //Current node is not a leaf as it has outgoing
            //edges from it.
            leaf = 0;
        }
    }
}

```

```

        setSuffixIndexByDFS(n->children[i], labelHeight +
            edgeLength(n->children[i]));
    }
}
if (leaf == 1)
{
    n->suffixIndex = size - labelHeight;
    //Uncomment below line to print suffix index
    //printf(" %d\n", n->suffixIndex);
}
}

void freeSuffixTreeByPostOrder(Node *n)
{
    if (n == NULL)
        return;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
    if (n->suffixIndex == -1)
        free(n->end);
    free(n);
}

/*Build the suffix tree and print the edge labels along with
suffixIndex. suffixIndex for leaf edges will be >= 0 and
for non-leaf edges will be -1*/
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;

    /*Root is a special node with start and end indices as -1,
    as it has no parent from where an edge comes to root*/
    root = newNode(-1, rootEnd);

    activeNode = root; //First activeNode will be root
    for (i=0; i<size; i++)
        extendSuffixTree(i);
    int labelHeight = 0;
    setSuffixIndexByDFS(root, labelHeight);
}

int traverseEdge(char *str, int idx, int start, int end)
{
    int k = 0;
    //Traverse the edge with character by character matching
    for(k=start; k<=end && str[idx] != '0'; k++, idx++)
    {
        if(text[k] != str[idx])
            return -1; // no match
    }
    if(str[idx] == '0')
        return 1; // match
    return 0; // more characters yet to match
}

int doTraversal(Node *n, char* str, int idx)
{
    if(n == NULL)
    {
        return -1; // no match
    }
    int res = -1;
    //If node n is not root node, then traverse edge
    //from node n's parent to node n.
    if(n->start != -1)
    {
        res = traverseEdge(str, idx, n->start, *(n->end));
        if(res != 0)
            return res; // match (res = 1) or no match (res = -1)
    }
    //Get the character index to search
    idx = idx + edgeLength(n);
    //If there is an edge from node n going out
    //with current character str[idx], travsrse that edge
    if(n->children[str[idx]] != NULL)
        return doTraversal(n->children[str[idx]], str, idx);
    else
        return -1; // no match
}

void checkForSubString(char* str)
{
    int res = doTraversal(root, str, 0);
    if(res == 1)
        printf("Pattern <%s> is a Substring\n", str);
    else
        printf("Pattern <%s> is NOT a Substring\n", str);
}

// driver program to test above functions
int main(int argc, char *argv[])
{
    strcpy(text, "THIS IS A TEST TEXT$");
    buildSuffixTree();

    checkForSubString("TEST");
    checkForSubString("A");
    checkForSubString(" ");
    checkForSubString("IS A");
    checkForSubString(" IS A ");
    checkForSubString("TEST1");
    checkForSubString("THIS IS GOOD");
    checkForSubString("TES");
    checkForSubString("TESA");
    checkForSubString("ISB");

    //Free the dynamically allocated memory

```

```

freeSuffixTreeByPostOrder(root);

return 0;
}

```

Output:

```

Pattern <TEST> is a Substring
Pattern <A> is a Substring
Pattern <> is a Substring
Pattern <IS A> is a Substring
Pattern <IS A > is a Substring
Pattern <TEST1> is NOT a Substring
Pattern <THIS IS GOOD> is NOT a Substring
Pattern <TES> is a Substring
Pattern <TESA> is NOT a Substring
Pattern <ISB> is NOT a Substring

```

Ukkonen's Suffix Tree Construction takes $O(N)$ time and space to build suffix tree for a string of length N and after that, traversal for substring check takes $O(M)$ for a pattern of length M .

With slight modification in traversal algorithm discussed here, we can answer following:

1. Find all occurrences of a given pattern P present in text T .
2. How to check if a pattern is prefix of a text?
3. How to check if a pattern is suffix of a text?

We have published following more articles on suffix tree applications:

- [Suffix Tree Application 2 – Searching All Patterns](#)
- [Suffix Tree Application 3 – Longest Repeated Substring](#)
- [Suffix Tree Application 4 – Build Linear Time Suffix Array](#)
- [Generalized Suffix Tree 1](#)
- [Suffix Tree Application 5 – Longest Common Substring](#)
- [Suffix Tree Application 6 – Longest Palindromic Substring](#)

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Pattern Searching
Pattern Searching
Suffix Tree

Suffix Tree Application 2 – Searching All Patterns

Given a text string and a pattern string, find all occurrences of the pattern in string.

Few pattern searching algorithms (KMP, [Rabin-Karp](#), [Naive Algorithm](#), [Finite Automata](#)) are already discussed, which can be used for this check.

Here we will discuss suffix tree based algorithm.

In the 1st Suffix Tree Application ([Substring Check](#)), we saw how to check whether a given pattern is substring of a text or not. It is advised to go through [Substring Check 1st](#).

In this article, we will go a bit further on same problem. If a pattern is substring of a text, then we will find all the positions on pattern in the text.

As a prerequisite, we must know how to build a suffix tree in one or the other way.

Here we will build suffix tree using Ukkonen's Algorithm, discussed already as below:

[Ukkonen's Suffix Tree Construction – Part 1](#)

[Ukkonen's Suffix Tree Construction – Part 2](#)

[Ukkonen's Suffix Tree Construction – Part 3](#)

[Ukkonen's Suffix Tree Construction – Part 4](#)

[Ukkonen's Suffix Tree Construction – Part 5](#)

[Ukkonen's Suffix Tree Construction – Part 6](#)

Lets look at following figure:

Suffix Tree Application



This is suffix tree for String "abcabxabc\$d", showing suffix indices and edge label indices (start, end). The (sub)string value on edges are shown only for explanatory purpose. We never store path label string in the tree.

Suffix Index of a path tells the index of a substring (starting from root) on that path.

Consider a path "bcd\$" in above tree with suffix index 7. It tells that substrings b, bc, bcd, bcd\$ are at index 7 in string.

Similarly path "bxabc\$d" with suffix index 4 tells that substrings b, bx, bxa, bxab, bxabc, bxabc, bxabc\$d are at index 4.

Similarly path "bcabxabc\$d" with suffix index 1 tells that substrings b, bc, bca, bcab, bcabx, bcabxa, bcabxab, bcabxabc, bcabxabcd, bcabxabcd\$ are at index 1.

If we see all the above three paths together, we can see that:

- Substring "b" is at indices 1, 4 and 7
- Substring "bc" is at indices 1 and 7

With above explanation, we should be able to see following:

- Substring "ab" is at indices 0, 3 and 6
- Substring "abc" is at indices 0 and 6
- Substring "c" is at indices 2 and 8
- Substring "xab" is at index 5
- Substring "d" is at index 9
- Substring "cd" is at index 8

.....

.....

Can you see how to find all the occurrences of a pattern in a string ?

1. 1st of all, check if the given pattern really exists in string or not (As we did in [Substring Check](#)). For this, traverse the suffix tree against the pattern.
2. If you find pattern in suffix tree (don't fall off the tree), then traverse the subtree below that point and find all suffix indices on leaf nodes. All those suffix indices will be pattern indices in string

Suffix Tree Application



```
// A C program to implement Ukkonen's Suffix Tree Construction
// And find all locations of a pattern in string
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];

    //pointer to other node via suffix link
    struct SuffixTreeNode *suffixLink;

    /*(start, end) interval specifies the edge, by which the
    node is connected to its parent node. Each edge will
    connect two nodes, one parent and one child, and
    (start, end) interval of a given edge will be stored
    in the child node. Lets say there are two nodes A and B
    connected by an edge with indices (5, 8) then this
    indices (5, 8) will be stored in node B. */
    int start;
    int *end;

    /*for leaf nodes, it stores the index of suffix for
    the path from root to leaf*/
    int suffixIndex;
};

typedef struct SuffixTreeNode Node;

char text[100]; //Input string
Node *root = NULL; //Pointer to root node

/*lastNewNode will point to newly created internal node,
waiting for its suffix link to be set, which might get
a new suffix link (other than root) in next extension of
same phase. lastNewNode will be set to NULL when last
newly created internal node (if there is any) got its
suffix link reset to new internal node created in next
extension of same phase. */
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represented as input string character
index (not the character itself)*/
int activeEdge = -1;
int activeLength = 0;

// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string

Node *newNode(int start, int *end)
{
    Node *node = (Node *) malloc(sizeof(Node));
    int i;
    for (i = 0; i < MAX_CHAR; i++)
        node->children[i] = NULL;

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

    /*suffixIndex will be set to -1 by default and
    actual suffix index will be set later for leaves
    at the end of all phases*/
    node->suffixIndex = -1;
    return node;
}

int edgeLength(Node *n) {
    if (n == root)
        return 0;
    return *(n->end) - (n->start) + 1;
}

int walkDown(Node *currNode)
{
    /*activePoint change for walk down (APCFWD) using
    Skip/Count Trick (Trick 1). If activeLength is greater
    than current edge length, set next internal node as
    activeNode and adjust activeEdge and activeLength
    accordingly to represent same activePoint*/
    if (activeLength >= edgeLength(currNode))
    {
        activeEdge += edgeLength(currNode);
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
        return 1;
    }
}
```

```

return 0;
}

void extendSuffixTree(int pos)
{
    /*Extension Rule 1, this takes care of extending all
    leaves created so far in tree*/
    leafEnd = pos;

    /*Increment remainingSuffixCount indicating that a
    new suffix added to the list of suffixes yet to be
    added in tree*/
    remainingSuffixCount++;

    /*set lastNewNode to NULL while starting a new phase,
    indicating there is no internal node waiting for
    it's suffix link reset in current phase*/
    lastNewNode = NULL;

    //Add all suffixes (yet to be added) one by one in tree
    while(remainingSuffixCount > 0) {

        if (activeLength == 0)
            activeEdge = pos; //APCFALZ

        // There is no outgoing edge starting with
        // activeEdge from activeNode
        if (activeNode->children[text[activeEdge]] == NULL)
        {
            /*Extension Rule 2 (A new leaf edge gets created)
            activeNode->children[text[activeEdge]] =
                newNode(pos, &leafEnd);

            /*A new leaf edge is created in above line starting
            from an existing node (the current activeNode), and
            if there is any internal node waiting for it's suffix
            link get reset, point the suffix link from that last
            internal node to current activeNode. Then set lastNewNode
            to NULL indicating no more node waiting for suffix link
            reset*/
            if (lastNewNode != NULL)
            {
                lastNewNode->suffixLink = activeNode;
                lastNewNode = NULL;
            }
        }

        // There is an outgoing edge starting with activeEdge
        // from activeNode
        else
        {
            // Get the next node at the end of edge starting
            // with activeEdge
            Node *next = activeNode->children[text[activeEdge]];
            if (walkDown(next))//Do walkdown
            {
                //Start from next node (the new activeNode)
                continue;
            }

            /*Extension Rule 3 (current character being processed
            is already on the edge)*/
            if (text[next->start + activeLength] == text[pos])
            {
                //If a newly created node waiting for it's
                //suffix link to be set, then set suffix link
                //of that waiting node to current active node
                if(lastNewNode != NULL && activeNode != root)
                {
                    lastNewNode->suffixLink = activeNode;
                    lastNewNode = NULL;
                }

                //APCFER3
                activeLength++;
                /*STOP all further processing in this phase
                and move on to next phase*/
                break;
            }

            /*We will be here when activePoint is in middle of
            the edge being traversed and current character
            being processed is not on the edge (we fall off
            the tree). In this case, we add a new internal node
            and a new leaf edge going out of that new node. This
            is Extension Rule 2, where a new leaf edge and a new
            internal node get created*/
            splitEnd = (int*) malloc(sizeof(int));
            *splitEnd = next->start + activeLength - 1;

            //New internal node
            Node *split = newNode(next->start, splitEnd);
            activeNode->children[text[activeEdge]] = split;

            //New leaf coming out of new internal node
            split->children[text[pos]] = newNode(pos, &leafEnd);
            next->start += activeLength;
            split->children[text[next->start]] = next;

            /*We got a new internal node here. If there is any
            internal node created in last extensions of same
            phase which is still waiting for it's suffix link
            reset, do it now.*/
            if (lastNewNode != NULL)
            {
                /*suffixLink of lastNewNode points to current newly
                created internal node*/
                lastNewNode->suffixLink = split;
            }

            /*Make the current newly created internal node waiting
            for it's suffix link reset (which is pointing to root
            at present). If we come across any other internal node
            (existing or newly created) in next extension of same
            phase, when a new leaf edge gets added (i.e. when
            Extension Rule 2 applies is any of the next extension
            of same phase) at that point, suffixLink of this node
            will point to that internal node.*/

```

```

        lastNewNode = split;
    }

    /* One suffix got added in tree, decrement the count of
    suffixes yet to be added.*/
    remainingSuffixCount--;
    if (activeNode == root && activeLength > 0) //APCFER2C1
    {
        activeLength--;
        activeEdge = pos - remainingSuffixCount + 1;
    }
    else if (activeNode != root) //APCFER2C2
    {
        activeNode = activeNode->suffixLink;
    }
    }
}

void print(int i, int j)
{
    int k;
    for (k=i; k<=j; k++)
        printf("%c", text[k]);
}

//Print the suffix tree as well along with setting suffix index
//So tree will be printed in DFS manner
//Each edge along with its suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL) return;

    if (n->start != -1) //A non-root node
    {
        //Print the label on edge from parent to current node
        //Uncomment below line to print suffix tree
        // printf("n->start, "(n->end));
    }
    int leaf = 1;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            //Uncomment below two lines to print suffix index
            // if (leaf == 1 && n->start != -1)
            //   printf(" [%d]n", n->suffixIndex);

            //Current node is not a leaf as it has outgoing
            //edges from it.
            leaf = 0;
            setSuffixIndexByDFS(n->children[i], labelHeight +
                               edgeLength(n->children[i]));
        }
    }
    if (leaf == 1)
    {
        n->suffixIndex = size - labelHeight;
        //Uncomment below line to print suffix index
        //printf(" [%d]n", n->suffixIndex);
    }
}

void freeSuffixTreeByPostOrder(Node *n)
{
    if (n == NULL)
        return;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
    if (n->suffixIndex == -1)
        free(n->end);
    free(n);
}

/*Build the suffix tree and print the edge labels along with
suffixIndex. suffixIndex for leaf edges will be >= 0 and
for non-leaf edges will be -1*/
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = -1;

    /*Root is a special node with start and end indices as -1,
    as it has no parent from where an edge comes to root*/
    root = newNode(-1, rootEnd);

    activeNode = root; //First activeNode will be root
    for (i=0; i<size; i++)
        extendSuffixTree(i);
    int labelHeight = 0;
    setSuffixIndexByDFS(root, labelHeight);
}

int traverseEdge(char *str, int idx, int start, int end)
{
    int k = 0;
    //Traverse the edge with character by character matching
    for(k=start; k<=end && str[idx] != '\0'; k++, idx++)
    {
        if(text[k] != str[idx])
            return -1; // no match
    }
    if(str[idx] == '\0')
        return 1; // match
    return 0; // more characters yet to match
}

int doTraversalToCountLeaf(Node *n)

```

```

{
    if(n == NULL)
        return 0;
    if(n->suffixIndex > -1)
    {
        printf("\nFound at position: %d", n->suffixIndex);
        return 1;
    }
    int count = 0;
    int i = 0;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if(n->children[i] != NULL)
        {
            count += doTraversalToCountLeaf(n->children[i]);
        }
    }
    return count;
}

int countLeaf(Node *n)
{
    if(n == NULL)
        return 0;
    return doTraversalToCountLeaf(n);
}

int doTraversal(Node *n, char *str, int idx)
{
    if(n == NULL)
    {
        return -1; // no match
    }
    int res = -1;
    //If node n is not root node, then traverse edge
    //from node n's parent to node n.
    if(n->start != -1)
    {
        res = traverseEdge(str, idx, n->start, "(n->end)");
        if(res == -1) //no match
            return -1;
        if(res == 1) //match
        {
            if(n->suffixIndex > -1)
                printf("nsubstring count: 1 and position: %d",
                    n->suffixIndex);

            else
                printf("nsubstring count: %d", countLeaf(n));
            return 1;
        }
    }
    //Get the character index to search
    idx = idx + edgeLength(n);
    //If there is an edge from node n going out
    //with current character str[idx], travsr that edge
    if(n->children[str[idx]] != NULL)
        return doTraversal(n->children[str[idx]], str, idx);
    else
        return -1; // no match
}

void checkForSubString(char* str)
{
    int res = doTraversal(root, str, 0);
    if(res == 1)
        printf("\nPattern <%s> is a Substring\n", str);
    else
        printf("\nPattern <%s> is NOT a Substring\n", str);
}

// driver program to test above functions
int main(int argc, char *argv[])
{
    strcpy(text, "GEEKSFORGEEKS$");
    buildSuffixTree();
    printf("\nText: GEEKSFORGEEKS, Pattern to search: GEEKS");
    checkForSubString("GEEKS");
    printf("\n\nText: GEEKSFORGEEKS, Pattern to search: GEEK1");
    checkForSubString("GEEK1");
    printf("\n\nText: GEEKSFORGEEKS, Pattern to search: FOR");
    checkForSubString("FOR");
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "AABAACAADAABAAABAA$");
    buildSuffixTree();
    printf("\n\nText: AABAACAADAABAAABAA, Pattern to search: AABA");
    checkForSubString("AABA");
    printf("\n\nText: AABAACAADAABAAABAA, Pattern to search: AA");
    checkForSubString("AA");
    printf("\n\nText: AABAACAADAABAAABAA, Pattern to search: AAE");
    checkForSubString("AAE");
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "AAAAAAAAA$");
    buildSuffixTree();
    printf("\n\nText: AAAAAAAAA, Pattern to search: AAAA");
    checkForSubString("AAAA");
    printf("\n\nText: AAAAAAAAA, Pattern to search: AA");
    checkForSubString("AA");
    printf("\n\nText: AAAAAAAAA, Pattern to search: A");
    checkForSubString("A");
    printf("\n\nText: AAAAAAAAA, Pattern to search: AB");
    checkForSubString("AB");
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    return 0;
}

```

Output:

```

Text: GEEKSFORGEEKS, Pattern to search: GEEKS
Found at position: 8

```


Found at position: 0
substring count: 2
Pattern <GEEKS> is a Substring

Text: GEEKSFORGEEKS, Pattern to search: GEEK1
Pattern <GEEK1> is NOT a Substring

Text: GEEKSFORGEEKS, Pattern to search: FOR
substring count: 1 and position: 5
Pattern <FOR> is a Substring

Text: AABAACAADAABAAABAA, Pattern to search: AABA
Found at position: 13
Found at position: 9
Found at position: 0
substring count: 3
Pattern <AABA> is a Substring

Text: AABAACAADAABAAABAA, Pattern to search: AA
Found at position: 16
Found at position: 12
Found at position: 13
Found at position: 9
Found at position: 0
Found at position: 3
Found at position: 6
substring count: 7
Pattern <AA> is a Substring

Text: AABAACAADAABAAABAA, Pattern to search: AAE
Pattern <AAE> is NOT a Substring

Text: AAAAAAAAA, Pattern to search: AAAA
Found at position: 5
Found at position: 4
Found at position: 3
Found at position: 2
Found at position: 1
Found at position: 0
substring count: 6
Pattern <AAAA> is a Substring

Text: AAAAAAAAA, Pattern to search: AA
Found at position: 7
Found at position: 6
Found at position: 5
Found at position: 4
Found at position: 3
Found at position: 2
Found at position: 1
Found at position: 0
substring count: 8
Pattern <AA> is a Substring

Text: AAAAAAAAA, Pattern to search: A
Found at position: 8
Found at position: 7
Found at position: 6
Found at position: 5
Found at position: 4
Found at position: 3
Found at position: 2
Found at position: 1
Found at position: 0
substring count: 9
Pattern <A> is a Substring

Text: AAAAAAAAA, Pattern to search: AB
Pattern <AB> is NOT a Substring

Ukkonen's Suffix Tree Construction takes $O(N)$ time and space to build suffix tree for a string of length N and after that, traversal for substring check takes $O(M)$ for a pattern of length M and then if there are Z occurrences of the pattern, it will take $O(Z)$ to find indices of all those Z occurrences.

Overall pattern complexity is linear: $O(M + Z)$.

A bit more detailed analysis

How many internal nodes will there in a suffix tree of string of length N ??

Answer: $N-1$ (Why ??)

There will be N suffixes in a string of length N .

Each suffix will have one leaf.

So a suffix tree of string of length N will have N leaves.

As each internal node has at least 2 children, an N -leaf suffix tree has at most $N-1$ internal nodes.

If a pattern occurs Z times in string, means it will be part of Z suffixes, so there will be Z leaves below in point (internal node and in between edge) where pattern match ends in tree and so subtree with Z leaves below that point will have $Z-1$ internal nodes. A tree with Z leaves can be traversed in $O(Z)$ time.

Overall pattern complexity is linear: $O(M + Z)$.

For a given pattern, Z (the number of occurrences) can be atmost N .

So worst case complexity can be: $O(M + N)$ if Z is close/equal to N (A tree traversal with N nodes take $O(N)$ time).

Followup questions:

1. Check if a pattern is prefix of a text?
2. Check if a pattern is suffix of a text?

We have published following more articles on suffix tree applications:

- [Suffix Tree Application 1 – Substring Check](#)
- [Suffix Tree Application 3 – Longest Repeated Substring](#)
- [Suffix Tree Application 4 – Build Linear Time Suffix Array](#)
- [Generalized Suffix Tree 1](#)
- [Suffix Tree Application 5 – Longest Common Substring](#)
- [Suffix Tree Application 6 – Longest Palindromic Substring](#)

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Suffix Tree Application 3 – Longest Repeated Substring

Given a text string, find **Longest Repeated Substring** in the text. If there are more than one Longest Repeated Substrings, get any one of them.

```
Longest Repeated Substring in GEEKSFORGEES is: GEEKS
Longest Repeated Substring in AAAAAAAAAA is: AAAAAAAAA
Longest Repeated Substring in ABCDEFG is: No repeated substring
Longest Repeated Substring in ABABABA is: ABABA
Longest Repeated Substring in ATCGATCGA is: ATCGA
Longest Repeated Substring in banana is: ana
Longest Repeated Substring in abcpqrabppq is: ab (pq is another LRS here)
```

This problem can be solved by different approaches with varying time and space complexities. Here we will discuss Suffix Tree approach (3rd Suffix Tree Application). Other approaches will be discussed soon.

As a prerequisite, we must know how to build a suffix tree in one or the other way.

Here we will build suffix tree using Ukkonen's Algorithm, discussed already as below:

[Ukkonen's Suffix Tree Construction – Part 1](#)

[Ukkonen's Suffix Tree Construction – Part 2](#)

[Ukkonen's Suffix Tree Construction – Part 3](#)

[Ukkonen's Suffix Tree Construction – Part 4](#)

[Ukkonen's Suffix Tree Construction – Part 5](#)

[Ukkonen's Suffix Tree Construction – Part 6](#)

Lets look at following figure:

Suffix Tree Application



This is suffix tree for string "ABABABA\$".

In this string, following substrings are repeated:

A, B, AB, BA, ABA, BAB, ABAB, BABA, ABABA

And Longest Repeated Substring is ABABA.

In a suffix tree, one node can't have more than one outgoing edge starting with same character, and so if there are repeated substring in the text, they will share on same path and that path in suffix tree will go through one or more internal node(s) down the tree (below the point where substring ends on that path).

In above figure, we can see that

- Path with Substring "A" has three internal nodes down the tree
- Path with Substring "AB" has two internal nodes down the tree
- Path with Substring "ABA" has two internal nodes down the tree
- Path with Substring "ABAB" has one internal node down the tree
- Path with Substring "ABABA" has one internal node down the tree
- Path with Substring "B" has two internal nodes down the tree
- Path with Substring "BA" has two internal nodes down the tree
- Path with Substring "BAB" has one internal node down the tree
- Path with Substring "BABA" has one internal node down the tree

All above substrings are repeated.

Substrings ABABAB, ABABABA, BABAB, BABABA have no internal node down the tree (after the point where substring end on the path), and so these are not repeated.

Can you see how to find longest repeated substring ??

We can see in figure that, longest repeated substring will end at the internal node which is farthest from the root (i.e. deepest node in the tree), because length of substring is the path label length from root to that internal node.

So finding longest repeated substring boils down to finding the deepest node in suffix tree and then get the path label from root to that deepest internal node.

```
// A C program to implement Ukkonen's Suffix Tree Construction
// And then find Longest Repeated Substring
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];

    //pointer to other node via suffix link
    struct SuffixTreeNode *suffixLink;

    /*(start, end) interval specifies the edge, by which the
    node is connected to its parent node. Each edge will
    connect two nodes, one parent and one child, and
    (start, end) interval of a given edge will be stored
    in the child node. Lets say there are two nodes A and B
    connected by an edge with indices (5, 8) then this
    indices (5, 8) will be stored in node B. */
    int start;
    int *end;

    /*for leaf nodes, it stores the index of suffix for
    the path from root to leaf*/
    int suffixIndex;
};

typedef struct SuffixTreeNode Node;

char text[100]; //Input string
```

```

Node *root = NULL; //Pointer to root node

/*lastNewNode will point to newly created internal node,
waiting for it's suffix link to be set, which might get
a new suffix link (other than root) in next extension of
same phase. lastNewNode will be set to NULL when last
newly created internal node (if there is any) got it's
suffix link reset to new internal node created in next
extension of same phase. */
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represented as input string character
index (not the character itself)*/
int activeEdge = -1;
int activeLength = 0;

// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string

Node *newNode(int start, int *end)
{
    Node *node =(Node*) malloc(sizeof(Node));
    int i;
    for (i = 0; i < MAX_CHAR; i++)
        node->children[i] = NULL;

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

    /*suffixIndex will be set to -1 by default and
    actual suffix index will be set later for leaves
    at the end of all phases*/
    node->suffixIndex = -1;
    return node;
}

int edgeLength(Node *n) {
    if(n == root)
        return 0;
    return *(n->end) - (n->start) + 1;
}

int walkDown(Node *currNode)
{
    /*activePoint change for walk down (APCFWD) using
    Skip/Count Trick (Trick 1). If activeLength is greater
    than current edge length, set next internal node as
    activeNode and adjust activeEdge and activeLength
    accordingly to represent same activePoint*/
    if (activeLength >= edgeLength(currNode))
    {
        activeEdge += edgeLength(currNode);
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
        return 1;
    }
    return 0;
}

void extendSuffixTree(int pos)
{
    /*Extension Rule 1, this takes care of extending all
    leaves created so far in tree*/
    leafEnd = pos;

    /*Increment remainingSuffixCount indicating that a
    new suffix added to the list of suffixes yet to be
    added in tree*/
    remainingSuffixCount++;

    /*set lastNewNode to NULL while starting a new phase,
    indicating there is no internal node waiting for
    it's suffix link reset in current phase*/
    lastNewNode = NULL;

    //Add all suffixes (yet to be added) one by one in tree
    while(remainingSuffixCount > 0) {

        if (activeLength == 0)
            activeEdge = pos; //APCFALZ

        // There is no outgoing edge starting with
        // activeEdge from activeNode
        if (activeNode->children[text[activeEdge]] == NULL)
        {
            /*Extension Rule 2 (A new leaf edge gets created)
            activeNode->children[text[activeEdge]] =
            newNode(pos, &leafEnd);

            /*A new leaf edge is created in above line starting
            from an existing node (the current activeNode), and
            if there is any internal node waiting for it's suffix
            link get reset, point the suffix link from that last
            internal node to current activeNode. Then set lastNewNode
            to NULL indicating no more node waiting for suffix link
            reset*/
            if (lastNewNode != NULL)
            {
                lastNewNode->suffixLink = activeNode;
                lastNewNode = NULL;
            }
        }
        // There is an outgoing edge starting with activeEdge
        // from activeNode
        else

```

```

{
    // Get the next node at the end of edge starting
    // with activeEdge
    Node *next = activeNode->children[text[activeEdge]];
    if (walkDown(next))//Do walkdown
    {
        //Start from next node (the new activeNode)
        continue;
    }
    /*Extension Rule 3 (current character being processed
    is already on the edge)*/
    if (text[next->start + activeLength] == text[pos])
    {
        //If a newly created node waiting for it's
        //suffix link to be set, then set suffix link
        //of that waiting node to current active node
        if (lastNewNode != NULL && activeNode != root)
        {
            lastNewNode->suffixLink = activeNode;
            lastNewNode = NULL;
        }

        //APCFER3
        activeLength++;
        /*STOP all further processing in this phase
        and move on to next phase*/
        break;
    }

    /*We will be here when activePoint is in middle of
    the edge being traversed and current character
    being processed is not on the edge (we fall off
    the tree). In this case, we add a new internal node
    and a new leaf edge going out of that new node. This
    is Extension Rule 2, where a new leaf edge and a new
    internal node get created*/
    splitEnd = (int*) malloc(sizeof(int));
    *splitEnd = next->start + activeLength - 1;

    //New internal node
    Node *split = newNode(next->start, splitEnd);
    activeNode->children[text[activeEdge]] = split;

    //New leaf coming out of new internal node
    split->children[text[pos]] = newNode(pos, &leafEnd);
    next->start += activeLength;
    split->children[text[next->start]] = next;

    /*We got a new internal node here. If there is any
    internal node created in last extensions of same
    phase which is still waiting for it's suffix link
    reset, do it now.*/
    if (lastNewNode != NULL)
    {
        /*suffixLink of lastNewNode points to current newly
        created internal node*/
        lastNewNode->suffixLink = split;
    }

    /*Make the current newly created internal node waiting
    for it's suffix link reset (which is pointing to root
    at present). If we come across any other internal node
    (existing or newly created) in next extension of same
    phase, when a new leaf edge gets added (i.e. when
    Extension Rule 2 applies is any of the next extension
    of same phase) at that point, suffixLink of this node
    will point to that internal node.*/
    lastNewNode = split;
}

/* One suffix got added in tree, decrement the count of
suffixes yet to be added.*/
remainingSuffixCount--;
if (activeNode == root && activeLength > 0) //APCFER2C1
{
    activeLength--;
    activeEdge = pos - remainingSuffixCount + 1;
}
else if (activeNode != root) //APCFER2C2
{
    activeNode = activeNode->suffixLink;
}
}

void print(int i, int j)
{
    int k;
    for (k=i; k<=j; k++)
        printf("%c", text[k]);
}

//Print the suffix tree as well along with setting suffix index
//So tree will be printed in DFS manner
//Each edge along with it's suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL) return;

    if (n->start != -1) //A non-root node
    {
        //Print the label on edge from parent to current node
        //Uncomment below line to print suffix tree
        // print(n->start, "(n->end)");
    }
    int leaf = 1;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            //Uncomment below two lines to print suffix index
            // if (leaf == 1 && n->start != -1)
            //     printf("%d\n", n->suffixIndex);

            //Current node is not a leaf as it has outgoing

```

```

//edges from it.
leaf = 0;
setSuffixIndexByDFS(n->children[i], labelHeight +
    edgeLength(n->children[i]));
}
}
if (leaf == 1)
{
    n->suffixIndex = size - labelHeight;
//Uncomment below line to print suffix index
//printf(" [%d]n", n->suffixIndex);
}
}

void freeSuffixTreeByPostOrder(Node *n)
{
    if (n == NULL)
        return;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
    if (n->suffixIndex == -1)
        free(n->end);
    free(n);
}

/*Build the suffix tree and print the edge labels along with
suffixIndex. suffixIndex for leaf edges will be >= 0 and
for non-leaf edges will be -1*/
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;

    /*Root is a special node with start and end indices as -1,
as it has no parent from where an edge comes to root*/
    root = newNode(-1, rootEnd);

    activeNode = root; //First activeNode will be root
    for (i=0; i<size; i++)
        extendSuffixTree(i);
    int labelHeight = 0;
    setSuffixIndexByDFS(root, labelHeight);
}

void doTraversal(Node *n, int labelHeight, int* maxHeight,
int* substringStartIndex)
{
    if(n == NULL)
    {
        return;
    }
    int i=0;
    if(n->suffixIndex == -1) //If it is internal node
    {
        for (i = 0; i < MAX_CHAR; i++)
        {
            if(n->children[i] != NULL)
            {
                doTraversal(n->children[i], labelHeight +
                    edgeLength(n->children[i]), maxHeight,
                    substringStartIndex);
            }
        }
    }
    else if(n->suffixIndex > -1 &&
        (*maxHeight < labelHeight - edgeLength(n)))
    {
        *maxHeight = labelHeight - edgeLength(n);
        *substringStartIndex = n->suffixIndex;
    }
}

void getLongestRepeatedSubstring()
{
    int maxHeight = 0;
    int substringStartIndex = 0;
    doTraversal(root, 0, &maxHeight, &substringStartIndex);
    // printf("maxHeight %d, substringStartIndex %d\n", maxHeight,
    //     substringStartIndex);
    printf("Longest Repeated Substring in %s is: ", text);
    int k;
    for (k=0; k<maxHeight; k++)
        printf("%c", text[k + substringStartIndex]);
    if(k == 0)
        printf("No repeated substring");
    printf("\n");
}

// driver program to test above functions
int main(int argc, char *argv[])
{
    strcpy(text, "GEEKSFORGEEKS$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "AAAAAAAAA$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "ABCDEFG$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
}

```

```

strcpy(text, "ABABABA$");
buildSuffixTree();
getLongestRepeatedSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

strcpy(text, "ATCGATCGA$");
buildSuffixTree();
getLongestRepeatedSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

strcpy(text, "banana$");
buildSuffixTree();
getLongestRepeatedSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

strcpy(text, "abcpqrabppqp$");
buildSuffixTree();
getLongestRepeatedSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

strcpy(text, "pqrpqabab$");
buildSuffixTree();
getLongestRepeatedSubstring();
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);

return 0;
}

```

Output:

```

Longest Repeated Substring in GEEKSFORGEEKS$ is: GEEKS
Longest Repeated Substring in AAAAAAAAAA$ is: AAAAAAAAA
Longest Repeated Substring in ABCDEFG$ is: No repeated substring
Longest Repeated Substring in ABABABA$ is: ABABA
Longest Repeated Substring in ATCGATCGA$ is: ATCGA
Longest Repeated Substring in banana$ is: ana
Longest Repeated Substring in abcpqrabppqp$ is: ab
Longest Repeated Substring in pqrpqabab$ is: ab

```

In case of multiple LRS (As we see in last two test cases), this implementation prints the LRS which comes 1st lexicographically.

Ukkonen's Suffix Tree Construction takes $O(N)$ time and space to build suffix tree for a string of length N and after that finding deepest node will take $O(N)$.

So it is linear in time and space.

Followup questions:

1. Find all repeated substrings in given text
2. Find all unique substrings in given text
3. Find all repeated substrings of a given length
4. Find all unique substrings of a given length
5. In case of multiple LRS in text, find the one which occurs most number of times

All these problems can be solved in linear time with few changes in above implementation.

We have published following more articles on suffix tree applications:

- [Suffix Tree Application 1 – Substring Check](#)
- [Suffix Tree Application 2 – Searching All Patterns](#)
- [Suffix Tree Application 4 – Build Linear Time Suffix Array](#)
- [Generalized Suffix Tree 1](#)
- [Suffix Tree Application 5 – Longest Common Substring](#)
- [Suffix Tree Application 6 – Longest Palindromic Substring](#)

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Pattern Searching
Pattern Searching
Suffix Tree

Suffix Tree Application 6 – Longest Palindromic Substring

Given a string, find the longest substring which is palindrome.

We have already discussed Naïve $O(n^3)$, quadratic $O(n^2)$ and linear $O(n)$ approaches in [Set 1](#), [Set 2](#) and [Manacher's Algorithm](#).

In this article, we will discuss another linear time approach based on suffix tree.

If given string is S , then approach is following:

- Reverse the string S (say reversed string is R)
- Get [Longest Common Substring](#) of S and R **given that LCS in S and R must be from same position in S**

Can you see why we say that **LCS in R and S must be from same position in S** ?

Let's look at following examples:

- For $S = xababayz$ and $R = zyababax$, LCS and LPS both are ababa (SAME)
- For $S = abacdgdxcaba$ and $R = abacdgdxcaba$, LCS is *abacd* and LPS is *aba* (DIFFERENT)
- For $S = pqrqpabcdfgdcba$ and $R = abcdgfdcbapqrp$, LCS and LPS both are *pqrqp* (SAME)
- For $S = pqpabcdfghfdcba$ and $R = abcdghfdcbapqp$, LCS is *abcdf* and LPS is *pqp* (DIFFERENT)

We can see that LCS and LPS are not same always. When they are different ?

When S has a reversed copy of a non-palindromic substring in it which is of same or longer length than LPS in S , then LCS and LPS will be different.

In 2nd example above ($S = abacdgdxcaba$), for substring *abacd*, there exists a reverse copy *dcaba* in S , which is of longer length than LPS *aba* and so LPS and LCS are different here. Same is the scenario in 4th example.

To handle this scenario we say that LPS in S is same as LCS in S and R **given that LCS in R and S must be from same position in S** .

If we look at 2nd example again, substring *aba* in R comes from exactly same position in S as substring *aba* in S which is ZERO (0th index) and so this is LPS.

The Position Constraint:

(Click to see it clearly)



We will refer string S index as forward index (S_i) and string R index as reverse index (R_i).

Based on above figure, a character with index i (forward index) in a string S of length N, will be at index $N-1-i$ (reverse index) in it's reversed string R.

If we take a substring of length L in string S with starting index i and ending index j ($j = i+L-1$), then in it's reversed string R, the reversed substring of the same will start at index $N-1-j$ and will end at index $N-1-i$.

If there is a common substring of length L at indices S_i (forward index) and R_i (reverse index) in S and R, then these will come from same position in S if $R_i = (N-1) - (S_i + L - 1)$ where N is string length.

So to find LPS of string S, we find longest common string of S and R where both substrings satisfy above constraint, i.e. if substring in S is at index S_i , then same substring should be in R at index $(N-1) - (S_i + L - 1)$. If this is not the case, then this substring is not LPS candidate.

Naive [$O(N^2M)$] and Dynamic Programming [$O(N^2M)$] approaches to find LCS of two strings are already discussed [here](#) which can be extended to add position constraint to give LPS of a given string.

Now we will discuss suffix tree approach which is nothing but an extension to [Suffix Tree LCS approach](#) where we will add the position constraint.

While finding LCS of two strings X and Y, we just take deepest node marked as XY (i.e. the node which has suffixes from both strings as it's children).

While finding LPS of string S, we will again find LCS of S and R with a condition that the common substring should satisfy the position constraint (the common substring should come from same position in S). To verify position constraint, we need to know all forward and reverse indices on each internal node (i.e. the suffix indices of all leaf children below the internal nodes).

In [Generalized Suffix Tree](#) of $S\#R\$, a$ substring on the path from root to an internal node is a common substring if the internal node has suffixes from both strings S and R. The index of the common substring in S and R can be found by looking at suffix index at respective leaf node.

If string $S\#$ is of length N then:

- If suffix index of a leaf is less than N, then that suffix belongs to S and same suffix index will become forward index of all ancestor nodes
- If suffix index of a leaf is greater than N, then that suffix belongs to R and reverse index for all ancestor nodes will be $N - \text{suffix index}$

Let's take string $S = cabbabbb$. The figure below is [Generalized Suffix Tree](#) for $cabbabbb\#bbaabbbac\$$ where we have shown forward and reverse indices of all children suffixes on all internal nodes (except root).

Forward indices are in Parentheses () and reverse indices are in square bracket [].

(Click to see it clearly)

Suffix Tree Application



In above figure, all leaf nodes will have one forward or reverse index depending on which string (S or R) they belong to. Then children's forward or reverse indices propagate to the parent.

Look at the figure to understand what would be the forward or reverse index on a leaf with a given suffix index. At the bottom of figure, it is shown that leaves with suffix indices from 0 to 8 will get same values (0 to 8) as their forward index in S and leaves with suffix indices 9 to 17 will get reverse index in R from 0 to 8.

For example, the highlighted internal node has two children with suffix indices 2 and 9. Leaf with suffix index 2 is from position 2 in S and so it's forward index is 2 and shown in (). Leaf with suffix index 9 is from position 0 in R and so it's reverse index is 0 and shown in []. These indices propagate to parent and the parent has one leaf with suffix index 14 for which reverse index is 4. So on this parent node forward index is (2) and reverse index is [0,4]. And in same way, we should be able to understand the how forward and reverse indices are calculated on all nodes.

In above figure, all internal nodes have suffixes from both strings S and R, i.e. all of them represent a common substring on the path from root to themselves. Now we need to find deepest node satisfying position constraint. For this, we need to check if there is a forward index S_i on a node, then there must be a reverse index R_i with value $(N-2) - (S_i + L - 1)$ where N is length of string $S\#$ and L is node depth (or substring length). If yes, then consider this node as a LPS candidate, else ignore it. In above figure, deepest node is highlighted which represents LPS as bbaabbb.

We have not shown forward and reverse indices on root node in figure. Because root node itself doesn't represent any common substring (In code implementation also, forward and reverse indices will not be calculated on root node)

How to implement this approach to find LPS? Here are the things that we need:

- We need to know forward and reverse indices on each node.
- For a given forward index S_i on an internal node, we need know if reverse index $R_i = (N-2) - (S_i + L - 1)$ also present on same node.
- Keep track of deepest internal node satisfying above condition.

One way to do above is:

While DFS on suffix tree, we can store forward and reverse indices on each node in some way (storage will help to avoid repeated traversals on tree when we need to know forward and reverse indices on a node). Later on, we can do another DFS to look for nodes satisfying position constraint. For position constraint check, we need to search in list of indices.

What data structure is suitable here to do all these in quickest way ?

- If we store indices in array, it will require linear search which will make overall approach non-linear in time.
- If we store indices in tree (set in C++, TreeSet in Java), we may use binary search but still overall approach will be non-linear in time.
- If we store indices in hash function based set (unordered_set in C++, HashSet in Java), it will provide a constant search on average and this will make overall approach linear in time. *A hash function based set may take more space depending on values being stored.*

We will use unordered_set (one for forward and other from reverse indices) in our implementation, added as a member variable in SuffixTreeNode structure.

```
// A C++ program to implement Ukkonen's Suffix Tree Construction
// Here we build generalized suffix tree for given string S
// and it's reverse R, then we find
// longest palindromic substring of given string S
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <iostream>
#include <unordered_set>
#define MAX_CHAR 256
using namespace std;

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];

    //pointer to other node via suffix link
    struct SuffixTreeNode *suffixLink;

    /*(start, end) interval specifies the edge, by which the
    node is connected to its parent node. Each edge will
    connect two nodes, one parent and one child, and
    (start, end) interval of a given edge will be stored
    in the child node. Lets say there are two nodes A and B
    connected by an edge with indices (5, 8) then this
    indices (5, 8) will be stored in node B. */
    int start;
    int *end;

    /*for leaf nodes, it stores the index of suffix for
    the path from root to leaf*/
    int suffixIndex;
};
```

```

//To store indices of children suffixes in given string
unordered_set<int> *forwardIndices;

//To store indices of children suffixes in reversed string
unordered_set<int> *reverseIndices;
};

typedef struct SuffixTreeNode Node;

char text[100]; //Input string
Node *root = NULL; //Pointer to root node

/*lastNewNode will point to newly created internal node,
waiting for it's suffix link to be set, which might get
a new suffix link (other than root) in next extension of
same phase. lastNewNode will be set to NULL when last
newly created internal node (if there is any) got it's
suffix link reset to new internal node created in next
extension of same phase. */
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represented as input string character
index (not the character itself)*/
int activeEdge = -1;
int activeLength = 0;

//remainingSuffixCount tells how many suffixes yet to
//be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string
int size1 = 0; //Size of 1st string
int reverseIndex; //Index of a suffix in reversed string
unordered_set<int>::iterator forwardIndex;

Node *newNode(int start, int *end)
{
    Node *node =(Node*) malloc(sizeof(Node));
    int i;
    for (i = 0; i < MAX_CHAR; i++)
        node->children[i] = NULL;

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

    /*suffixIndex will be set to -1 by default and
    actual suffix index will be set later for leaves
    at the end of all phases*/
    node->suffixIndex = -1;
    node->forwardIndices = new unordered_set<int>;
    node->reverseIndices = new unordered_set<int>;
    return node;
}

int edgeLength(Node *n) {
    if(n == root)
        return 0;
    return *(n->end) - (n->start) + 1;
}

int walkDown(Node *currNode)
{
    /*activePoint change for walk down (APCFWD) using
    Skip/Count Trick (Trick 1). If activeLength is greater
    than current edge length, set next internal node as
    activeNode and adjust activeEdge and activeLength
    accordingly to represent same activePoint*/
    if (activeLength >= edgeLength(currNode))
    {
        activeEdge += edgeLength(currNode);
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
        return 1;
    }
    return 0;
}

void extendSuffixTree(int pos)
{
    /*Extension Rule 1, this takes care of extending all
    leaves created so far in tree*/
    leafEnd = pos;

    /*Increment remainingSuffixCount indicating that a
    new suffix added to the list of suffixes yet to be
    added in tree*/
    remainingSuffixCount++;

    /*set lastNewNode to NULL while starting a new phase,
    indicating there is no internal node waiting for
    it's suffix link reset in current phase*/
    lastNewNode = NULL;

    //Add all suffixes (yet to be added) one by one in tree
    while(remainingSuffixCount > 0) {

        if (activeLength == 0)
            activeEdge = pos; //APCFALZ

        // There is no outgoing edge starting with
        // activeEdge from activeNode
        if (activeNode->children[text[activeEdge]] == NULL)
        {
            //Extension Rule 2 (A new leaf edge gets created)
            activeNode->children[text[activeEdge]] =
                newNode(pos, &leafEnd);

```



```

/*A new leaf edge is created in above line starting
from an existing node (the current activeNode), and
if there is any internal node waiting for it's suffix
link get reset, point the suffix link from that last
internal node to current activeNode. Then set lastNewNode
to NULL indicating no more node waiting for suffix link
reset*/
if (lastNewNode != NULL)
{
    lastNewNode->suffixLink = activeNode;
    lastNewNode = NULL;
}
}

// There is an outgoing edge starting with activeEdge
// from activeNode
else
{
    // Get the next node at the end of edge starting
    // with activeEdge
    Node *next = activeNode->children[text[activeEdge]];
    if (walkDown(next))//Do walkdown
    {
        //Start from next node (the new activeNode)
        continue;
    }
    /*Extension Rule 3 (current character being processed
    is already on the edge)*/
    if (text[next->start + activeLength] == text[pos])
    {
        //APCFER3
        activeLength++;
        /*STOP all further processing in this phase
        and move on to next phase*/
        break;
    }

    /*We will be here when activePoint is in middle of
    the edge being traversed and current character
    being processed is not on the edge (we fall off
    the tree). In this case, we add a new internal node
    and a new leaf edge going out of that new node. This
    is Extension Rule 2, where a new leaf edge and a new
    internal node get created*/
    splitEnd = (int*) malloc(sizeof(int));
    *splitEnd = next->start + activeLength - 1;

    //New internal node
    Node *split = newNode(next->start, splitEnd);
    activeNode->children[text[activeEdge]] = split;

    //New leaf coming out of new internal node
    split->children[text[pos]] = newNode(pos, &leafEnd);
    next->start += activeLength;
    split->children[text[next->start]] = next;

    /*We got a new internal node here. If there is any
    internal node created in last extensions of same
    phase which is still waiting for it's suffix link
    reset, do it now.*/
    if (lastNewNode != NULL)
    {
        /*suffixLink of lastNewNode points to current newly
        created internal node*/
        lastNewNode->suffixLink = split;
    }

    /*Make the current newly created internal node waiting
    for it's suffix link reset (which is pointing to root
    at present). If we come across any other internal node
    (existing or newly created) in next extension of same
    phase, when a new leaf edge gets added (i.e. when
    Extension Rule 2 applies is any of the next extension
    of same phase) at that point, suffixLink of this node
    will point to that internal node.*/
    lastNewNode = split;
}

}

/* One suffix got added in tree, decrement the count of
suffixes yet to be added.*/
remainingSuffixCount--;
if (activeNode == root && activeLength > 0) //APCFER2C1
{
    activeLength--;
    activeEdge = pos - remainingSuffixCount + 1;
}
else if (activeNode != root) //APCFER2C2
{
    activeNode = activeNode->suffixLink;
}
}
}

void print(int i, int j)
{
    int k;
    for (k=i; k<=j && text[k] != '#'; k++)
        printf("%c", text[k]);
    if (k<=j)
        printf("#");
}

//Print the suffix tree as well along with setting suffix index
//So tree will be printed in DFS manner
//Each edge along with it's suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL) return;

    if (n->start != -1) //A non-root node
    {
        //Print the label on edge from parent to current node
        //Uncomment below line to print suffix tree
        //printf(n->start, "(n->end)");
    }
    int leaf = 1;
    int i;

```

```

for (i = 0; i < MAX_CHAR; i++)
{
    if (n->children[i] != NULL)
    {
        //Uncomment below two lines to print suffix index
        // if (leaf == 1 && n->start != -1)
        //     printf("%d\n", n->suffixIndex);

        //Current node is not a leaf as it has outgoing
        //edges from it.
        leaf = 0;
        setSuffixIndexByDFS(n->children[i], labelHeight +
            edgeLength(n->children[i]));
    }

    if (n != root)
    {
        //Add children's suffix indices in parent
        n->forwardIndices->insert(
            n->children[i]->forwardIndices->begin(),
            n->children[i]->forwardIndices->end());
        n->reverseIndices->insert(
            n->children[i]->reverseIndices->begin(),
            n->children[i]->reverseIndices->end());
    }
}

if (leaf == 1)
{
    for (i = n->start; i <= (n->end); i++)
    {
        if (text[i] == '#')
        {
            n->end = (int*) malloc(sizeof(int));
            *(n->end) = i;
        }
    }

    n->suffixIndex = size - labelHeight;

    if (n->suffixIndex < size1) //Suffix of Given String
        n->forwardIndices->insert(n->suffixIndex);
    else //Suffix of Reversed String
        n->reverseIndices->insert(n->suffixIndex - size1);

    //Uncomment below line to print suffix index
    // printf("%d\n", n->suffixIndex);
}
}

```

```

void freeSuffixTreeByPostOrder(Node *n)
{
    if (n == NULL)
        return;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
    if (n->suffixIndex == -1)
        free(n->end);
    free(n);
}

```

/*Build the suffix tree and print the edge labels along with suffixIndex. suffixIndex for leaf edges will be >= 0 and for non-leaf edges will be -1*/

```

void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = -1;

    /*Root is a special node with start and end indices as -1,
    as it has no parent from where an edge comes to root*/
    root = newNode(-1, rootEnd);

    activeNode = root; //First activeNode will be root
    for (i=0; i<size; i++)
        extendSuffixTree(i);
    int labelHeight = 0;
    setSuffixIndexByDFS(root, labelHeight);
}

```

```

void doTraversal(Node *n, int labelHeight, int* maxHeight,
int* substringStartIndex)
{
    if (n == NULL)
    {
        return;
    }
    int i=0;
    int ret = -1;
    if (n->suffixIndex < 0) //If it is internal node
    {
        for (i = 0; i < MAX_CHAR; i++)
        {
            if (n->children[i] != NULL)
            {
                doTraversal(n->children[i], labelHeight +
                    edgeLength(n->children[i]),
                    maxHeight, substringStartIndex);
            }
        }

        if (*maxHeight < labelHeight
            && n->forwardIndices->size() > 0 &&
            n->reverseIndices->size() > 0)
        {
            for (forwardIndex=n->forwardIndices->begin();
                forwardIndex!=n->forwardIndices->end();
                ++forwardIndex)
            {
                reverseIndex = (size1 - 2) -
                    (*forwardIndex + labelHeight - 1);
                //If reverse suffix comes from
                //SAME position in given string
            }
        }
    }
}

```

```

//Keep track of deepest node
if(n->reverseIndices->find(reverseIndex) !=
n->reverseIndices->end())
{
    *maxHeight = labelHeight;
    *substringStartIndex = *(n->end) -
    labelHeight + 1;
    break;
}
}
}
}
}
}
}

void getLongestPalindromicSubstring()
{
    int maxHeight = 0;
    int substringStartIndex = 0;
    doTraversal(root, 0, &maxHeight, &substringStartIndex);

    int k;
    for (k=0; k<maxHeight; k++)
        printf("%c", text[k + substringStartIndex]);
    if(k == 0)
        printf("No palindromic substring");
    else
        printf(", of length: %d",maxHeight);
    printf("\n");
}

// driver program to test above functions
int main(int argc, char *argv[])
{
    size1 = 9;
    printf("Longest Palindromic Substring in cabbaabb is: ");
    strcpy(text, "cabbaabb#bbaabbac$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 17;
    printf("Longest Palindromic Substring in forgeeksskeegfor is: ");
    strcpy(text, "forgeeksskeegfor#rforgeeksskeegro$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 6;
    printf("Longest Palindromic Substring in abcde is: ");
    strcpy(text, "abcde#edcba$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 7;
    printf("Longest Palindromic Substring in abcdae is: ");
    strcpy(text, "abcdae#eadcba$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 6;
    printf("Longest Palindromic Substring in abacd is: ");
    strcpy(text, "abacd#dcaba$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 6;
    printf("Longest Palindromic Substring in abdc is: ");
    strcpy(text, "abdc#cdcba$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 13;
    printf("Longest Palindromic Substring in abacdfgdcaba is: ");
    strcpy(text, "abacdfgdcaba#abacdfgdcaba$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 15;
    printf("Longest Palindromic Substring in xyabacdfgdcaba is: ");
    strcpy(text, "xyabacdfgdcaba#xyabacdfgdcabayx$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 9;
    printf("Longest Palindromic Substring in xababayz is: ");
    strcpy(text, "xababayz#xyababax$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 6;
    printf("Longest Palindromic Substring in xabax is: ");
    strcpy(text, "xabax#xabax$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    return 0;
}

```

Output:

```

Longest Palindromic Substring in cabbaabb is: bbaabb, of length: 6
Longest Palindromic Substring in forgeeksskeegfor is: geeksskeeg, of length: 10
Longest Palindromic Substring in abcde is: a, of length: 1
Longest Palindromic Substring in abcdae is: a, of length: 1
Longest Palindromic Substring in abacd is: aba, of length: 3

```

Longest Palindromic Substring in abcdc is: cdc, of length: 3
Longest Palindromic Substring in abacdfigdcaba is: aba, of length: 3
Longest Palindromic Substring in xyabacdfigdcaba is: aba, of length: 3
Longest Palindromic Substring in xababayz is: ababa, of length: 5
Longest Palindromic Substring in xabax is: xabax, of length: 5

Followup:

Detect ALL palindromes in a given string.

e.g. For string abcdcbefgl, all possible palindromes are a, b, c, d, e, f, g, dd, fgf, cddc, bddcb.

We have published following more articles on suffix tree applications:

- [Suffix Tree Application 1 – Substring Check](#)
- [Suffix Tree Application 2 – Searching All Patterns](#)
- [Suffix Tree Application 3 – Longest Repeated Substring](#)
- [Suffix Tree Application 4 – Build Linear Time Suffix Array](#)
- [Suffix Tree Application 5 – Longest Common Substring](#)
- [Generalized Suffix Tree 1](#)

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Pattern Searching
palindrome
Suffix Tree

AVL Tree | Set 1 (Insertion)

AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

An Example Tree that is an AVL Tree



The above tree is AVL because differences between heights of left and right subtrees for every node is less than or equal to 1.

An Example Tree that is NOT an AVL Tree



The above tree is not AVL because differences between heights of left and right subtrees for 8 and 18 is greater than 1.

Images are taken from [here](#).

Why AVL Trees?

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take $O(h)$ time where h is the height of the BST. The cost of these operations may become $O(n)$ for a skewed Binary tree. If we make sure that height of the tree remains $O(\log n)$ after every insertion and deletion, then we can guarantee an upper bound of $O(\log n)$ for all these operations. The height of an AVL tree is always $O(\log n)$ where n is the number of nodes in the tree (See [this](#) video lecture for proof).

Insertion

To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property (keys(left) T_1 , T_2 and T_3 are subtrees of the tree rooted with y (on left side) or x (on right side) $y \times \setminus \setminus$ Right Rotation $/ \setminus \times T_3 \text{-----} \rightarrow T_1 y \setminus$

Steps to follow for insertion

Let the newly inserted node be w

1) Perform standard BST insert for w .

2) Starting from w , travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z that comes on the path from w to z .

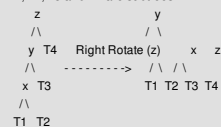
3) Re-balance the tree by performing appropriate rotations on the subtree rooted with z . There can be 4 possible cases that needs to be handled as x , y and z can be arranged in 4 ways. Following are the possible 4 arrangements:

- y is left child of z and x is left child of y (Left Left Case)
- y is left child of z and x is right child of y (Left Right Case)
- y is right child of z and x is right child of y (Right Right Case)
- y is right child of z and x is left child of y (Right Left Case)

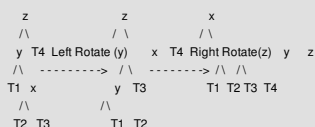
Following are the operations to be performed in above mentioned 4 cases. In all of the cases, we only need to re-balance the subtree rooted with z and the complete tree becomes balanced as the height of subtree (After appropriate rotations) rooted with z becomes same as it was before insertion. (See [this](#) video lecture for proof)

a) Left Left Case

T_1 , T_2 , T_3 and T_4 are subtrees.



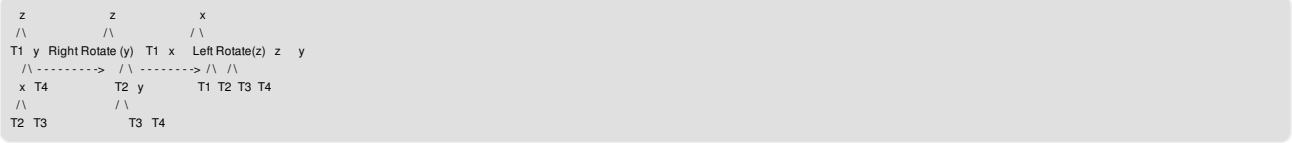
b) Left Right Case



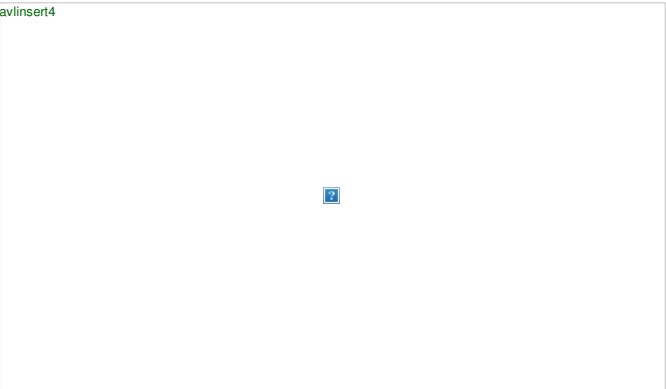
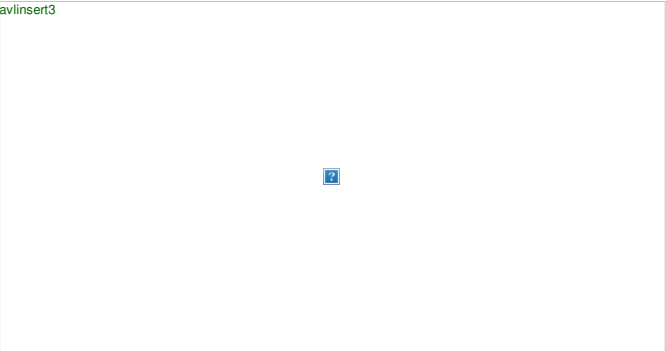
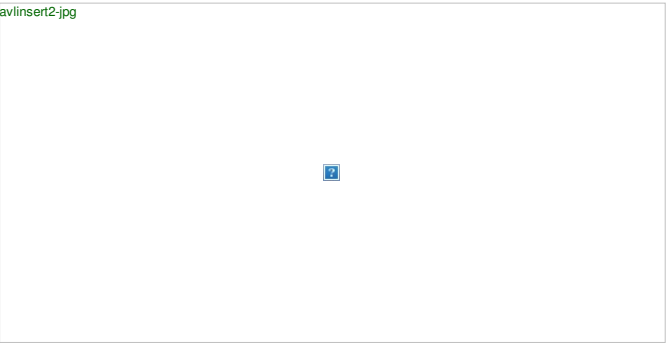
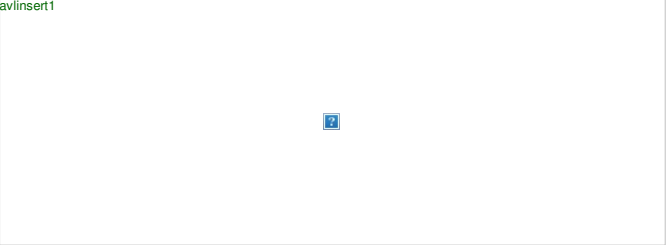
c) Right Right Case



d) Right Left Case



Insertion Examples:



avlinsert5



The above images are taken from [here](#).

implementation

Following is the implementation for AVL Tree Insertion. The following implementation uses the recursive BST insert to insert a new node. In the recursive BST insert, after insertion, we get pointers to all ancestors one by one in bottom up manner. So we don't need parent pointer to travel up. The recursive code itself travels up and visits all the ancestors of the newly inserted node.

- 1) Perform the normal BST insertion.
- 2) The current node must be one of the ancestors of the newly inserted node. Update the height of the current node.
- 3) Get the balance factor (left subtree height – right subtree height) of the current node.
- 4) If balance factor is greater than 1, then the current node is unbalanced and we are either in Left Left case or left Right case. To check whether it is left left case or not, compare the newly inserted key with the key in left subtree root.
- 5) If balance factor is less than -1, then the current node is unbalanced and we are either in Right Right case or Right Left case. To check whether it is Right Right case or not, compare the newly inserted key with the key in right subtree root.

C

```
// C program to insert a node in AVL tree
#include<stdio.h>
#include<stdlib.h>

// An AVL tree node
struct Node
{
    int key;
    struct Node *left;
    struct Node *right;
    int height;
};

// A utility function to get maximum of two integers
int max(int a, int b);

// A utility function to get height of the tree
int height(struct Node *N)
{
    if (N == NULL)
        return 0;
    return N->height;
}

// A utility function to get maximum of two integers
int max(int a, int b)
{
    return (a > b)? a : b;
}

/* Helper function that allocates a new node with the given key and
   NULL left and right pointers. */
struct Node* newNode(int key)
{
    struct Node* node = (struct Node*)
        malloc(sizeof(struct Node));
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1; // new node is initially added at leaf
    return(node);
}

// A utility function to right rotate subtree rooted with y
// See the diagram given above.
struct Node *rightRotate(struct Node *y)
{
    struct Node *x = y->left;
    struct Node *T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left), height(y->right))+1;
    x->height = max(height(x->left), height(x->right))+1;

    // Return new root
    return x;
}

// A utility function to left rotate subtree rooted with x
// See the diagram given above.
struct Node *leftRotate(struct Node *x)
{
    struct Node *y = x->right;
    struct Node *T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;
```

```

// Update heights
x->height = max(height(x->left), height(x->right))+1;
y->height = max(height(y->left), height(y->right))+1;

// Return new root
return y;
}

// Get Balance factor of node N
int getBalance(struct Node *N)
{
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}

// Recursive function to insert key in subtree rooted
// with node and returns new root of subtree.
struct Node* insert(struct Node* node, int key)
{
    /* 1. Perform the normal BST insertion */
    if (node == NULL)
        return (newNode(key));

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else // Equal keys are not allowed in BST
        return node;

    /* 2. Update height of this ancestor node */
    node->height = 1 + max(height(node->left),
        height(node->right));

    /* 3. Get the balance factor of this ancestor
    node to check whether this node became
    unbalanced */
    int balance = getBalance(node);

    // If this node becomes unbalanced, then
    // there are 4 cases

    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node->left->key)
    {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->key)
    {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    /* return the (unchanged) node pointer */
    return node;
}

// A utility function to print preorder traversal
// of the tree.
// The function also prints height of every node
void preOrder(struct Node *root)
{
    if(root != NULL)
    {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}

/* Driver program to test above function */
int main()
{
    struct Node *root = NULL;

    /* Constructing tree given in the above figure */
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);

    /* The constructed AVL Tree would be
        30
       / \
      20  40
     / \  \
    10 25 50
    */

    printf("Preorder traversal of the constructed AVL"
        " tree is \n");
    preOrder(root);

    return 0;
}

```

Java

```

// Java program for insertion in AVL Tree

```

```

class Node {
    int key, height;
    Node left, right;

    Node(int d) {
        key = d;
        height = 1;
    }
}

class AVLTree {

    Node root;

    // A utility function to get height of the tree
    int height(Node N) {
        if (N == null)
            return 0;

        return N.height;
    }

    // A utility function to get maximum of two integers
    int max(int a, int b) {
        return (a > b) ? a : b;
    }

    // A utility function to right rotate subtree rooted with y
    // See the diagram given above.
    Node rightRotate(Node y) {
        Node x = y.left;
        Node T2 = x.right;

        // Perform rotation
        x.right = y;
        y.left = T2;

        // Update heights
        y.height = max(height(y.left), height(y.right)) + 1;
        x.height = max(height(x.left), height(x.right)) + 1;

        // Return new root
        return x;
    }

    // A utility function to left rotate subtree rooted with x
    // See the diagram given above.
    Node leftRotate(Node x) {
        Node y = x.right;
        Node T2 = y.left;

        // Perform rotation
        y.left = x;
        x.right = T2;

        // Update heights
        x.height = max(height(x.left), height(x.right)) + 1;
        y.height = max(height(y.left), height(y.right)) + 1;

        // Return new root
        return y;
    }

    // Get Balance factor of node N
    int getBalance(Node N) {
        if (N == null)
            return 0;

        return height(N.left) - height(N.right);
    }

    Node insert(Node node, int key) {

        /* 1. Perform the normal BST insertion */
        if (node == null)
            return (new Node(key));

        if (key < node.key)
            node.left = insert(node.left, key);
        else if (key > node.key)
            node.right = insert(node.right, key);
        else // Duplicate keys not allowed
            return node;

        /* 2. Update height of this ancestor node */
        node.height = 1 + max(height(node.left),
                               height(node.right));

        /* 3. Get the balance factor of this ancestor
        node to check whether this node became
        unbalanced */
        int balance = getBalance(node);

        // If this node becomes unbalanced, then there
        // are 4 cases Left Left Case
        if (balance > 1 && key < node.left.key)
            return rightRotate(node);

        // Right Right Case
        if (balance < -1 && key > node.right.key)
            return leftRotate(node);

        // Left Right Case
        if (balance > 1 && key > node.left.key) {
            node.left = leftRotate(node.left);
            return rightRotate(node);
        }

        // Right Left Case
        if (balance < -1 && key < node.right.key) {
            node.right = rightRotate(node.right);
            return leftRotate(node);
        }

        /* return the (unchanged) node pointer */
        return node;
    }
}

```



```

}

// A utility function to print preorder traversal
// of the tree.
// The function also prints height of every node
void preOrder(Node node) {
    if (node != null) {
        System.out.print(node.key + " ");
        preOrder(node.left);
        preOrder(node.right);
    }
}

public static void main(String[] args) {
    AVLTree tree = new AVLTree();

    /* Constructing tree given in the above figure */
    tree.root = tree.insert(tree.root, 10);
    tree.root = tree.insert(tree.root, 20);
    tree.root = tree.insert(tree.root, 30);
    tree.root = tree.insert(tree.root, 40);
    tree.root = tree.insert(tree.root, 50);
    tree.root = tree.insert(tree.root, 25);

    /* The constructed AVL Tree would be
        30
       / \
      20 40
     / \  \
    10 25 50
    */
    System.out.println("Preorder traversal" +
        " of constructed tree is :");
    tree.preOrder(tree.root);
}
// This code has been contributed by Mayank Jaiswal

```

Output:

```

Preorder traversal of the constructed AVL tree is
30 20 10 25 40 50

```

Time Complexity: The rotation operations (left and right rotate) take constant time as only few pointers are being changed there. Updating the height and getting the balance factor also take constant time. So the time complexity of AVL insert remains same as BST insert which is $O(h)$ where h is height of the tree. Since AVL tree is balanced, the height is $O(\log n)$. So time complexity of AVL insert is $O(\log n)$.

Comparison with Red Black Tree

The AVL tree and other self balancing search trees like Red Black are useful to get all basic operations done in $O(\log n)$ time. The AVL trees are more balanced compared to Red Black Trees, but they may cause more rotations during insertion and deletion. So if your application involves many frequent insertions and deletions, then Red Black trees should be preferred. And if the insertions and deletions are less frequent and search is more frequent operation, then AVL tree should be preferred over [Red Black Tree](#).

Following is the post for delete.

[AVL Tree | Set 2 \(Deletion\)](#)

Following are some posts that have used self-balancing search trees.

[Median in a stream of integers \(running integers\)](#)

[Maximum of all subarrays of size k](#)

[Count smaller elements on right side](#)

References:

[IITD Video Lecture on AVL Tree Introduction](#)

[IITD Video Lecture on AVL Tree Insertion and Deletion](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Advance Data Structures
AVL-Tree
Self-Balancing-BST

AVL Tree | Set 2 (Deletion)

We have discussed AVL insertion in the [previous post](#). In this post, we will follow a similar approach for deletion.

Steps to follow for deletion.

To make sure that the given tree remains AVL after every deletion, we must augment the standard BST delete operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property (keys(left) T_1 , T_2 and T_3 are subtrees of the tree rooted with y (on left side) or x (on right side) $y \times / \backslash$ Right Rotation $/ \backslash \times T_3 \text{-----} \rightarrow T_1 y / \backslash$

Let w be the node to be deleted

1) Perform standard BST delete for w .

2) Starting from w , travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the larger height child of z , and x be the larger height child of y . Note that the definitions of x and y are different from [insertion](#) here.

3) Re-balance the tree by performing appropriate rotations on the subtree rooted with z . There can be 4 possible cases that needs to be handled as x , y and z can be arranged in 4 ways. Following are the possible 4 arrangements:

- y is left child of z and x is left child of y (Left Left Case)
- y is left child of z and x is right child of y (Left Right Case)
- y is right child of z and x is right child of y (Right Right Case)
- y is right child of z and x is left child of y (Right Left Case)

Like insertion, following are the operations to be performed in above mentioned 4 cases. Note that, unlike insertion, fixing the node z won't fix the complete AVL tree. After fixing z , we may have to fix ancestors of z as well (See [this video lecture](#) for proof)

a) Left Left Case

T_1 , T_2 , T_3 and T_4 are subtrees.

```

      z           y
     /\         /\
    y T4  Right Rotate (z)  x  z
   /\  ----->  /\ /\
  x T3           T1 T2 T3 T4
 /\
T1 T2

```

b) Left Right Case

```

      z           z           x
     /\         /\         /\

```

```

y  T4 Left Rotate (y)    x  T4 Right Rotate(z)  y  z
/\ -----> /\ -----> /\ /\
T1 x          y  T3      T1 T2 T3 T4
/\           /\
T2 T3       T1 T2

```

c) Right Right Case

```

z          y
/\         /\
T1 y  Left Rotate(z)  z  x
/\ -----> /\ /\
T2 x          T1 T2 T3 T4
/\
T3 T4

```

d) Right Left Case

```

z          z          x
/\         /\         /\
T1 y  Right Rotate (y)  T1 x  Left Rotate(z)  z  x
/\ -----> /\ -----> /\ /\
x  T4          T2 y          T1 T2 T3 T4
/\           /\
T2 T3       T3 T4

```

Unlike insertion, in deletion, after we perform a rotation at z, we may have to perform a rotation at ancestors of z. Thus, we must continue to trace the path until we reach the root.

Example:

avl-delete1



A node with value 32 is being deleted. After deleting 32, we travel up and find the first unbalanced node which is 44. We mark it as z, its higher height child as y which is 52, and y's higher height child as x which could be either 78 or 50 as both are of same height. We have considered 78. Now the case is Right Left, so we perform left rotation.

Image source: <https://www.cs.purdue.edu/homes/ayg/CS251/slides/chap7b.pdf>

C implementation

Following is the C implementation for AVL Tree Deletion. The following C implementation uses the recursive BST delete as basis. In the recursive BST delete, after deletion, we get pointers to all ancestors one by one in bottom up manner. So we don't need parent pointer to travel up. The recursive code itself travels up and visits all the ancestors of the deleted node.

- 1) Perform the normal BST deletion.
- 2) The current node must be one of the ancestors of the deleted node. Update the height of the current node.
- 3) Get the balance factor (left subtree height – right subtree height) of the current node.
- 4) If balance factor is greater than 1, then the current node is unbalanced and we are either in Left Left case or Left Right case. To check whether it is Left Left case or Left Right case, get the balance factor of left subtree. If balance factor of the left subtree is greater than or equal to 0, then it is Left Left case, else Left Right case.
- 5) If balance factor is less than -1, then the current node is unbalanced and we are either in Right Right case or Right Left case. To check whether it is Right Right case or Right Left case, get the balance factor of right subtree. If the balance factor of the right subtree is smaller than or equal to 0, then it is Right Right case, else Right Left case.

C

```

// C program to delete a node from AVL Tree
#include<stdio.h>
#include<stdlib.h>

// An AVL tree node
struct Node
{
    int key;
    struct Node *left;
    struct Node *right;
    int height;
};

// A utility function to get maximum of two integers
int max(int a, int b);

// A utility function to get height of the tree
int height(struct Node *N)
{
    if (N == NULL)
        return 0;
    return N->height;
}

// A utility function to get maximum of two integers
int max(int a, int b)
{
    return (a > b)? a : b;
}

```

```

/* Helper function that allocates a new node with the given key and
   NULL left and right pointers. */
struct Node* newNode(int key)
{
    struct Node* node = (struct Node*)
        malloc(sizeof(struct Node));
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1; // new node is initially added at leaf
    return(node);
}

// A utility function to right rotate subtree rooted with y
// See the diagram given above.
struct Node *rightRotate(struct Node *y)
{
    struct Node *x = y->left;
    struct Node *T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left), height(y->right))+1;
    x->height = max(height(x->left), height(x->right))+1;

    // Return new root
    return x;
}

// A utility function to left rotate subtree rooted with x
// See the diagram given above.
struct Node *leftRotate(struct Node *x)
{
    struct Node *y = x->right;
    struct Node *T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = max(height(x->left), height(x->right))+1;
    y->height = max(height(y->left), height(y->right))+1;

    // Return new root
    return y;
}

// Get Balance factor of node N
int getBalance(struct Node *N)
{
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}

struct Node* insert(struct Node* node, int key)
{
    /* 1. Perform the normal BST rotation */
    if (node == NULL)
        return(newNode(key));

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else // Equal keys not allowed
        return node;

    /* 2. Update height of this ancestor node */
    node->height = 1 + max(height(node->left),
        height(node->right));

    /* 3. Get the balance factor of this ancestor
       node to check whether this node became
       unbalanced */
    int balance = getBalance(node);

    // If this node becomes unbalanced, then there are 4 cases

    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node->left->key)
    {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->key)
    {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    /* return the (unchanged) node pointer */
    return node;
}

/* Given a non-empty binary search tree, return the
   node with minimum key value found in that tree.
   Note that the entire tree does not need to be
   searched. */
struct Node *minValueNode(struct Node* node)
{
    struct Node* current = node;

```

```

/* loop down to find the leftmost leaf */
while (current->left != NULL)
    current = current->left;

return current;
}

// Recursive function to delete a node with given key
// from subtree with given root. It returns root of
// the modified subtree.
struct Node* deleteNode(struct Node* root, int key)
{
    // STEP 1: PERFORM STANDARD BST DELETE

    if (root == NULL)
        return root;

    // If the key to be deleted is smaller than the
    // root's key, then it lies in left subtree
    if (key < root->key)
        root->left = deleteNode(root->left, key);

    // If the key to be deleted is greater than the
    // root's key, then it lies in right subtree
    else if (key > root->key)
        root->right = deleteNode(root->right, key);

    // if key is same as root's key, then This is
    // the node to be deleted
    else
    {
        // node with only one child or no child
        if ((root->left == NULL) || (root->right == NULL))
        {
            struct Node *temp = root->left ? root->left :
                root->right;

            // No child case
            if (temp == NULL)
            {
                temp = root;
                root = NULL;
            }
            else // One child case
                *root = *temp; // Copy the contents of
                // the non-empty child
            free(temp);
        }
        else
        {
            // node with two children: Get the inorder
            // successor (smallest in the right subtree)
            struct Node* temp = minValueNode(root->right);

            // Copy the inorder successor's data to this node
            root->key = temp->key;

            // Delete the inorder successor
            root->right = deleteNode(root->right, temp->key);
        }
    }

    // If the tree had only one node then return
    if (root == NULL)
        return root;

    // STEP 2: UPDATE HEIGHT OF THE CURRENT NODE
    root->height = 1 + max(height(root->left),
        height(root->right));

    // STEP 3: GET THE BALANCE FACTOR OF THIS NODE (to
    // check whether this node became unbalanced)
    int balance = getBalance(root);

    // If this node becomes unbalanced, then there are 4 cases

    // Left Left Case
    if (balance > 1 && getBalance(root->left) >= 0)
        return rightRotate(root);

    // Left Right Case
    if (balance > 1 && getBalance(root->left) < 0)
    {
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }

    // Right Right Case
    if (balance < -1 && getBalance(root->right) <= 0)
        return leftRotate(root);

    // Right Left Case
    if (balance < -1 && getBalance(root->right) > 0)
    {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }

    return root;
}

// A utility function to print preorder traversal of
// the tree.
// The function also prints height of every node
void preOrder(struct Node *root)
{
    if(root != NULL)
    {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}

/* Driver program to test above function*/

```

```

int main()
{
    struct Node *root = NULL;

    /* Constructing tree given in the above figure */
    root = insert(root, 9);
    root = insert(root, 5);
    root = insert(root, 10);
    root = insert(root, 0);
    root = insert(root, 6);
    root = insert(root, 11);
    root = insert(root, -1);
    root = insert(root, 1);
    root = insert(root, 2);

    /* The constructed AVL Tree would be
        9
       / \
      1  10
     / \  \
    0  5  11
   / \ \
  -1 2  6
 */

    printf("Preorder traversal of the constructed AVL -\n");
    preOrder(root);

    root = deleteNode(root, 10);

    /* The AVL Tree after deletion of 10
        1
       / \
      0  9
     / \ \
    -1  5 11
   / \ \
    2  6
 */

    printf("\nPreorder traversal after deletion of 10\n");
    preOrder(root);

    return 0;
}

```

Java

```

// Java program for deletion in AVL Tree

class Node
{
    int key, height;
    Node left, right;

    Node(int d)
    {
        key = d;
        height = 1;
    }
}

class AVLTree
{
    Node root;

    // A utility function to get height of the tree
    int height(Node N)
    {
        if (N == null)
            return 0;
        return N.height;
    }

    // A utility function to get maximum of two integers
    int max(int a, int b)
    {
        return (a > b) ? a : b;
    }

    // A utility function to right rotate subtree rooted with y
    // See the diagram given above.
    Node rightRotate(Node y)
    {
        Node x = y.left;
        Node T2 = x.right;

        // Perform rotation
        x.right = y;
        y.left = T2;

        // Update heights
        y.height = max(height(y.left), height(y.right)) + 1;
        x.height = max(height(x.left), height(x.right)) + 1;

        // Return new root
        return x;
    }

    // A utility function to left rotate subtree rooted with x
    // See the diagram given above.
    Node leftRotate(Node x)
    {
        Node y = x.right;
        Node T2 = y.left;

        // Perform rotation
        y.left = x;
        x.right = T2;

        // Update heights
        x.height = max(height(x.left), height(x.right)) + 1;

```

```
y.height = max(height(y.left), height(y.right)) + 1;
```

```
// Return new root  
return y;  
}
```

```
// Get Balance factor of node N  
int getBalance(Node N)
```

```
{  
    if (N == null)  
        return 0;  
    return height(N.left) - height(N.right);  
}
```

```
Node insert(Node node, int key)
```

```
{  
    /* 1. Perform the normal BST rotation */  
    if (node == null)  
        return (new Node(key));  
  
    if (key < node.key)  
        node.left = insert(node.left, key);  
    else if (key > node.key)  
        node.right = insert(node.right, key);  
    else // Equal keys not allowed  
        return node;
```

```
    /* 2. Update height of this ancestor node */  
    node.height = 1 + max(height(node.left),  
        height(node.right));
```

```
    /* 3. Get the balance factor of this ancestor  
    node to check whether this node became  
    Unbalanced */  
    int balance = getBalance(node);
```

```
    // If this node becomes unbalanced, then  
    // there are 4 cases Left Left Case  
    if (balance > 1 && key < node.left.key)  
        return rightRotate(node);
```

```
    // Right Right Case  
    if (balance < -1 && key > node.right.key)  
        return leftRotate(node);
```

```
    // Left Right Case  
    if (balance > 1 && key > node.left.key)  
    {  
        node.left = leftRotate(node.left);  
        return rightRotate(node);  
    }
```

```
    // Right Left Case  
    if (balance < -1 && key < node.right.key)  
    {  
        node.right = rightRotate(node.right);  
        return leftRotate(node);  
    }
```

```
    /* return the (unchanged) node pointer */  
    return node;  
}
```

```
/* Given a non-empty binary search tree, return the  
node with minimum key value found in that tree.  
Note that the entire tree does not need to be  
searched. */
```

```
Node minValueNode(Node node)
```

```
{  
    Node current = node;  
  
    /* loop down to find the leftmost leaf */  
    while (current.left != null)  
        current = current.left;  
  
    return current;  
}
```

```
Node deleteNode(Node root, int key)
```

```
{  
    // STEP 1: PERFORM STANDARD BST DELETE  
    if (root == null)  
        return root;
```

```
    // If the key to be deleted is smaller than  
    // the root's key, then it lies in left subtree  
    if (key < root.key)  
        root.left = deleteNode(root.left, key);
```

```
    // If the key to be deleted is greater than the  
    // root's key, then it lies in right subtree  
    else if (key > root.key)  
        root.right = deleteNode(root.right, key);
```

```
    // If key is same as root's key, then this is the node  
    // to be deleted  
    else  
    {
```

```
        // node with only one child or no child  
        if ((root.left == null) || (root.right == null))  
        {  
            Node temp = null;  
            if (temp == root.left)  
                temp = root.right;  
            else  
                temp = root.left;
```

```
        // No child case  
        if (temp == null)  
        {  
            temp = root;  
            root = null;  
        }  
        else // One child case  
            root = temp; // Copy the contents of
```

```

        // the non-empty child
    }
    else
    {

        // node with two children: Get the inorder
        // successor (smallest in the right subtree)
        Node temp = minValueNode(root.right);

        // Copy the inorder successor's data to this node
        root.key = temp.key;

        // Delete the inorder successor
        root.right = deleteNode(root.right, temp.key);
    }
}

// If the tree had only one node then return
if (root == null)
    return root;

// STEP 2: UPDATE HEIGHT OF THE CURRENT NODE
root.height = max(height(root.left), height(root.right)) + 1;

// STEP 3: GET THE BALANCE FACTOR OF THIS NODE (to check whether
// this node became unbalanced)
int balance = getBalance(root);

// If this node becomes unbalanced, then there are 4 cases
// Left Left Case
if (balance > 1 && getBalance(root.left) >= 0)
    return rightRotate(root);

// Left Right Case
if (balance > 1 && getBalance(root.left) < 0)
{
    root.left = leftRotate(root.left);
    return rightRotate(root);
}

// Right Right Case
if (balance < -1 && getBalance(root.right) <= 0)
    return leftRotate(root);

// Right Left Case
if (balance < -1 && getBalance(root.right) > 0)
{
    root.right = rightRotate(root.right);
    return leftRotate(root);
}

return root;
}

// A utility function to print preorder traversal of
// the tree. The function also prints height of every
// node
void preOrder(Node node)
{
    if (node != null)
    {
        System.out.print(node.key + " ");
        preOrder(node.left);
        preOrder(node.right);
    }
}

public static void main(String[] args)
{
    AVLTree tree = new AVLTree();

    /* Constructing tree given in the above figure */
    tree.root = tree.insert(tree.root, 9);
    tree.root = tree.insert(tree.root, 5);
    tree.root = tree.insert(tree.root, 10);
    tree.root = tree.insert(tree.root, 0);
    tree.root = tree.insert(tree.root, 6);
    tree.root = tree.insert(tree.root, 11);
    tree.root = tree.insert(tree.root, -1);
    tree.root = tree.insert(tree.root, 1);
    tree.root = tree.insert(tree.root, 2);

    /* The constructed AVL Tree would be
    9
    / \
    1  10
    / \ \
    0  5 11
    / \ \
    -1 2 6
    */
    System.out.println("Preorder traversal of "+
        "constructed tree is :");
    tree.preOrder(tree.root);

    tree.root = tree.deleteNode(tree.root, 10);

    /* The AVL Tree after deletion of 10
    1
    / \
    0  9
    / \
    -1 5 11
    / \
    2  6
    */
    System.out.println("");
    System.out.println("Preorder traversal after "+
        "deletion of 10 :");
    tree.preOrder(tree.root);
}
}

```

// This code has been contributed by Mayank Jaiswal

Output:

```
Preorder traversal of the constructed AVL tree is
9 1 0 -1 5 2 6 10 11
Preorder traversal after deletion of 10
1 0 -1 9 5 2 6 11
```

Time Complexity: The rotation operations (left and right rotate) take constant time as only few pointers are being changed there. Updating the height and getting the balance factor also take constant time. So the time complexity of AVL delete remains same as BST delete which is $O(h)$ where h is height of the tree. Since AVL tree is balanced, the height is $O(\log n)$. So time complexity of AVL delete is $O(\log n)$.

References:

<https://www.cs.purdue.edu/homes/ayg/CS251/slides/chap7b.pdf>

IITD Video Lecture on AVL Tree Insertion and Deletion

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Advance Data Structures
AVL-Tree
Self-Balancing-BST

AVL with duplicate keys

Please refer below post before reading about AVL tree handling of duplicates.

How to handle duplicates in Binary Search Tree?

The is to augment **AVL tree** node to store count together with regular fields like key, left and right pointers.

Insertion of keys 12, 10, 20, 9, 11, 10, 12, 12 in an empty Binary Search Tree would create following.

```
      12(3)
     /   \
    10(2) 20(1)
   /  \
  9(1) 11(1)
```

Count of a key is shown in bracket

Below is C implementation of normal AVL Tree with count with every key. This code basically is taken from [code for insert and delete in AVL tree](#). The changes made for handling duplicates are highlighted, rest of the code is same.

The important thing to note is changes are very similar to simple Binary Search Tree changes.

```
// AVL tree that handles duplicates
#include<stdio.h>
#include<stdlib.h>

// An AVL tree node
struct node
{
    int key;
    struct node *left;
    struct node *right;
    int height;
    int count;
};

// A utility function to get maximum of two integers
int max(int a, int b);

// A utility function to get height of the tree
int height(struct node *N)
{
    if (N == NULL)
        return 0;
    return N->height;
}

// A utility function to get maximum of two integers
int max(int a, int b)
{
    return (a > b)? a : b;
}

/* Helper function that allocates a new node with the given key and
   NULL left and right pointers. */
struct node* newNode(int key)
{
    struct node* node = (struct node*)
        malloc(sizeof(struct node));
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1; // new node is initially added at leaf
    node->count = 1;
    return(node);
}

// A utility function to right rotate subtree rooted with y
// See the diagram given above.
struct node *rightRotate(struct node *y)
{
    struct node *x = y->left;
    struct node *T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left), height(y->right))+1;
    x->height = max(height(x->left), height(x->right))+1;

    // Return new root
    return x;
}

// A utility function to left rotate subtree rooted with x
// See the diagram given above.
struct node *leftRotate(struct node *x)
{

```



```

struct node *y = x->right;
struct node *T2 = y->left;

// Perform rotation
y->left = x;
x->right = T2;

// Update heights
x->height = max(height(x->left), height(x->right))+1;
y->height = max(height(y->left), height(y->right))+1;

// Return new root
return y;
}

// Get Balance factor of node N
int getBalance(struct node *N)
{
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}

struct node* insert(struct node* node, int key)
{
    /* 1. Perform the normal BST rotation */
    if (node == NULL)
        return (newNode(key));

    // If key already exists in BST, increment count and return
    if (key == node->key)
    {
        (node->count)++;
        return node;
    }

    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left = insert(node->left, key);
    else
        node->right = insert(node->right, key);

    /* 2. Update height of this ancestor node */
    node->height = max(height(node->left), height(node->right)) + 1;

    /* 3. Get the balance factor of this ancestor node to check whether
    this node became unbalanced */
    int balance = getBalance(node);

    // If this node becomes unbalanced, then there are 4 cases

    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node->left->key)
    {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->key)
    {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    /* return the (unchanged) node pointer */
    return node;
}

/* Given a non-empty binary search tree, return the node with minimum
key value found in that tree. Note that the entire tree does not
need to be searched. */
struct node * minValueNode(struct node* node)
{
    struct node* current = node;

    /* loop down to find the leftmost leaf */
    while (current->left != NULL)
        current = current->left;

    return current;
}

struct node* deleteNode(struct node* root, int key)
{
    // STEP 1: PERFORM STANDARD BST DELETE

    if (root == NULL)
        return root;

    // If the key to be deleted is smaller than the root's key,
    // then it lies in left subtree
    if ( key < root->key )
        root->left = deleteNode(root->left, key);

    // If the key to be deleted is greater than the root's key,
    // then it lies in right subtree
    else if( key > root->key )
        root->right = deleteNode(root->right, key);

    // If key is same as root's key, then This is the node
    // to be deleted
    else
    {
        // If key is present more than once, simply decrement
        // count and return
        if (root->count > 1)
        {

```

```

        (root->count)--;
        return;
    }
    // ELSE, delete the node

    // node with only one child or no child
    if( (root->left == NULL) || (root->right == NULL) )
    {
        struct node *temp = root->left ? root->left : root->right;

        // No child case
        if(temp == NULL)
        {
            temp = root;
            root = NULL;
        }
        else // One child case
            *root = *temp; // Copy the contents of the non-empty child

        free(temp);
    }
    else
    {
        // node with two children: Get the inorder successor (smallest
        // in the right subtree)
        struct node* temp = minValueNode(root->right);

        // Copy the inorder successor's data to this node
        root->key = temp->key;

        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->key);
    }
}

// If the tree had only one node then return
if (root == NULL)
    return root;

// STEP 2: UPDATE HEIGHT OF THE CURRENT NODE
root->height = max(height(root->left), height(root->right)) + 1;

// STEP 3: GET THE BALANCE FACTOR OF THIS NODE (to check whether
// this node became unbalanced)
int balance = getBalance(root);

// If this node becomes unbalanced, then there are 4 cases

// Left Left Case
if (balance > 1 && getBalance(root->left) >= 0)
    return rightRotate(root);

// Left Right Case
if (balance > 1 && getBalance(root->left) < 0)
{
    root->left = leftRotate(root->left);
    return rightRotate(root);
}

// Right Right Case
if (balance < -1 && getBalance(root->right) <= 0)
    return leftRotate(root);

// Right Left Case
if (balance < -1 && getBalance(root->right) > 0)
{
    root->right = rightRotate(root->right);
    return leftRotate(root);
}

return root;
}

// A utility function to print preorder traversal of the tree.
// The function also prints height of every node
void preOrder(struct node *root)
{
    if(root != NULL)
    {
        printf("%d(%d) ", root->key, root->count);
        preOrder(root->left);
        preOrder(root->right);
    }
}

/* Driver program to test above function */
int main()
{
    struct node *root = NULL;

    /* Constructing tree given in the above figure */
    root = insert(root, 9);
    root = insert(root, 5);
    root = insert(root, 10);
    root = insert(root, 5);
    root = insert(root, 9);
    root = insert(root, 7);
    root = insert(root, 17);

    printf("Pre order traversal of the constructed AVL tree is \n");
    preOrder(root);

    root = deleteNode(root, 9);

    printf("\nPre order traversal after deletion of 10 \n");
    preOrder(root);

    return 0;
}

```

Output:

```

Pre order traversal of the constructed AVL tree is
9(2) 5(2) 7(1) 10(1) 17(1)
Pre order traversal after deletion of 10

```

Thanks to **Rounaq Jhunjhunu Wala** for sharing initial code. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: Tree

Splay Tree | Set 1 (Search)

The worst case time complexity of Binary Search Tree (BST) operations like search, delete, insert is $O(n)$. The worst case occurs when the tree is skewed. We can get the worst case time complexity as $O(\log n)$ with **AVL** and Red-Black Trees.

Can we do better than AVL or Red-Black trees in practical situations?

Like **AVL** and Red-Black Trees, Splay tree is also **self-balancing BST**. The main idea of splay tree is to bring the recently accessed item to root of the tree, this makes the recently searched item to be accessible in $O(1)$ time if accessed again. The idea is to use locality of reference (In a typical application, 80% of the access are to 20% of the items). Imagine a situation where we have millions or billions of keys and only few of them are accessed frequently, which is very likely in many practical applications.

All splay tree operations run in $O(\log n)$ time on average, where n is the number of entries in the tree. Any single operation can take $\Theta(n)$ time in the worst case.

Search Operation

The search operation in Splay tree does the standard BST search, in addition to search, it also splays (move a node to the root). If the search is successful, then the node that is found is splayed and becomes the new root. Else the last node accessed prior to reaching the NULL is splayed and becomes the new root.

There are following cases for the node being accessed.

1) Node is root We simply return the root, don't do anything else as the accessed node is already root.

2) Zig: Node is child of root (the node has no grandparent). Node is either a left child of root (we do a right rotation) or node is a right child of its parent (we do a left rotation).

T_1 , T_2 and T_3 are subtrees of the tree rooted with y (on left side) or x (on right side)

```

      y          x
     /\        /\
    x  T3 -----> T1 y
     /\
  
```

3) Node has both parent and grandparent. There can be following subcases.

.....**3.a) Zig-Zig and Zag-Zag** Node is left child of parent and parent is also left child of grand parent (Two right rotations) OR node is right child of its parent and parent is also right child of grand parent (Two Left Rotations).

Zig-Zig (Left Left Case):

```

      G          P          X
     /\        /\        /\
    P  T4  rightRotate(G) X  G  rightRotate(P) T1  P
     /\ -----> /\ -----> /\
    X  T3          T1 T2 T3 T4          T2  G
     /\
    T1 T2          T3 T4
  
```

Zag-Zag (Right Right Case):

```

      G          P          X
     /\        /\        /\
    T1 P  leftRotate(G) G  X  leftRotate(P) P  T4
     /\ -----> /\ -----> /\
    T2 X          T1 T2 T3 T4          G  T3
     /\
    T3 T4          T1 T2
  
```

.....**3.b) Zig-Zag and Zag-Zig** Node is left child of parent and parent is right child of grand parent (Left Rotation followed by right rotation) OR node is right child of its parent and parent is left child of grand parent (Right Rotation followed by

Zig-Zag (Left Right Case):

```

      G          G          X
     /\        /\        /\
    P  T4  leftRotate(P) X  T4  rightRotate(G) P  G
     /\ -----> /\ -----> /\
    T1 X          P  T3          T1 T2 T3 T4
     /\
    T2 T3          T1 T2
  
```

Zag-Zig (Right Left Case):

```

      G          G          X
     /\        /\        /\
    T1 P  rightRotate(P) T1 X  leftRotate(P) G  P
     /\ -----> /\ -----> /\
    X  T4          T2 P          T1 T2 T3 T4
     /\
    T2 T3          T3 T4
  
```

Example:

```

      100          100          [20]
     /\        /\        \
    50 200    50 200    50
   /  search(20) /  search(20) / \
  40 -----> [20] -----> 30 100
 /  1. Zig-Zig \ 2. Zig-Zig \ \
30   at 40    30   at 100   40 200
 /
[20]          40
  
```

The important thing to note is, the search or splay operation not only brings the searched key to root, but also balances the BST. For example in above case, height of BST is reduced by 1.

Implementation:

```
// The code is adopted from http://goo.gl/SDH9hH
#include<stdio.h>
#include<stdlib.h>
```

```

// An AVL tree node
struct node
{
    int key;
    struct node *left, *right;
};

/* Helper function that allocates a new node with the given key and
NULL left and right pointers. */
struct node* newNode(int key)
{
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->key = key;
    node->left = node->right = NULL;
    return (node);
}

// A utility function to right rotate subtree rooted with y
// See the diagram given above.
struct node *rightRotate(struct node *x)
{
    struct node *y = x->left;
    x->left = y->right;
    y->right = x;
    return y;
}

// A utility function to left rotate subtree rooted with x
// See the diagram given above.
struct node *leftRotate(struct node *x)
{
    struct node *y = x->right;
    x->right = y->left;
    y->left = x;
    return y;
}

// This function brings the key at root if key is present in tree.
// If key is not present, then it brings the last accessed item at
// root. This function modifies the tree and returns the new root
struct node *splay(struct node *root, int key)
{
    // Base cases: root is NULL or key is present at root
    if (root == NULL || root->key == key)
        return root;

    // Key lies in left subtree
    if (root->key > key)
    {
        // Key is not in tree, we are done
        if (root->left == NULL) return root;

        // Zig-Zig (Left Left)
        if (root->left->key > key)
        {
            // First recursively bring the key as root of left-left
            root->left->left = splay(root->left->left, key);

            // Do first rotation for root, second rotation is done after else
            root = rightRotate(root);
        }
        else if (root->left->key < key) // Zig-Zag (Left Right)
        {
            // First recursively bring the key as root of left-right
            root->left->right = splay(root->left->right, key);

            // Do first rotation for root->left
            if (root->left->right != NULL)
                root->left = leftRotate(root->left);
        }

        // Do second rotation for root
        return (root->left == NULL)? root: rightRotate(root);
    }
    else // Key lies in right subtree
    {
        // Key is not in tree, we are done
        if (root->right == NULL) return root;

        // Zag-Zig (Right Left)
        if (root->right->key > key)
        {
            // Bring the key as root of right-left
            root->right->left = splay(root->right->left, key);

            // Do first rotation for root->right
            if (root->right->left != NULL)
                root->right = rightRotate(root->right);
        }
        else if (root->right->key < key) // Zag-Zag (Right Right)
        {
            // Bring the key as root of right-right and do first rotation
            root->right->right = splay(root->right->right, key);
            root = leftRotate(root);
        }

        // Do second rotation for root
        return (root->right == NULL)? root: leftRotate(root);
    }
}

// The search function for Splay tree. Note that this function
// returns the new root of Splay Tree. If key is present in tree
// then, it is moved to root.
struct node *search(struct node *root, int key)
{
    return splay(root, key);
}

// A utility function to print preorder traversal of the tree.
// The function also prints height of every node
void preOrder(struct node *root)
{
    if (root != NULL)
    {

```

```

        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}

/* Drier program to test above function */
int main()
{
    struct node *root = newNode(100);
    root->left = newNode(50);
    root->right = newNode(200);
    root->left->left = newNode(40);
    root->left->left->left = newNode(30);
    root->left->left->left->left = newNode(20);

    root = search(root, 20);
    printf("Preorder traversal of the modified Splay tree is \n");
    preOrder(root);
    return 0;
}

```

Output:

```

Preorder traversal of the modified Splay tree is
20 50 30 40 100 200

```

Summary

- 1) Splay trees have excellent locality properties. Frequently accessed items are easy to find. Infrequent items are out of way.
- 2) All splay tree operations take $O(\log n)$ time on average. Splay trees can be rigorously shown to run in $O(\log n)$ average time per operation, over any sequence of operations (assuming we start from an empty tree)
- 3) Splay trees are simpler compared to AVL and Red-Black Trees as no extra field is required in every tree node.
- 4) Unlike AVL tree, a splay tree can change even with read-only operations like search.

Applications of Splay Trees

Splay trees have become the most widely used basic data structure invented in the last 30 years, because they're the fastest type of balanced search tree for many applications.

Splay trees are used in Windows NT (in the virtual memory, networking, and file system code), the gcc compiler and GNU C++ library, the sed string editor, Fore Systems network routers, the most popular implementation of Unix malloc, Lin

See [Splay Tree | Set 2 \(Insert\)](#) for splay tree insertion.

References:

<http://www.cs.berkeley.edu/~jrs/61b/lec/36>

<http://www.cs.cornell.edu/courses/cs3110/2009fa/recitations/rec-splay.html>

<http://courses.cs.washington.edu/courses/cse326/01au/lectures/SplayTrees.ppt>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner

Company Wise Coding Practice

[Advanced Data Structure](#)
[Advance Data Structures](#)
[Advanced Data Structures](#)
[Self-Balancing BST](#)
[splay-tree](#)

Splay Tree | Set 2 (Insert)

It is recommended to refer following post as prerequisite of this post.

[Splay Tree | Set 1 \(Search\)](#)

As discussed in the [previous post](#), Splay tree is a self-balancing data structure where the last accessed key is always at root. The insert operation is similar to Binary Search Tree insert with additional steps to make sure that the newly inserted key becomes the new root.

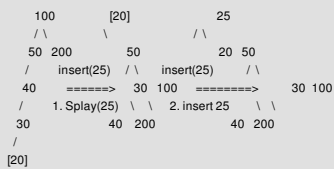
Following are different cases to insert a key k in splay tree.

- 1) Root is NULL: We simply allocate a new node and return it as root.
- 2) **Splay** the given key k. If k is already present, then it becomes the new root. If not present, then last accessed leaf node becomes the new root.
- 3) If new root's key is same as k, don't do anything as k is already present.
- 4) Else allocate memory for new node and compare root's key with k.
**4.a)** If k is smaller than root's key, make root as right child of new node, copy left child of root as left child of new node and make left child of root as NULL.

.....4.b) If k is greater than root's key, make root as left child of new node, copy right child of root as right child of new node and make right child of root as NULL.

5) Return new node as new root of tree.

Example:



// This code is adopted from <http://algs4.cs.princeton.edu/33balanced/SplayBST.java.html>

```
#include<stdio.h>
#include<stdlib.h>
```

```
// An AVL tree node
struct node
```

```
{
    int key;
    struct node *left, *right;
};
```

```
/* Helper function that allocates a new node with the given key and
   NULL left and right pointers. */
```

```
struct node* newNode(int key)
{
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->key = key;
    node->left = node->right = NULL;
    return (node);
}
```

```
// A utility function to right rotate subtree rooted with y
// See the diagram given above.
```

```
struct node *rightRotate(struct node *x)
{
    struct node *y = x->left;
    x->left = y->right;
    y->right = x;
    return y;
}
```

```
// A utility function to left rotate subtree rooted with x
// See the diagram given above.
```

```
struct node *leftRotate(struct node *x)
{
    struct node *y = x->right;
    x->right = y->left;
    y->left = x;
    return y;
}
```

```
// This function brings the key at root if key is present in tree.
```

```
// If key is not present, then it brings the last accessed item at
// root. This function modifies the tree and returns the new root
```

```
struct node *splay(struct node *root, int key)
```

```
{
    // Base cases: root is NULL or key is present at root
    if (root == NULL || root->key == key)
        return root;
```

```
    // Key lies in left subtree
    if (root->key > key)
```

```
{
    // Key is not in tree, we are done
    if (root->left == NULL) return root;
```

```
    // Zig-Zig (Left Left)
    if (root->left->key > key)
```

```
{
    // First recursively bring the key as root of left-left
    root->left->left = splay(root->left->left, key);
```

```
    // Do first rotation for root, second rotation is done after else
    root = rightRotate(root);
```

```
}
else if (root->left->key < key) // Zig-Zag (Left Right)
```

```
{
    // First recursively bring the key as root of left-right
    root->left->right = splay(root->left->right, key);
```

```
    // Do first rotation for root->left
    if (root->left->right != NULL)
        root->left = leftRotate(root->left);
```

```
    // Do second rotation for root
    return (root->left == NULL)? root: rightRotate(root);
}
```

```
else // Key lies in right subtree
```

```
{
    // Key is not in tree, we are done
    if (root->right == NULL) return root;
```

```
    // Zig-Zag (Right Left)
    if (root->right->key > key)
```

```
{
    // Bring the key as root of right-left
    root->right->left = splay(root->right->left, key);
```

```
    // Do first rotation for root->right
    if (root->right->left != NULL)
        root->right = rightRotate(root->right);
```

```
}
else if (root->right->key < key) // Zag-Zag (Right Right)
```

```
{
    // Bring the key as root of right-right and do first rotation
    root->right->right = splay(root->right->right, key);
    root = leftRotate(root);
}
```

```

    }

    // Do second rotation for root
    return (root->right == NULL)? root: leftRotate(root);
}
}

// Function to insert a new key k in splay tree with given root
struct node *insert(struct node *root, int k)
{
    // Simple Case: If tree is empty
    if (root == NULL) return newNode(k);

    // Bring the closest leaf node to root
    root = splay(root, k);

    // If key is already present, then return
    if (root->key == k) return root;

    // Otherwise allocate memory for new node
    struct node *newnode = newNode(k);

    // If root's key is greater, make root as right child
    // of newnode and copy the left child of root to newnode
    if (root->key > k)
    {
        newnode->right = root;
        newnode->left = root->left;
        root->left = NULL;
    }

    // If root's key is smaller, make root as left child
    // of newnode and copy the right child of root to newnode
    else
    {
        newnode->left = root;
        newnode->right = root->right;
        root->right = NULL;
    }

    return newnode; // newnode becomes new root
}

// A utility function to print preorder traversal of the tree.
// The function also prints height of every node
void preOrder(struct node *root)
{
    if (root != NULL)
    {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}

/* Driver program to test above function */
int main()
{
    struct node *root = newNode(100);
    root->left = newNode(50);
    root->right = newNode(200);
    root->left->left = newNode(40);
    root->left->left->left = newNode(30);
    root->left->left->left->left = newNode(20);
    root = insert(root, 25);
    printf("Preorder traversal of the modified Splay tree is\n");
    preOrder(root);
    return 0;
}

```

Output:

```

Preorder traversal of the modified Splay tree is
25 20 50 30 40 100 200

```

This article is compiled by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Advance Data Structures
Advanced Data Structures
Self-Balancing-BST
splay-tree

B-Tree | Set 1 (Introduction)

B-Tree is a self-balancing search tree. In most of the other self-balancing search trees (like AVL and Red Black Trees), it is assumed that everything is in main memory. To understand use of B-Trees, we must think of huge amount of data that cannot fit in main memory. When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, max, min, ..etc) require $O(h)$ disk accesses where h is height of the tree. B-tree is a fat tree. Height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. Generally, a B-Tree node size is kept equal to the disk block size. Since h is low for B-Tree, total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red Black Tree, ..etc.

Properties of B-Tree

- 1) All leaves are at same level.
- 2) A B-Tree is defined by the term *minimum degree* 't'. The value of t depends upon disk block size.
- 3) Every node except root must contain at least $t-1$ keys. Root may contain minimum 1 key.
- 4) All nodes (including root) may contain at most $2t - 1$ keys.
- 5) Number of children of a node is equal to the number of keys in it plus 1.
- 6) All keys of a node are sorted in increasing order. The child between two keys k_1 and k_2 contains all keys in range from k_1 and k_2 .
- 7) B-Tree grows and shrinks from root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
- 8) Like other balanced Binary Search Trees, time complexity to search, insert and delete is $O(\log n)$.

Following is an example B-Tree of minimum degree 3. Note that in practical B-Trees, the value of minimum degree is much more than 3.

BTreeIntro



Search

Search is similar to search in Binary Search Tree. Let the key to be searched be k . We start from root and recursively traverse down. For every visited non-leaf node, if the node has key, we simply return the node. Otherwise we recur down to the appropriate child (The child which is just before the first greater key) of the node. If we reach a leaf node and don't find k in the leaf node, we return NULL.

Traverse

Traversal is also similar to Inorder traversal of Binary Tree. We start from the leftmost child, recursively print the leftmost child, then repeat the same process for remaining children and keys. In the end, recursively print the rightmost child.

```
// C++ implementation of search() and traverse() methods
#include<iostream>
using namespace std;

// A BTree node
class BTreeNode
{
    int *keys; // An array of keys
    int t; // Minimum degree (defines the range for number of keys)
    BTreeNode **C; // An array of child pointers
    int n; // Current number of keys
    bool leaf; // Is true when node is leaf. Otherwise false
public:
    BTreeNode(int _t, bool _leaf); // Constructor

    // A function to traverse all nodes in a subtree rooted with this node
    void traverse();

    // A function to search a key in subtree rooted with this node.
    BTreeNode *search(int k); // returns NULL if k is not present.
};

// Make BTree friend of this so that we can access private members of this
// class in BTree functions
friend class BTree;
};

// A BTree
class BTree
{
    BTreeNode *root; // Pointer to root node
    int t; // Minimum degree
public:
    // Constructor (Initializes tree as empty)
    BTree(int _t)
    { root = NULL; t = _t; }

    // function to traverse the tree
    void traverse()
    { if (root != NULL) root->traverse(); }

    // function to search a key in this tree
    BTreeNode* search(int k)
    { return (root == NULL)? NULL : root->search(k); }
};

// Constructor for BTreeNode class
BTreeNode::BTreeNode(int _t, bool _leaf)
{
    // Copy the given minimum degree and leaf property
    t = _t;
    leaf = _leaf;

    // Allocate memory for maximum number of possible keys
    // and child pointers
    keys = new int[2*t-1];
    C = new BTreeNode*[2*t];

    // Initialize the number of keys as 0
    n = 0;
}

// Function to traverse all nodes in a subtree rooted with this node
void BTreeNode::traverse()
{
    // There are n keys and n+1 children, travers through n keys
    // and first n children
    int i;
    for (i = 0; i < n; i++)
    {
        // If this is not leaf, then before printing key[i],
        // traverse the subtree rooted with child C[i].
        if (!leaf == false)
            C[i]->traverse();
        cout << " " << keys[i];
    }

    // Print the subtree rooted with last child
    if (leaf == false)
        C[i]->traverse();
}

// Function to search key k in subtree rooted with this node
BTreeNode *BTreeNode::search(int k)
{
    // Find the first key greater than or equal to k
    int i = 0;
    while (i < n && k > keys[i])
        i++;

    // If the found key is equal to k, return this node
    if (keys[i] == k)
        return this;

    // If key is not found here and this is a leaf node
    if (leaf == true)
        return NULL;
}
```



```
// Go to the appropriate child
return C[j]->search(k);
}
```

The above code doesn't contain driver program. We will be covering the complete program in our next post on B-Tree Insertion.

There are two conventions to define a B-Tree, one is to define by minimum degree (followed in [Cormen book](#)), second is define by order. We have followed the minimum degree convention and will be following same in coming posts on B-Tree. The variable names used in the above program are also kept same as Cormen book for better readability.

Insertion and Deletion

[B-Tree Insertion](#)

[B-Tree Deletion](#)

References:

[Introduction to Algorithms 3rd Edition](#) by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

[Advanced Data Structure](#)
[Advance Data Structures](#)
[Advanced Data Structures](#)
[Self-Balancing-BST](#)

B-Tree | Set 2 (Insert)

In the [previous post](#), we introduced B-Tree. We also discussed search() and traverse() functions.

In this post, insert() operation is discussed. A new key is always inserted at leaf node. Let the key to be inserted be k. Like BST, we start from root and traverse down till we reach a leaf node. Once we reach a leaf node, we insert the key in that leaf node. Unlike BSTs, we have a predefined range on number of keys that a node can contain. So before inserting a key to node, we make sure that the node has extra space.

How to make sure that a node has space available for key before the key is inserted? We use an operation called splitChild() that is used to split a child of a node. See the following diagram to understand split. In the following diagram, child y of x is being split into two nodes y and z. Note that the splitChild operation moves a key up and this is the reason B-Trees grow up unlike BSTs which grow down.



As discussed above, to insert a new key, we go down from root to leaf. Before traversing down to a node, we first check if the node is full. If the node is full, we split it to create space. Following is complete algorithm.

Insertion

1) Initialize x as root.

2) While x is not leaf, do following

..a) Find the child of x that is going to be traversed next. Let the child be y.

..b) If y is not full, change x to point to y.

..c) If y is full, split it and change x to point to one of the two parts of y. If k is smaller than mid key in y, then set x as first part of y. Else second part of y. When we split y, we move a key from y to its parent x.

3) The loop in step 2 stops when x is leaf. x must have space for 1 extra key as we have been splitting all nodes in advance. So simply insert k to x.

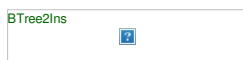
Note that the algorithm follows the Cormen book. It is actually a proactive insertion algorithm where before going down to a node, we split it if it is full. The advantage of splitting before is, we never traverse a node twice. If we don't split a node before going down to it and split it only if new key is inserted (reactive), we may end up traversing all nodes again from leaf to root. This happens in cases when all nodes on the path from root to leaf are full. So when we come to the leaf node, we split it and move a key up. Moving a key up will cause a split in parent node (because parent was already full). This cascading effect never happens in this proactive insertion algorithm. There is a disadvantage of this proactive insertion though, we may do unnecessary splits.

Let us understand the algorithm with an example tree of minimum degree 't' as 3 and a sequence of integers 10, 20, 30, 40, 50, 60, 70, 80 and 90 in an initially empty B-Tree.

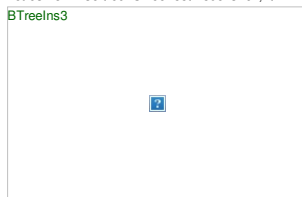
Initially root is NULL. Let us first insert 10.



Let us now insert 20, 30, 40 and 50. They all will be inserted in root because maximum number of keys a node can accommodate is $2^t - 1$ which is 5.



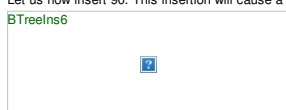
Let us now insert 60. Since root node is full, it will first split into two, then 60 will be inserted into the appropriate child.



Let us now insert 70 and 80. These new keys will be inserted into the appropriate leaf without any split.



Let us now insert 90. This insertion will cause a split. The middle key will go up to the parent.



See [this](#) for more examples.

Following is C++ implementation of the above proactive algorithm.

```
// C++ program for B-Tree insertion
#include<iostream>
using namespace std;

// A BTree node
class BTreeNode
{
```

```

int *keys; // An array of keys
int t; // Minimum degree (defines the range for number of keys)
BTreeNode **C; // An array of child pointers
int n; // Current number of keys
bool leaf; // Is true when node is leaf. Otherwise false

public:
    BTreeNode(int _t, bool _leaf); // Constructor

    // A utility function to insert a new key in the subtree rooted with
    // this node. The assumption is, the node must be non-full when this
    // function is called
    void insertNonFull(int k);

    // A utility function to split the child y of this node. i is index of y in
    // child array C[]. The Child y must be full when this function is called
    void splitChild(int i, BTreeNode *y);

    // A function to traverse all nodes in a subtree rooted with this node
    void traverse();

    // A function to search a key in subtree rooted with this node.
    BTreeNode *search(int k); // returns NULL if k is not present.

// Make BTree friend of this so that we can access private members of this
// class in BTree functions
friend class BTree;
};

// A BTree
class BTree
{
    BTreeNode *root; // Pointer to root node
    int t; // Minimum degree
public:
    // Constructor (Initializes tree as empty)
    BTree(int _t)
    { root = NULL; t = _t; }

    // function to traverse the tree
    void traverse()
    { if (root != NULL) root->traverse(); }

    // function to search a key in this tree
    BTreeNode* search(int k)
    { return (root == NULL)? NULL : root->search(k); }

    // The main function that inserts a new key in this B-Tree
    void insert(int k);
};

// Constructor for BTreeNode class
BTreeNode::BTreeNode(int t1, bool leaf1)
{
    // Copy the given minimum degree and leaf property
    t = t1;
    leaf = leaf1;

    // Allocate memory for maximum number of possible keys
    // and child pointers
    keys = new int[2*t-1];
    C = new BTreeNode *[2*t];

    // Initialize the number of keys as 0
    n = 0;
}

// Function to traverse all nodes in a subtree rooted with this node
void BTreeNode::traverse()
{
    // There are n keys and n+1 children, travers through n keys
    // and first n children
    int i;
    for (i = 0; i < n; i++)
    {
        // If this is not leaf, then before printing key[i],
        // traverse the subtree rooted with child C[i].
        if (leaf == false)
            C[i]->traverse();
        cout << " " << keys[i];
    }

    // Print the subtree rooted with last child
    if (leaf == false)
        C[i]->traverse();
}

// Function to search key k in subtree rooted with this node
BTreeNode *BTreeNode::search(int k)
{
    // Find the first key greater than or equal to k
    int i = 0;
    while (i < n && k > keys[i])
        i++;

    // If the found key is equal to k, return this node
    if (keys[i] == k)
        return this;

    // If key is not found here and this is a leaf node
    if (leaf == true)
        return NULL;

    // Go to the appropriate child
    return C[i]->search(k);
}

// The main function that inserts a new key in this B-Tree
void BTree::insert(int k)
{
    // If tree is empty
    if (root == NULL)
    {
        // Allocate memory for root
        root = new BTreeNode(t, true);
        root->keys[0] = k; // Insert key
        root->n = 1; // Update number of keys in root
    }
}

```

```

}
else // If tree is not empty
{
    // If root is full, then tree grows in height
    if (root->n == 2*t-1)
    {
        // Allocate memory for new root
        BTreeNode *s = new BTreeNode(t, false);

        // Make old root as child of new root
        s->C[0] = root;

        // Split the old root and move 1 key to the new root
        s->splitChild(0, root);

        // New root has two children now. Decide which of the
        // two children is going to have new key
        int i = 0;
        if (s->keys[0] < k)
            i++;
        s->C[i] -> insertNonFull(k);

        // Change root
        root = s;
    }
    else // If root is not full, call insertNonFull for root
        root->insertNonFull(k);
}
}

// A utility function to insert a new key in this node
// The assumption is, the node must be non-full when this
// function is called
void BTreeNode::insertNonFull(int k)
{
    // Initialize index as index of rightmost element
    int i = n-1;

    // If this is a leaf node
    if (leaf == true)
    {
        // The following loop does two things
        // a) Finds the location of new key to be inserted
        // b) Moves all greater keys to one place ahead
        while (i >= 0 && keys[i] > k)
        {
            keys[i+1] = keys[i];
            i--;
        }

        // Insert the new key at found location
        keys[i+1] = k;
        n = n+1;
    }
    else // If this node is not leaf
    {
        // Find the child which is going to have the new key
        while (i >= 0 && keys[i] > k)
            i--;

        // See if the found child is full
        if (C[i+1]->n == 2*t-1)
        {
            // If the child is full, then split it
            splitChild(i+1, C[i+1]);

            // After split, the middle key of C[i] goes up and
            // C[i] is splitted into two. See which of the two
            // is going to have the new key
            if (keys[i+1] < k)
                i++;
        }
        C[i+1] -> insertNonFull(k);
    }
}

// A utility function to split the child y of this node
// Note that y must be full when this function is called
void BTreeNode::splitChild(int i, BTreeNode *y)
{
    // Create a new node which is going to store (t-1) keys
    // of y
    BTreeNode *z = new BTreeNode(y->t, y->leaf);
    z->n = t-1;

    // Copy the last (t-1) keys of y to z
    for (int j = 0; j < t-1; j++)
        z->keys[j] = y->keys[j+t];

    // Copy the last t children of y to z
    if (y->leaf == false)
    {
        for (int j = 0; j < t; j++)
            z->C[j] = y->C[j+t];
    }

    // Reduce the number of keys in y
    y->n = t-1;

    // Since this node is going to have a new child,
    // create space of new child
    for (int j = n; j >= i+1; j--)
        C[j+1] = C[j];

    // Link the new child to this node
    C[i+1] = z;

    // A key of y will move to this node. Find location of
    // new key and move all greater keys one space ahead
    for (int j = n-1; j >= i; j--)
        keys[j+1] = keys[j];

    // Copy the middle key of y to this node
    keys[i] = y->keys[t-1];

    // Increment count of keys in this node

```

```

n = n + 1;
}

// Driver program to test above functions
int main()
{
    BTree t(3); // A B-Tree with minium degree 3
    t.insert(10);
    t.insert(20);
    t.insert(5);
    t.insert(6);
    t.insert(12);
    t.insert(30);
    t.insert(7);
    t.insert(17);

    cout << "Traversal of the constucted tree is ";
    t.traverse();

    int k = 6;
    (t.search(k) != NULL)? cout << "nPresent" : cout << "nNot Present";

    k = 15;
    (t.search(k) != NULL)? cout << "nPresent" : cout << "nNot Present";

    return 0;
}

```

Output:

```

Traversal of the constucted tree is 5 6 7 10 12 17 20 30
Present
Not Present

```

References:

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest
<http://www.cs.utexas.edu/users/djimeenez/utsa/cs3343/lecture17.html>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
 Advance Data Structures
 Advanced Data Structures

B-Tree | Set 3 (Delete)

It is recommended to refer following posts as prerequisite of this post.

[B-Tree | Set 1 \(Introduction\)](#)

[B-Tree | Set 2 \(Insert\)](#)

B-Tree is a type of a multi-way search tree. So, if you are not familiar with multi-way search trees in general, it is better to take a look at [this video lecture from IIT-Delhi](#), before proceeding further. Once you get the basics of a multi-way search tree clear, B-Tree operations will be easier to understand.

Source of the following explanation and algorithm is [Introduction to Algorithms 3rd Edition](#) by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

Deletion process:

Deletion from a B-tree is more complicated than insertion, because we can delete a key from any node—not just a leaf—and when we delete a key from an internal node, we will have to rearrange the node's children.

As in insertion, we must make sure the deletion doesn't violate the [B-tree properties](#). Just as we had to ensure that a node didn't get too big due to insertion, we must ensure that a node doesn't get too small during deletion (except that the root is allowed to have fewer than the minimum number $t-1$ of keys). Just as a simple insertion algorithm might have to back up if a node on the path to where the key was to be inserted was full, a simple approach to deletion might have to back up if a node (other than the root) along the path to where the key is to be deleted has the minimum number of keys.

The deletion procedure deletes the key k from the subtree rooted at x . This procedure guarantees that whenever it calls itself recursively on a node x , the number of keys in x is at least the minimum degree t . Note that this condition requires one more key than the minimum required by the usual B-tree conditions, so that sometimes a key may have to be moved into a child node before recursion descends to that child. This strengthened condition allows us to delete a key from the tree in one downward pass without having to "back up" (with one exception, which we'll explain). You should interpret the following specification for deletion from a B-tree with the understanding that if the root node x ever becomes an internal node having no keys (this situation can occur in cases 2c and 3b then we delete x , and x 's only child $x.c[1]$ becomes the new root of the tree, decreasing the height of the tree by one and preserving the property that the root of the tree contains at least one key (unless the tree is empty).

We sketch how deletion works with various cases of deleting keys from a B-tree.

- If the key k is in node x and x is a leaf, delete the key k from x .
- If the key k is in node x and x is an internal node, do the following.
 - If the child y that precedes k in node x has at least t keys, then find the predecessor k_0 of k in the sub-tree rooted at y . Recursively delete k_0 , and replace k by k_0 in x . (We can find k_0 and delete it in a single downward pass.)
 - If y has fewer than t keys, then, symmetrically, examine the child z that follows k in node x . If z has at least t keys, then find the successor k_0 of k in the subtree rooted at z . Recursively delete k_0 , and replace k by k_0 in x . (We can find k_0 and delete it in a single downward pass.)
 - Otherwise, if both y and z have only $t-1$ keys, merge k and all of z into y , so that x loses both k and the pointer to z , and y now contains $2t-1$ keys. Then free z and recursively delete k from y .
- If the key k is not present in internal node x , determine the root $x.c[i]$ of the appropriate subtree that must contain k , if k is in the tree at all. If $x.c[i]$ has only $t-1$ keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys. Then finish by recursing on the appropriate child of x .
 - If $x.c[i]$ has only $t-1$ keys but has an immediate sibling with at least t keys, give $x.c[i]$ an extra key by moving a key from x down into $x.c[i]$, moving a key from $x.c[i]$'s immediate left or right sibling up into x , and moving the appropriate child pointer from the sibling into $x.c[i]$.
 - If $x.c[i]$ and both of $x.c[i]$'s immediate siblings have $t-1$ keys, merge $x.c[i]$ with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

Since most of the keys in a B-tree are in the leaves, deletion operations are most often used to delete keys from leaves. The recursive delete procedure then acts in one downward pass through the tree, without having to back up. When deleting a key in an internal node, however, the procedure makes a downward pass through the tree but may have to return to the node from which the key was deleted to replace the key with its predecessor or successor (cases 2a and 2b).

The following figures from [CLRS book](#) explain the deletion process.

BTreeDelet1



BTreeDelet2



Implementation:

Following is C++ implementation of deletion process.

```
/* The following program performs deletion on a B-Tree. It contains functions
specific for deletion along with all the other functions provided in the
previous articles on B-Trees. See http://www.geeksforgeeks.org/b-tree-set-1-introduction-2/
for previous article.
```

The deletion function has been compartmentalized into 8 functions for ease
of understanding and clarity

The following functions are exclusive for deletion

In class BTreeNode:

- 1) remove
- 2) removeFromLeaf
- 3) removeFromNonLeaf
- 4) getPred
- 5) getSucc
- 6) borrowFromPrev
- 7) borrowFromNext
- 8) merge
- 9) findKey

In class BTree:

- 1) remove

The removal of a key from a B-Tree is a fairly complicated process. The program handles
all the 6 different cases that might arise while removing a key.

Testing: The code has been tested using the B-Tree provided in the CLRS book(included
in the main function) along with other cases.

Reference: CLRS3 - Chapter 18 - (499-502)

It is advised to read the material in CLRS before taking a look at the code. */

```
#include<iostream>
using namespace std;
```

```
// A BTree node
class BTreeNode
```

```
{
    int *keys; // An array of keys
    int t; // Minimum degree (defines the range for number of keys)
    BTreeNode **C; // An array of child pointers
    int n; // Current number of keys
    bool leaf; // Is true when node is leaf. Otherwise false
```

```
public:
```

```
    BTreeNode(int _t, bool _leaf); // Constructor
```

```
    // A function to traverse all nodes in a subtree rooted with this node
    void traverse();
```

```
    // A function to search a key in subtree rooted with this node.
    BTreeNode *search(int k); // returns NULL if k is not present.
```

```
    // A function that returns the index of the first key that is greater
    // or equal to k
    int findKey(int k);
```

```
    // A utility function to insert a new key in the subtree rooted with
    // this node. The assumption is, the node must be non-full when this
    // function is called
    void insertNonFull(int k);
```

```

// A utility function to split the child y of this node. i is index
// of y in child array C[]. The Child y must be full when this
// function is called
void splitChild(int i, BTreeNode *y);

// A wrapper function to remove the key k in subtree rooted with
// this node.
void remove(int k);

// A function to remove the key present in idx-th position in
// this node which is a leaf
void removeFromLeaf(int idx);

// A function to remove the key present in idx-th position in
// this node which is a non-leaf node
void removeFromNonLeaf(int idx);

// A function to get the predecessor of the key- where the key
// is present in the idx-th position in the node
int getPred(int idx);

// A function to get the successor of the key- where the key
// is present in the idx-th position in the node
int getSucc(int idx);

// A function to fill up the child node present in the idx-th
// position in the C[] array if that child has less than t-1 keys
void fill(int idx);

// A function to borrow a key from the C[idx-1]-th node and place
// it in C[idx]th node
void borrowFromPrev(int idx);

// A function to borrow a key from the C[idx+1]-th node and place it
// in C[idx]th node
void borrowFromNext(int idx);

// A function to merge idx-th child of the node with (idx+1)th child of
// the node
void merge(int idx);

// Make BTree friend of this so that we can access private members of
// this class in BTree functions
friend class BTree;
};

class BTree
{
    BTreeNode *root; // Pointer to root node
    int t; // Minimum degree
public:

    // Constructor (Initializes tree as empty)
    BTree(int _t)
    {
        root = NULL;
        t = _t;
    }

    void traverse()
    {
        if (root != NULL) root->traverse();
    }

    // function to search a key in this tree
    BTreeNode* search(int k)
    {
        return (root == NULL)? NULL : root->search(k);
    }

    // The main function that inserts a new key in this B-Tree
    void insert(int k);

    // The main function that removes a new key in this B-Tree
    void remove(int k);
};

BTreeNode::BTreeNode(int t1, bool leaf1)
{
    // Copy the given minimum degree and leaf property
    t = t1;
    leaf = leaf1;

    // Allocate memory for maximum number of possible keys
    // and child pointers
    keys = new int[2*t-1];
    C = new BTreeNode *[2*t];

    // Initialize the number of keys as 0
    n = 0;
}

// A utility function that returns the index of the first key that is
// greater than or equal to k
int BTreeNode::findKey(int k)
{
    int idx=0;
    while (idx<n && keys[idx] < k)
        ++idx;
    return idx;
}

// A function to remove the key k from the sub-tree rooted with this node
void BTreeNode::remove(int k)
{
    int idx = findKey(k);

    // The key to be removed is present in this node
    if (idx < n && keys[idx] == k)
    {
        // If the node is a leaf node - removeFromLeaf is called
        // Otherwise, removeFromNonLeaf function is called
        if (!leaf)

```

```

        removeFromLeaf(idx);
    else
        removeFromNonLeaf(idx);
}
else
{
    // If this node is a leaf node, then the key is not present in tree
    if (!leaf)
    {
        cout << "The key "<< k <<" is does not exist in the tree\n";
        return;
    }

    // The key to be removed is present in the sub-tree rooted with this node
    // The flag indicates whether the key is present in the sub-tree rooted
    // with the last child of this node
    bool flag = ( (idx==n)? true : false );

    // If the child where the key is supposed to exist has less than t keys,
    // we fill that child
    if (C[idx]>n < t)
        fill(idx);

    // If the last child has been merged, it must have merged with the previous
    // child and so we recurse on the (idx-1)th child. Else, we recurse on the
    // (idx)th child which now has atleast t keys
    if (flag && idx > n)
        C[idx-1]>remove(k);
    else
        C[idx]>remove(k);
}
return;
}

// A function to remove the idx-th key from this node - which is a leaf node
void BTreeNode::removeFromLeaf (int idx)
{
    // Move all the keys after the idx-th pos one place backward
    for (int i=idx+1; i<n; ++i)
        keys[i-1] = keys[i];

    // Reduce the count of keys
    n--;

    return;
}

// A function to remove the idx-th key from this node - which is a non-leaf node
void BTreeNode::removeFromNonLeaf(int idx)
{
    int k = keys[idx];

    // If the child that precedes k (C[idx]) has atleast t keys,
    // find the predecessor 'pred' of k in the subtree rooted at
    // C[idx]. Replace k by pred. Recursively delete pred
    // in C[idx]
    if (C[idx]>n >= t)
    {
        int pred = getPred(idx);
        keys[idx] = pred;
        C[idx]>remove(pred);
    }

    // If the child C[idx] has less than t keys, examine C[idx+1].
    // If C[idx+1] has atleast t keys, find the successor 'succ' of k in
    // the subtree rooted at C[idx+1]
    // Replace k by succ
    // Recursively delete succ in C[idx+1]
    else if (C[idx+1]>n >= t)
    {
        int succ = getSucc(idx);
        keys[idx] = succ;
        C[idx+1]>remove(succ);
    }

    // If both C[idx] and C[idx+1] has less than t keys, merge k and all of C[idx+1]
    // into C[idx]
    // Now C[idx] contains 2t-1 keys
    // Free C[idx+1] and recursively delete k from C[idx]
    else
    {
        merge(idx);
        C[idx]>remove(k);
    }
    return;
}

// A function to get predecessor of keys[idx]
int BTreeNode::getPred(int idx)
{
    // Keep moving to the right most node until we reach a leaf
    BTreeNode *cur=C[idx];
    while (!cur->leaf)
        cur = cur->C[cur->n];

    // Return the last key of the leaf
    return cur->keys[cur->n-1];
}

int BTreeNode::getSucc(int idx)
{
    // Keep moving the left most node starting from C[idx+1] until we reach a leaf
    BTreeNode *cur = C[idx+1];
    while (!cur->leaf)
        cur = cur->C[0];

    // Return the first key of the leaf
    return cur->keys[0];
}

// A function to fill child C[idx] which has less than t-1 keys
void BTreeNode::fill(int idx)

```

```

{

// If the previous child(C[idx-1]) has more than t-1 keys, borrow a key
// from that child
if (idx!=0 && C[idx-1]->n>=t)
    borrowFromPrev(idx);

// If the next child(C[idx+1]) has more than t-1 keys, borrow a key
// from that child
else if (idx!=n && C[idx+1]->n>=t)
    borrowFromNext(idx);

// Merge C[idx] with its sibling
// If C[idx] is the last child, merge it with its previous sibling
// Otherwise merge it with its next sibling
else
{
    if (idx != n)
        merge(idx);
    else
        merge(idx-1);
}
return;
}

// A function to borrow a key from C[idx-1] and insert it
// into C[idx]
void BTreeNode::borrowFromPrev(int idx)
{
    BTreeNode *child=C[idx];
    BTreeNode *sibling=C[idx-1];

    // The last key from C[idx-1] goes up to the parent and key[idx-1]
    // from parent is inserted as the first key in C[idx]. Thus, the  loses
    // sibling one key and child gains one key

    // Moving all key in C[idx] one step ahead
    for (int i=child->n-1; i>=0; --i)
        child->keys[i+1] = child->keys[i];

    // If C[idx] is not a leaf, move all its child pointers one step ahead
    if (!child->leaf)
    {
        for(int i=child->n; i>=0; --i)
            child->C[i+1] = child->C[i];
    }

    // Setting child's first key equal to keys[idx-1] from the current node
    child->keys[0] = keys[idx-1];

    // Moving sibling's last child as C[idx]'s first child
    if (!leaf)
        child->C[0] = sibling->C[sibling->n];

    // Moving the key from the sibling to the parent
    // This reduces the number of keys in the sibling
    keys[idx-1] = sibling->keys[sibling->n-1];

    child->n += 1;
    sibling->n -= 1;

    return;
}

// A function to borrow a key from the C[idx+1] and place
// it in C[idx]
void BTreeNode::borrowFromNext(int idx)
{
    BTreeNode *child=C[idx];
    BTreeNode *sibling=C[idx+1];

    // keys[idx] is inserted as the last key in C[idx]
    child->keys[(child->n)] = keys[idx];

    // Sibling's first child is inserted as the last child
    // into C[idx]
    if (!(!child->leaf))
        child->C[(child->n)+1] = sibling->C[0];

    //The first key from sibling is inserted into keys[idx]
    keys[idx] = sibling->keys[0];

    // Moving all keys in sibling one step behind
    for (int i=1; i<sibling->n; ++i)
        sibling->keys[i-1] = sibling->keys[i];

    // Moving the child pointers one step behind
    if (!sibling->leaf)
    {
        for(int i=1; i<=sibling->n; ++i)
            sibling->C[i-1] = sibling->C[i];
    }

    // Increasing and decreasing the key count of C[idx] and C[idx+1]
    // respectively
    child->n += 1;
    sibling->n -= 1;

    return;
}

// A function to merge C[idx] with C[idx+1]
// C[idx+1] is freed after merging
void BTreeNode::merge(int idx)
{
    BTreeNode *child = C[idx];
    BTreeNode *sibling = C[idx+1];

    // Pulling a key from the current node and inserting it into (t-1)th
    // position of C[idx]
    child->keys[t-1] = keys[idx];

    // Copying the keys from C[idx+1] to C[idx] at the end
    for (int i=0; i<sibling->n; ++i)

```



```

    child->keys[i++] = sibling->keys[j];

// Copying the child pointers from C[idx+1] to C[idx]
if (!child->leaf)
{
    for(int i=0; i<=sibling->n; ++i)
        child->C[i+i] = sibling->C[i];
}

// Moving all keys after idx in the current node one step before -
// to fill the gap created by moving keys[idx] to C[idx]
for (int i=idx+1; i<n; ++i)
    keys[i-1] = keys[i];

// Moving the child pointers after (idx+1) in the current node one
// step before
for (int i=idx+2; i<=n; ++i)
    C[i-1] = C[i];

// Updating the key count of child and the current node
child->n += sibling->n+1;
n--;

// Freeing the memory occupied by sibling
delete(sibling);
return;
}

// The main function that inserts a new key in this B-Tree
void BTree::insert(int k)
{
    // If tree is empty
    if (root == NULL)
    {
        // Allocate memory for root
        root = new BTreeNode(t, true);
        root->keys[0] = k; // Insert key
        root->n = 1; // Update number of keys in root
    }
    else // If tree is not empty
    {
        // If root is full, then tree grows in height
        if (root->n == 2*t-1)
        {
            // Allocate memory for new root
            BTreeNode *s = new BTreeNode(t, false);

            // Make old root as child of new root
            s->C[0] = root;

            // Split the old root and move 1 key to the new root
            s->splitChild(0, root);

            // New root has two children now. Decide which of the
            // two children is going to have new key
            int i = 0;
            if (s->keys[0] < k)
                i++;
            s->C[i]->insertNonFull(k);

            // Change root
            root = s;
        }
        else // If root is not full, call insertNonFull for root
            root->insertNonFull(k);
    }
}

// A utility function to insert a new key in this node
// The assumption is, the node must be non-full when this
// function is called
void BTreeNode::insertNonFull(int k)
{
    // Initialize index as index of rightmost element
    int i = n-1;

    // If this is a leaf node
    if (leaf == true)
    {
        // The following loop does two things
        // a) Finds the location of new key to be inserted
        // b) Moves all greater keys to one place ahead
        while (i >= 0 && keys[i] > k)
        {
            keys[i+1] = keys[i];
            i--;
        }

        // Insert the new key at found location
        keys[i+1] = k;
        n = n+1;
    }
    else // If this node is not leaf
    {
        // Find the child which is going to have the new key
        while (i >= 0 && keys[i] > k)
            i--;

        // See if the found child is full
        if (C[i+1]->n == 2*t-1)
        {
            // If the child is full, then split it
            splitChild(i+1, C[i+1]);

            // After split, the middle key of C[i] goes up and
            // C[i] is splitted into two. See which of the two
            // is going to have the new key
            if (keys[i+1] < k)
                i++;
        }
        C[i+1]->insertNonFull(k);
    }
}

// A utility function to split the child y of this node
// Note that y must be full when this function is called

```

```

void BTreeNode::splitChild(int i, BTreeNode *y)
{
    // Create a new node which is going to store (t-1) keys
    // of y
    BTreeNode *z = new BTreeNode(y->t, y->leaf);
    z->n = t - 1;

    // Copy the last (t-1) keys of y to z
    for (int j = 0; j < t-1; j++)
        z->keys[j] = y->keys[j+t];

    // Copy the last t children of y to z
    if (y->leaf == false)
    {
        for (int j = 0; j < t; j++)
            z->C[j] = y->C[j+t];
    }

    // Reduce the number of keys in y
    y->n = t - 1;

    // Since this node is going to have a new child,
    // create space of new child
    for (int j = n; j >= i+1; j--)
        C[j+1] = C[j];

    // Link the new child to this node
    C[i+1] = z;

    // A key of y will move to this node. Find location of
    // new key and move all greater keys one space ahead
    for (int j = n-1; j >= i; j--)
        keys[j+1] = keys[j];

    // Copy the middle key of y to this node
    keys[i] = y->keys[t-1];

    // Increment count of keys in this node
    n = n + 1;
}

// Function to traverse all nodes in a subtree rooted with this node
void BTreeNode::traverse()
{
    // There are n keys and n+1 children, travers through n keys
    // and first n children
    int i;
    for (i = 0; i < n; i++)
    {
        // If this is not leaf, then before printing key[i],
        // traverse the subtree rooted with child C[i].
        if (leaf == false)
            C[i]->traverse();
        cout << " " << keys[i];
    }

    // Print the subtree rooted with last child
    if (leaf == false)
        C[i]->traverse();
}

// Function to search key k in subtree rooted with this node
BTreeNode *BTreeNode::search(int k)
{
    // Find the first key greater than or equal to k
    int i = 0;
    while (i < n && k > keys[i])
        i++;

    // If the found key is equal to k, return this node
    if (keys[i] == k)
        return this;

    // If key is not found here and this is a leaf node
    if (leaf == true)
        return NULL;

    // Go to the appropriate child
    return C[i]->search(k);
}

void BTree::remove(int k)
{
    if (root)
    {
        cout << "The tree is empty\n";
        return;
    }

    // Call the remove function for root
    root->remove(k);

    // If the root node has 0 keys, make its first child as the new root
    // if it has a child, otherwise set root as NULL
    if (root->n==0)
    {
        BTreeNode *tmp = root;
        if (root->leaf)
            root = NULL;
        else
            root = root->C[0];

        // Free the old root
        delete tmp;
    }
    return;
}

// Driver program to test above functions
int main()
{
    BTree t(3); // A B-Tree with minium degree 3

    t.insert(1);
    t.insert(3);
    t.insert(7);
}

```

```

t.insert(10);
t.insert(11);
t.insert(13);
t.insert(14);
t.insert(15);
t.insert(18);
t.insert(16);
t.insert(19);
t.insert(24);
t.insert(25);
t.insert(26);
t.insert(21);
t.insert(4);
t.insert(5);
t.insert(20);
t.insert(22);
t.insert(2);
t.insert(17);
t.insert(12);
t.insert(6);

cout << "Traversal of tree constructed is\n";
t.traverse();
cout << endl;

t.remove(6);
cout << "Traversal of tree after removing 6\n";
t.traverse();
cout << endl;

t.remove(13);
cout << "Traversal of tree after removing 13\n";
t.traverse();
cout << endl;

t.remove(7);
cout << "Traversal of tree after removing 7\n";
t.traverse();
cout << endl;

t.remove(4);
cout << "Traversal of tree after removing 4\n";
t.traverse();
cout << endl;

t.remove(2);
cout << "Traversal of tree after removing 2\n";
t.traverse();
cout << endl;

t.remove(16);
cout << "Traversal of tree after removing 16\n";
t.traverse();
cout << endl;

return 0;
}

```

Output:

```

Traversal of tree constructed is
1 2 3 4 5 6 7 10 11 12 13 14 15 16 17 18 19 20 21 22 24 25 26
Traversal of tree after removing 6
1 2 3 4 5 7 10 11 12 13 14 15 16 17 18 19 20 21 22 24 25 26
Traversal of tree after removing 13
1 2 3 4 5 7 10 11 12 14 15 16 17 18 19 20 21 22 24 25 26
Traversal of tree after removing 7
1 2 3 4 5 10 11 12 14 15 16 17 18 19 20 21 22 24 25 26
Traversal of tree after removing 4
1 2 3 5 10 11 12 14 15 16 17 18 19 20 21 22 24 25 26
Traversal of tree after removing 2
1 3 5 10 11 12 14 15 16 17 18 19 20 21 22 24 25 26
Traversal of tree after removing 16
1 3 5 10 11 12 14 15 17 18 19 20 21 22 24 25 26

```

This article is contributed by [Balasubramanian.N](#) . Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Advance Data Structures
Advanced Data Structures

Red-Black Tree | Set 1 (Introduction)

Red-Black Tree is a self-balancing Binary Search Tree (BST) where every node follows following rules.

RedBlackTree



- 1) Every node has a color either red or black.
- 2) Root of tree is always black.
- 3) There are no two adjacent red nodes (A red node cannot have a red parent or red child).
- 4) Every path from root to a NULL node has same number of black nodes.

Why Red-Black Trees?

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take $O(h)$ time where h is the height of the BST. The cost of these operations may become $O(n)$ for a skewed Binary tree. If we make sure that height of the tree remains $O(\log n)$ after every insertion and deletion, then we can guarantee an upper bound of $O(\log n)$ for all these operations. The height of a Red Black tree is always $O(\log n)$ where n is the number of nodes in the tree.

Comparison with AVL Tree

The AVL trees are more balanced compared to Red Black Trees, but they may cause more rotations during insertion and deletion. So if your application involves many frequent insertions and deletions, then Red Black trees should be preferred. And if the insertions and deletions are less frequent and search is more frequent operation, then AVL tree should be preferred over Red Black Tree.

How does a Red-Black Tree ensure balance?

A simple example to understand balancing is, a chain of 3 nodes is not possible in red black tree. We can try any combination of colors and see all of them violate Red-Black tree property.

A chain of 3 nodes is not possible in Red-Black Trees.

Following are **NOT** Red-Black Trees



Following are different possible Red-Black Trees with above 3 keys



From the above examples, we get some idea how Red-Black trees ensure balance. Following is an important fact about balancing in Red-Black Trees.

Every Red Black Tree with n nodes has height $\leq 2\log_2(n+1)$

This can be proved using following facts:

- 1) For a general Binary Tree, let k be the minimum number of nodes on all root to NULL paths, then $n \geq 2^k - 1$ (Ex. If k is 3, then n is atleast 7). This expression can also be written as $k \leq 2\log_2(n+1)$
- 2) From property 4 of Red-Black trees and above claim, we can say in a Red-Black Tree with n nodes, there is a root to leaf path with at-most $\log_2(n+1)$ black nodes.
- 3) From property 3 of Red-Black trees, we can claim that the number black nodes in a Red-Black tree is at least $\lfloor n/2 \rfloor$ where n is total number of nodes.

From above 2 points, we can conclude the fact that Red Black Tree with n nodes has height $\leq 2\log_2(n+1)$

In this post, we introduced Red-Black trees and discussed how balance is ensured. The hard part is to maintain balance when keys are added and removed. We will soon be discussing insertion and deletion operations in coming posts on Red-Black tree.

Exercise:

- 1) Is it possible to have all black nodes in a Red-Black tree?
- 2) Draw a Red-Black Tree that is not an AVL tree structure wise?

Insertion and Deletion

[Red Black Tree Insertion](#)

[Red-Black Tree Deletion](#)

References:

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

http://en.wikipedia.org/wiki/Red%E2%80%93black_tree

Video Lecture on Red-Black Tree by Tim Roughgarden

MIT Video Lecture on Red-Black Tree

MIT Lecture Notes on Red Black Tree

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Advance Data Structures
Advanced Data Structures
Self-Balancing BST

Red-Black Tree | Set 2 (Insert)

In the [previous post](#), we discussed introduction to Red-Black Trees. In this post, insertion is discussed.

In [AVL tree insertion](#), we used rotation as a tool to do balancing after insertion caused imbalance. In Red-Black tree, we use two tools to do balancing.

- 1) Recoloring
- 2) Rotation

We try recoloring first, if recoloring doesn't work, then we go for rotation. Following is detailed algorithm. The algorithms has mainly two cases depending upon the color of uncle. If uncle is red, we do recoloring. If uncle is black, we do rotations and/or recoloring.

Color of a NULL node is considered as BLACK.

Let x be the newly inserted node.

- 1) Perform [standard BST insertion](#) and make the color of newly inserted nodes as RED.

- 2) If x is root, change color of x as BLACK (Black height of complete tree increases by 1).

- 3) Do following if color of x 's parent is not BLACK or x is not root.

.....a) If x 's uncle is RED (Grand parent must have been black from [property 4](#))

.....(i) Change color of parent and uncle as BLACK.

.....(ii) color of grand parent as RED.

.....(iii) Change $x = x$'s grandparent, repeat steps 2 and 3 for new x .

[redBlackCase2](#)



.....b) If x 's uncle is BLACK, then there can be four configurations for x , x 's parent (p) and x 's grandparent (g) (This is similar to [AVL Tree](#))

.....i) Left Left Case (p is left child of g and x is left child of p)

.....ii) Left Right Case (p is left child of g and x is right child of p)

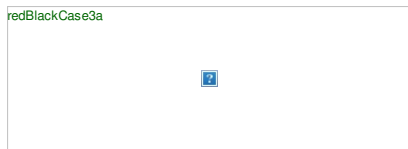
.....iii) Right Right Case (Mirror of case a)

.....iv) Right Left Case (Mirror of case c)

Following are operations to be performed in four subcases when uncle is BLACK.

All four cases when Uncle is BLACK

Left Left Case (See g , p and x)



Left Right Case (See g, p and x)



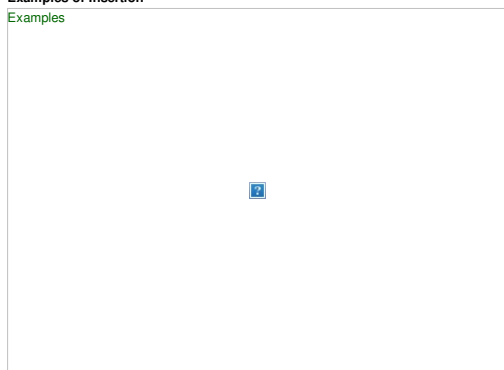
Right Right Case (See g, p and x)



Right Left Case (See g, p and x)



Examples of Insertion



Exercise:

Insert 2, 6 and 13 in below tree.



Please refer [C Program for Red Black Tree Insertion](#) for complete implementation of above algorithm.

[Red-Black Tree | Set 3 \(Delete\)](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

[Advanced Data Structure](#)
[Advance Data Structures](#)
[Advanced Data Structures](#)
[Self-Balancing-BST](#)

Red-Black Tree | Set 3 (Delete)

We have discussed following topics on Red-Black tree in previous posts. We strongly recommend to refer following post as prerequisite of this post.

[Red-Black Tree Introduction](#)

[Red Black Tree Insert](#)

Insertion Vs Deletion:

Like Insertion, recoloring and rotations are used to maintain the Red-Black properties.

In insert operation, we check color of uncle to decide the appropriate case. In delete operation, ***we check color of sibling*** to decide the appropriate case.

The main property that violates after insertion is two consecutive reds. In delete, the main violated property is, change of black height in subtrees as deletion of a black node may cause reduced black height in one root to leaf path.

Deletion is fairly complex process. To understand deletion, notion of double black is used. When a black node is deleted and replaced by a black child, the child is marked as ***double black***. The main task now becomes to

convert this double black to single black.

Deletion Steps

Following are detailed steps for deletion.

1) Perform **standard BST delete**. When we perform standard delete operation in BST, we always end up deleting a node which is either leaf or has only one child (For an internal node, we copy the successor and then recursively call delete for successor, successor is always a leaf node or a node with one child). So we only need to handle cases where a node is leaf or has one child. Let v be the node to be deleted and u be the child that replaces v (Note that u is NULL when v is a leaf and color of NULL is considered as Black).

2) **Simple Case: If either u or v is red**, we mark the replaced child as black (No change in black height). Note that both u and v cannot be red as v is parent of u and two consecutive reds are not allowed in red-black tree.

rbdelete11



3) If Both u and v are Black.

3.1) Color u as double black. Now our task reduces to convert this double black to single black. Note that If v is leaf, then u is NULL and color of NULL is considered as black. So the deletion of a black leaf also causes a double black.

rbdelete12_new



3.2) Do following while the current node u is double black or it is not root. Let sibling of node be s .

.....(a): If sibling s is black and at least one of sibling's children is red, perform rotation(s). Let the red child of s be r . This case can be divided in four subcases depending upon positions of s and r .

.....(i) Left Left Case (s is left child of its parent and r is left child of s or both children of s are red). This is mirror of right right case shown in below diagram.

.....(ii) Left Right Case (s is left child of its parent and r is right child). This is mirror of right left case shown in below diagram.

.....(iii) Right Right Case (s is right child of its parent and r is right child of s or both children of s are red)

rbdelete13New



.....(iv) Right Left Case (s is right child of its parent and r is left child of s)

rbdelete14



.....(b): If sibling is black and its both children are black, perform recoloring, and recur for the parent if parent is black.

rbdelete15

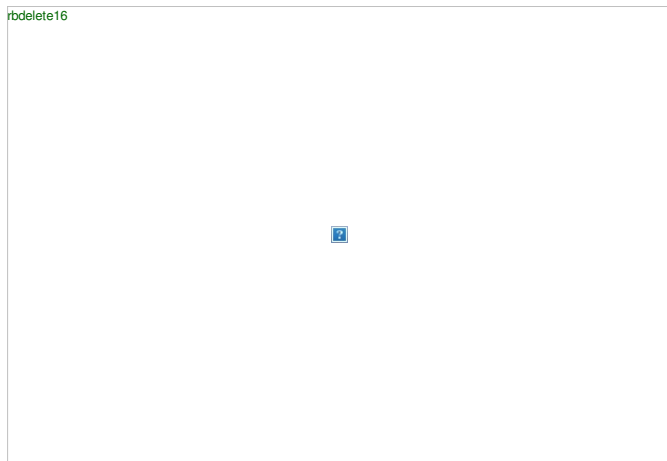


In this case, if parent was red, then we didn't need to recur for parent, we can simply make it black (red + double black = single black)

.....(c): If sibling is red, perform a rotation to move old sibling up, recolor the old sibling and parent. The new sibling is always black (See the below diagram). This mainly converts the tree to black sibling case (by rotation) and leads to case (a) or (b). This case can be divided in two subcases.

.....(i) Left Case (s is left child of its parent). This is mirror of right right case shown in below diagram. We right rotate the parent p .

.....(iii) Right Case (s is right child of its parent). We left rotate the parent p .



3.3) If u is root, make it single black and return (Black height of complete tree reduces by 1).

References:

<https://www.cs.purdue.edu/homes/ayg/CS251/slides/chap13c.pdf>

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Advance Data Structures
Advanced Data Structures

C Program for Red Black Tree Insertion

Following article is extension of article discussed [here](#).

In [AVL tree insertion](#), we used rotation as a tool to do balancing after insertion caused imbalance. In Red-Black tree, we use two tools to do balancing.

1) Recoloring

2) Rotation

We try recoloring first, if recoloring doesn't work, then we go for rotation. Following is detailed algorithm. The algorithm has mainly two cases depending upon the color of uncle. If uncle is red, we do recoloring. If uncle is black, we do rotations and/or recoloring.

Color of a NULL node is considered as BLACK.

Let x be the newly inserted node.

1) Perform [standard BST insertion](#) and make the color of newly inserted nodes as RED.

2) If x is root, change color of x as BLACK (Black height of complete tree increases by 1).

3) Do following if color of x's parent is not BLACK or x is not root.

.....a) **If x's uncle is RED** (Grand parent must have been black from [property 4](#))

.....(i) Change color of parent and uncle as BLACK.

.....(ii) color of grand parent as RED.

.....(iii) Change x = x's grandparent, repeat steps 2 and 3 for new x.

[redBlackCase2](#)



.....b) **If x's uncle is BLACK**, then there can be four configurations for x, x's parent (p) and x's grandparent (g) (This is similar to [AVL Tree](#))

.....i) Left Left Case (p is left child of g and x is left child of p)

.....ii) Left Right Case (p is left child of g and x is right child of p)

.....iii) Right Right Case (Mirror of case a)

.....iv) Right Left Case (Mirror of case c)

Following are operations to be performed in four subcases when uncle is BLACK.

All four cases when Uncle is BLACK

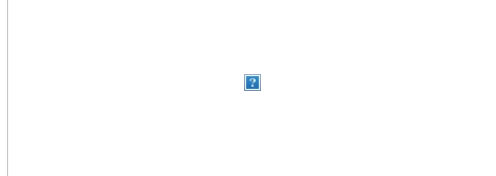
Left Left Case (See g, p and x)

[redBlackCase3a](#)



Left Right Case (See g, p and x)

[redBlackCase3b](#)



Right Right Case (See g, p and x)

redBlackCase3c

A diagram showing a single node with the value 2 inside a blue square box.

Right Left Case (See g, p and x)

redBlackCase3d

A diagram showing a single node with the value 4 inside a blue square box.

Examples of Insertion

Examples

A diagram showing a single node with the value 2 inside a blue square box.

Below is C++ Code.

```
/** C++ implementation for Red-Black Tree Insertion
 * This code is adopted from the code provided by
 * Dinesh Khandelwal in comments */
#include <bits/stdc++.h>
using namespace std;

enum Color {RED, BLACK};

struct Node
{
    int data;
    bool color;
    Node *left, *right, *parent;

    // Constructor
    Node(int data)
    {
        this->data = data;
        left = right = parent = NULL;
    }
};

// Class to represent Red-Black Tree
class RBTree
{
private:
    Node *root;
protected:
    void rotateLeft(Node *&, Node *&);
    void rotateRight(Node *&, Node *&);
    void fixViolation(Node *&, Node *&);
public:
    // Constructor
    RBTree() { root = NULL; }
    void insert(const int &n);
    void inorder();
    void levelOrder();
};

// A recursive function to do level order traversal
void inorderHelper(Node *root)
{
    if (root == NULL)
        return;

    inorderHelper(root->left);
    cout << root->data << " ";
    inorderHelper(root->right);
}

// A utility function to insert a new node with given key
// in BST */
Node* BSTInsert(Node* root, Node *pt)
{
    /* If the tree is empty, return a new node */
    if (root == NULL)
        return pt;

    /* Otherwise, recur down the tree */
    if (pt->data < root->data)
    {
        root->left = BSTInsert(root->left, pt);
        root->left->parent = root;
    }
    else if (pt->data > root->data)
```



```

{
    root->right = BSTInsert(root->right, pt);
    root->right->parent = root;
}

/* return the (unchanged) node pointer */
return root;
}

// Utility function to do level order traversal
void levelOrderHelper(Node *root)
{
    if (root == NULL)
        return;

    std::queue<Node *> q;
    q.push(root);

    while (!q.empty())
    {
        Node *temp = q.front();
        cout << temp->data << " ";
        q.pop();

        if (temp->left != NULL)
            q.push(temp->left);

        if (temp->right != NULL)
            q.push(temp->right);
    }
}

void RBTree::rotateLeft(Node *&root, Node *&pt)
{
    Node *pt_right = pt->right;

    pt->right = pt_right->left;

    if (pt->right != NULL)
        pt->right->parent = pt;

    pt_right->parent = pt->parent;

    if (pt->parent == NULL)
        root = pt_right;

    else if (pt == pt->parent->left)
        pt->parent->left = pt_right;

    else
        pt->parent->right = pt_right;

    pt_right->left = pt;
    pt->parent = pt_right;
}

void RBTree::rotateRight(Node *&root, Node *&pt)
{
    Node *pt_left = pt->left;

    pt->left = pt_left->right;

    if (pt->left != NULL)
        pt->left->parent = pt;

    pt_left->parent = pt->parent;

    if (pt->parent == NULL)
        root = pt_left;

    else if (pt == pt->parent->right)
        pt->parent->right = pt_left;

    else
        pt->parent->left = pt_left;

    pt_left->right = pt;
    pt->parent = pt_left;
}

// This function fixes violations caused by BST insertion
void RBTree::fixViolation(Node *&root, Node *&pt)
{
    Node *parent_pt = NULL;
    Node *grand_parent_pt = NULL;

    while ((pt != root) && (pt->color != BLACK) &&
           (pt->parent->color == RED))
    {
        parent_pt = pt->parent;
        grand_parent_pt = pt->parent->parent;

        /* Case : A
           Parent of pt is left child of Grand-parent of pt */
        if (parent_pt == grand_parent_pt->left)
        {
            Node *uncle_pt = grand_parent_pt->right;

            /* Case : 1
               The uncle of pt is also red
               Only Recoloring required */
            if (uncle_pt != NULL && uncle_pt->color == RED)
            {
                grand_parent_pt->color = RED;
                parent_pt->color = BLACK;
                uncle_pt->color = BLACK;
                pt = grand_parent_pt;
            }

            else
            {
                /* Case : 2
                   pt is right child of its parent
                   Left-rotation required */

```

```

        if (pt == parent_pt->right)
        {
            rotateLeft(root, parent_pt);
            pt = parent_pt;
            parent_pt = pt->parent;
        }

        /* Case : 3
        pt is left child of its parent
        Right-rotation required */
        rotateRight(root, grand_parent_pt);
        swap(parent_pt->color, grand_parent_pt->color);
        pt = parent_pt;
    }
}

/* Case : B
Parent of pt is right child of Grand-parent of pt */
else
{
    Node *uncle_pt = grand_parent_pt->left;

    /* Case : 1
    The uncle of pt is also red
    Only Recoloring required */
    if ((uncle_pt != NULL) && (uncle_pt->color == RED))
    {
        grand_parent_pt->color = RED;
        parent_pt->color = BLACK;
        uncle_pt->color = BLACK;
        pt = grand_parent_pt;
    }
    else
    {
        /* Case : 2
        pt is left child of its parent
        Right-rotation required */
        if (pt == parent_pt->left)
        {
            rotateRight(root, parent_pt);
            pt = parent_pt;
            parent_pt = pt->parent;
        }

        /* Case : 3
        pt is right child of its parent
        Left-rotation required */
        rotateLeft(root, grand_parent_pt);
        swap(parent_pt->color, grand_parent_pt->color);
        pt = parent_pt;
    }
}

root->color = BLACK;
}

// Function to insert a new node with given data
void RBTree::insert(const int &data)
{
    Node *pt = new Node(data);

    // Do a normal BST insert
    root = BSTInsert(root, pt);

    // fix Red Black Tree violations
    fixViolation(root, pt);
}

// Function to do inorder and level order traversals
void RBTree::inorder() { inorderHelper(root); }
void RBTree::levelOrder() { levelOrderHelper(root); }

// Driver Code
int main()
{
    RBTree tree;

    tree.insert(7);
    tree.insert(6);
    tree.insert(5);
    tree.insert(4);
    tree.insert(3);
    tree.insert(2);
    tree.insert(1);

    cout << "Inoder Traversal of Created Tree\n";
    tree.inorder();

    cout << "\n\nLevel Order Traversal of Created Tree\n";
    tree.levelOrder();

    return 0;
}

```

Output:

```

Inoder Traversal of Created Tree
1 2 3 4 5 6 7

```

```

Level Order Traversal of Created Tree
6 4 7 2 5 1 3

```

This article is contributed by **Mohsin Mohammad**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: Programs Tree

K Dimensional Tree | Set 1 (Search and Insert)

A K-D Tree(also called as K-Dimensional Tree) is a binary search tree where data in each node is a K-Dimensional point in space. In short, it is a space partitioning(details below) data structure for organizing points in a K-Dimensional space.

A non-leaf node in K-D tree divides the space into two parts, called as half-spaces.

Points to the left of this space are represented by the left subtree of that node and points to the right of the space are represented by the right subtree. We will soon be explaining the concept on how the space is divided and tree is formed.

For the sake of simplicity, let us understand a 2-D Tree with an example.

The root would have an x-aligned plane, the root's children would both have y-aligned planes, the root's grandchildren would all have x-aligned planes, and the root's great-grandchildren would all have y-aligned planes and so on.

Generalization:

Let us number the planes as 0, 1, 2, ... $(K - 1)$. From the above example, it is quite clear that a point (node) at depth D will have A aligned plane where A is calculated as:

$$A = D \bmod K$$

How to determine if a point will lie in the left subtree or in right subtree?

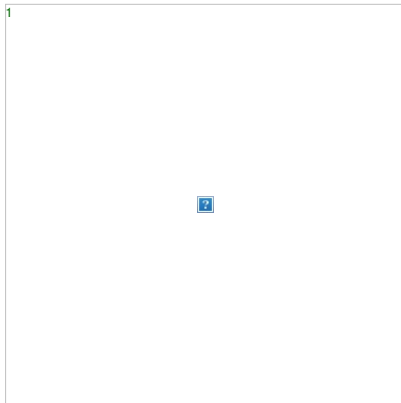
If the root node is aligned in planeA, then the left subtree will contain all points whose coordinates in that plane are smaller than that of root node. Similarly, the right subtree will contain all points whose coordinates in that plane are greater-equal to that of root node.

Creation of a 2-D Tree:

Consider following points in a 2-D plane:

(3, 6), (17, 15), (13, 15), (6, 12), (9, 1), (2, 7), (10, 19)

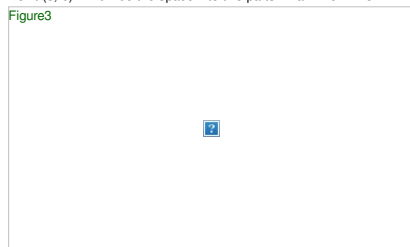
1. Insert (3, 6): Since tree is empty, make it the root node.
2. Insert (17, 15): Compare it with root node point. Since root node is X-aligned, the X-coordinate value will be compared to determine if it lies in the left subtree or in the right subtree. This point will be Y-aligned.
3. Insert (13, 15): X-value of this point is greater than X-value of point in root node. So, this will lie in the right subtree of (3, 6). Again Compare Y-value of this point with the Y-value of point (17, 15) (Why?). Since, they are equal, this point will lie in the right subtree of (17, 15). This point will be X-aligned.
4. Insert (6, 12): X-value of this point is greater than X-value of point in root node. So, this will lie in the right subtree of (3, 6). Again Compare Y-value of this point with the Y-value of point (17, 15) (Why?). Since, $12 < 15$, this point will lie in the left subtree of (17, 15). This point will be X-aligned.
5. Insert (9, 1): Similarly, this point will lie in the right of (6, 12).
6. Insert (2, 7): Similarly, this point will lie in the left of (3, 6).
7. Insert (10, 19): Similarly, this point will lie in the right of (13, 15).



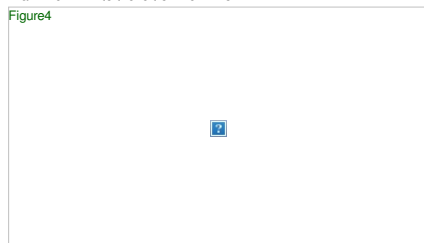
How is space partitioned?

All 7 points will be plotted in the X-Y plane as follows:

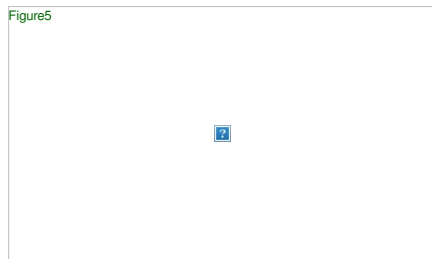
1. Point (3, 6) will divide the space into two parts: Draw line $X = 3$.



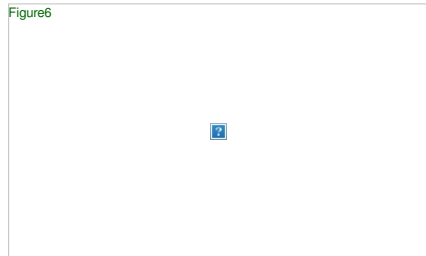
2. Point (2, 7) will divide the space to the left of line $X = 3$ into two parts horizontally. Draw line $Y = 7$ to the left of line $X = 3$.



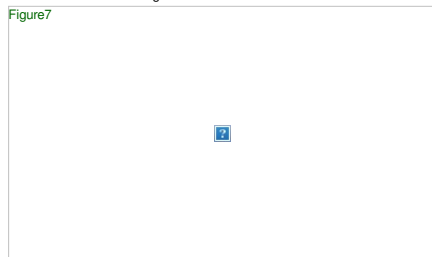
3. Point (17, 15) will divide the space to the right of line $X = 3$ into two parts horizontally. Draw line $Y = 15$ to the right of line $X = 3$.



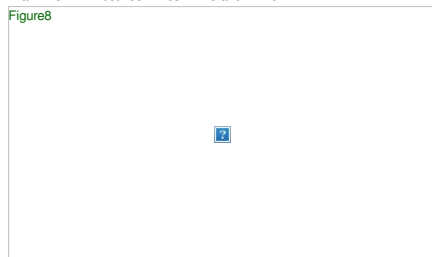
- Point (6, 12) will divide the space below line $Y = 15$ and to the right of line $X = 3$ into two parts.
Draw line $X = 6$ to the right of line $X = 3$ and below line $Y = 15$.



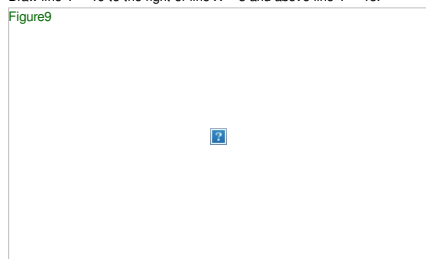
- Point (13, 15) will divide the space below line $Y = 15$ and to the right of line $X = 6$ into two parts.
Draw line $X = 13$ to the right of line $X = 6$ and below line $Y = 15$.



- Point (9, 1) will divide the space between lines $X = 3$, $X = 6$ and $Y = 15$ into two parts.
Draw line $Y = 1$ between lines $X = 3$ and $X = 6$.



- Point (10, 19) will divide the space to the right of line $X = 3$ and above line $Y = 15$ into two parts.
Draw line $Y = 19$ to the right of line $X = 3$ and above line $Y = 15$.



Following is C++ implementation of KD Tree basic operations like search, insert and delete.

```
// A C++ program to demonstrate operations of KD tree
#include<bits/stdc++.h>
using namespace std;

const int k = 2;

// A structure to represent node of kd tree
struct Node
{
    int point[k]; // To store k dimensional point
    Node *left, *right;
};

// A method to create a node of K D tree
struct Node* newNode(int arr[])
{
    struct Node* temp = new Node;

    for (int i=0; i<k; i++)
        temp->point[i] = arr[i];

    temp->left = temp->right = NULL;
    return temp;
}
```

```

}

// Inserts a new node and returns root of modified tree
// The parameter depth is used to decide axis of comparison
Node *insertRec(Node *root, int point[], unsigned depth)
{
    // Tree is empty?
    if (root == NULL)
        return newNode(point);

    // Calculate current dimension (cd) of comparison
    unsigned cd = depth % k;

    // Compare the new point with root on current dimension 'cd'
    // and decide the left or right subtree
    if (point[cd] < (root->point[cd]))
        root->left = insertRec(root->left, point, depth + 1);
    else
        root->right = insertRec(root->right, point, depth + 1);

    return root;
}

// Function to insert a new point with given point in
// KD Tree and return new root. It mainly uses above recursive
// function "insertRec()"
Node* insert(Node *root, int point[])
{
    return insertRec(root, point, 0);
}

// A utility method to determine if two Points are same
// in K Dimensional space
bool arePointsSame(int point1[], int point2[])
{
    // Compare individual pointinate values
    for (int i = 0; i < k; ++i)
        if (point1[i] != point2[i])
            return false;

    return true;
}

// Searches a Point represented by "point[]" in the K D tree.
// The parameter depth is used to determine current axis.
bool searchRec(Node* root, int point[], unsigned depth)
{
    // Base cases
    if (root == NULL)
        return false;
    if (arePointsSame(root->point, point))
        return true;

    // Current dimension is computed using current depth and total
    // dimensions (k)
    unsigned cd = depth % k;

    // Compare point with root with respect to cd (Current dimension)
    if (point[cd] < root->point[cd])
        return searchRec(root->left, point, depth + 1);

    return searchRec(root->right, point, depth + 1);
}

// Searches a Point in the K D tree. It mainly uses
// searchRec()
bool search(Node* root, int point[])
{
    // Pass current depth as 0
    return searchRec(root, point, 0);
}

// Driver program to test above functions
int main()
{
    struct Node *root = NULL;
    int points[][k] = {{3, 6}, {17, 15}, {13, 15}, {6, 12},
                      {9, 1}, {2, 7}, {10, 19}};

    int n = sizeof(points)/sizeof(points[0]);

    for (int i=0; i<n; i++)
        root = insert(root, points[i]);

    int point1[] = {10, 19};
    (search(root, point1))? cout << "Found\n": cout << "Not Found\n";

    int point2[] = {12, 19};
    (search(root, point2))? cout << "Found\n": cout << "Not Found\n";

    return 0;
}

```

Output:

```

Found
Not Found

```

Refer below articles for find minimum and delete operations.

- [K D Tree \(Find Minimum\)](#)
- [K D Tree \(Delete\)](#)

This article is compiled by **Aashish Barnwal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

K Dimensional Tree | Set 2 (Find Minimum)

We strongly recommend to refer below post as a prerequisite of this.

[K Dimensional Tree | Set 1 \(Search and Insert\)](#)

In this post find minimum is discussed. The operation is to find minimum in the given dimension. This is especially needed in delete operation.

For example, consider below KD Tree, if given dimension is x, then output should be 5 and if given dimensions is 1, then output should be 12. Below image is taken from [this](#) source.

kdtrenew



In KD tree, points are divided dimension by dimension. For example, root divides keys by dimension 0, level next to root divides by dimension 1, next level by dimension 2 if k is more than 2 (else by dimension 0), and so on.

To find minimum we traverse nodes starting from root. **If dimension of current level is same as given dimension, then required minimum lies on left side if there is left child.** This is same as [Binary Search Tree Minimum](#).

Above is simple, what to do when current level's dimension is different. **When dimension of current level is different, minimum may be either in left subtree or right subtree or current node may also be minimum.** So we take minimum of three and return. This is different from Binary Search tree.

Below is C++ implementation of find minimum operation.

```
// A C++ program to demonstrate find minimum on KD tree
#include<bits/stdc++.h>
using namespace std;

const int k = 2;

// A structure to represent node of kd tree
struct Node
{
    int point[k]; // To store k dimensional point
    Node *left, *right;
};

// A method to create a node of K D tree
struct Node* newNode(int arr[])
{
    struct Node* temp = new Node;

    for (int i=0; i<k; i++)
        temp->point[i] = arr[i];

    temp->left = temp->right = NULL;
    return temp;
}

// Inserts a new node and returns root of modified tree
// The parameter depth is used to decide axis of comparison
Node *insertRec(Node *root, int point[], unsigned depth)
{
    // Tree is empty?
    if (root == NULL)
        return newNode(point);

    // Calculate current dimension (cd) of comparison
    unsigned cd = depth % k;

    // Compare the new point with root on current dimension 'cd'
    // and decide the left or right subtree
    if (point[cd] < (root->point[cd]))
        root->left = insertRec(root->left, point, depth + 1);
    else
        root->right = insertRec(root->right, point, depth + 1);

    return root;
}

// Function to insert a new point with given point in
// KD Tree and return new root. It mainly uses above recursive
// function "insertRec()"
Node* insert(Node *root, int point[])
{
    return insertRec(root, point, 0);
}

// A utility function to find minimum of three integers
int min(int x, int y, int z)
{
    return min(x, min(y, z));
}

// Recursively finds minimum of d'th dimension in KD tree
// The parameter depth is used to determine current axis.
int findMinRec(Node* root, int d, unsigned depth)
{
    // Base cases
    if (root == NULL)
        return INT_MAX;

    // Current dimension is computed using current depth and total
    // dimensions (k)
    unsigned cd = depth % k;

    // Compare point with root with respect to cd (Current dimension)
    if (cd == d)
    {
        if (root->left == NULL)
            return root->point[d];
    }
```

```

    return findMinRec(root->left, d, depth+1);
}

// If current dimension is different then minimum can be anywhere
// in this subtree
return min(root->point[d],
    findMinRec(root->left, d, depth+1),
    findMinRec(root->right, d, depth+1));
}

// A wrapper over findMinRec(). Returns minimum of d'th dimension
int findMin(Node* root, int d)
{
    // Pass current level or depth as 0
    return findMinRec(root, d, 0);
}

// Driver program to test above functions
int main()
{
    struct Node *root = NULL;
    int points[][k] = {{30, 40}, {5, 25}, {70, 70},
        {10, 12}, {50, 30}, {35, 45}};

    int n = sizeof(points)/sizeof(points[0]);

    for (int i=0; i<n; i++)
        root = insert(root, points[i]);

    cout << "Minimum of 0'th dimension is " << findMin(root, 0) << endl;
    cout << "Minimum of 1'th dimension is " << findMin(root, 1) << endl;

    return 0;
}

```

Output:

```

Minimum of 0'th dimension is 5
Minimum of 1'th dimension is 12

```

Source:

<https://www.cs.umd.edu/class/spring2008/cmsc420/L19.kd-trees.pdf>

This article is contributed by **Ashish Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Advanced Data Structures

K Dimensional Tree | Set 3 (Delete)

We strongly recommend to refer below posts as a prerequisite of this.

[K Dimensional Tree | Set 1 \(Search and Insert\)](#)

[K Dimensional Tree | Set 2 \(Find Minimum\)](#)

In this post delete is discussed. The operation is to delete a given point from K D Tree.

Like [Binary Search Tree Delete](#), we recursively traverse down and search for the point to be deleted. Below are steps are followed for every node visited.

1) If current node contains the point to be deleted

- If node to be deleted is a leaf node, simply delete it (Same as [BST Delete](#))
- If node to be deleted has right child as not NULL (Different from BST)
 - Find minimum of current node's dimension in right subtree.
 - Replace the node with above found minimum and recursively delete minimum in right subtree.
- Else If node to be deleted has left child as not NULL (Different from BST)
 - Find minimum of current node's dimension in left subtree.
 - Replace the node with above found minimum and recursively delete minimum in left subtree.
 - Make new left subtree as right child of current node.

2) If current doesn't contain the point to be deleted

- If node to be deleted is smaller than current node on current dimension, recur for left subtree.
- Else recur for right subtree.

Why 1.b and 1.c are different from BST?

In BST delete, if a node's left child is empty and right is not empty, we replace the node with right child. In K D Tree, doing this would violate the KD tree property as dimension of right child of node is different from node's dimension. For example, if node divides point by x axis values. then its children divide by y axis, so we can't simply replace node with right child. Same is true for the case when right child is not empty and left child is empty.

Why 1.c doesn't find max in left subtree and recur for max like 1.b?

Doing this violates the property that all equal values are in right subtree. For example, if we delete (10, 10) in below subtree and replace it with

```

Wrong Way (Equal key in left subtree after deletion)
      (5, 6)          (4, 10)
       /             /
      (4, 10) -----> (4, 20)
       \
      (4, 20)

Right way (Equal key in right subtree after deletion)
      (5, 6)          (4, 10)
       /             / \
      (4, 10) -----> (4, 20)
       \
      (4, 20)

```

Example of Delete:

Delete (30, 40): Since right child is not NULL and dimension of node is x, we find the node with minimum x value in right child. The node is (35, 45), we replace (30, 40) with (35, 45) and delete (35, 45).

kdtreedelete2



Delete (70, 70): Dimension of node is y. Since right child is NULL, we find the node with minimum y value in left child. The node is (50, 30), we replace (70, 70) with (50, 30) and recursively delete (50, 30) in left subtree. Finally we make the modified left subtree as right subtree of (50, 30).

kdtreedelete



Below is C++ implementation of K D Tree delete.

```
// A C++ program to demonstrate delete in K D tree
#include <bits/stdc++.h>
using namespace std;

const int k = 2;

// A structure to represent node of kd tree
struct Node
{
    int point[k]; // To store k dimensional point
    Node *left, *right;
};

// A method to create a node of K D tree
struct Node* newNode(int arr[])
{
    struct Node* temp = new Node;

    for (int i=0; i<k; i++)
        temp->point[i] = arr[i];

    temp->left = temp->right = NULL;
    return temp;
}

// Inserts a new node and returns root of modified tree
// The parameter depth is used to decide axis of comparison
Node *insertRec(Node *root, int point[], unsigned depth)
{
    // Tree is empty?
    if (root == NULL)
        return newNode(point);

    // Calculate current dimension (cd) of comparison
    unsigned cd = depth % k;

    // Compare the new point with root on current dimension 'cd'
    // and decide the left or right subtree
    if (point[cd] < (root->point[cd]))
        root->left = insertRec(root->left, point, depth + 1);
    else
        root->right = insertRec(root->right, point, depth + 1);

    return root;
}

// Function to insert a new point with given point in
// KD Tree and return new root. It mainly uses above recursive
// function "insertRec()"
Node* insert(Node *root, int point[])
{
    return insertRec(root, point, 0);
}

// A utility function to find minimum of three integers
Node *minNode(Node *x, Node *y, Node *z, int d)
{
    Node *res = x;
    if (y != NULL && y->point[d] < res->point[d])
        res = y;
    if (z != NULL && z->point[d] < res->point[d])
        res = z;
    return res;
}

// Recursively finds minimum of d'th dimension in KD tree
// The parameter depth is used to determine current axis.
Node *findMinRec(Node* root, int d, unsigned depth)
{
    // Base cases
    if (root == NULL)
        return NULL;

    // Current dimension is computed using current depth and total
    // dimensions (k)
    unsigned cd = depth % k;

    // Compare point with root with respect to cd (Current dimension)
    if (cd == d)
    {
        if (root->left == NULL)
            return root;
        return findMinRec(root->left, d, depth+1);
    }

    // If current dimension is different then minimum can be anywhere
    // in this subtree
    return minNode(root,
```



```

        findMinRec(root->left, d, depth+1),
        findMinRec(root->right, d, depth+1), d);
}

// A wrapper over findMinRec(). Returns minimum of d'th dimension
Node *findMin(Node *root, int d)
{
    // Pass current level or depth as 0
    return findMinRec(root, d, 0);
}

// A utility method to determine if two Points are same
// in K Dimensional space
bool arePointsSame(int point1[], int point2[])
{
    // Compare individual pointinate values
    for (int i = 0; i < k; i++)
        if (point1[i] != point2[i])
            return false;

    return true;
}

// Copies point p2 to p1
void copyPoint(int p1[], int p2[])
{
    for (int i=0; i<k; i++)
        p1[i] = p2[i];
}

// Function to delete a given point 'point[]' from tree with root
// as 'root'. depth is current depth and passed as 0 initially.
// Returns root of the modified tree.
Node *deleteNodeRec(Node *root, int point[], int depth)
{
    // Given point is not present
    if (root == NULL)
        return NULL;

    // Find dimension of current node
    int cd = depth % k;

    // If the point to be deleted is present at root
    if (arePointsSame(root->point, point))
    {
        // 2.b) If right child is not NULL
        if (root->right != NULL)
        {
            // Find minimum of root's dimension in right subtree
            Node *min = findMin(root->right, cd);

            // Copy the minimum to root
            copyPoint(root->point, min->point);

            // Recursively delete the minimum
            root->right = deleteNodeRec(root->right, min->point, depth+1);
        }
        else if (root->left != NULL) // same as above
        {
            Node *min = findMin(root->left, cd);
            copyPoint(root->point, min->point);
            root->right = deleteNodeRec(root->left, min->point, depth+1);
        }
        else // If node to be deleted is leaf node
        {
            delete root;
            return NULL;
        }
        return root;
    }

    // 2) If current node doesn't contain point, search downward
    if (point[cd] < root->point[cd])
        root->left = deleteNodeRec(root->left, point, depth+1);
    else
        root->right = deleteNodeRec(root->right, point, depth+1);
    return root;
}

// Function to delete a given point from K D Tree with 'root'
Node* deleteNode(Node *root, int point[])
{
    // Pass depth as 0
    return deleteNodeRec(root, point, 0);
}

// Driver program to test above functions
int main()
{
    struct Node *root = NULL;
    int points[][k] = {{30, 40}, {5, 25}, {70, 70},
                      {10, 12}, {50, 30}, {35, 45}};

    int n = sizeof(points)/sizeof(points[0]);

    for (int i=0; i<n; i++)
        root = insert(root, points[i]);

    // Delet (30, 40);
    root = deleteNode(root, points[0]);

    cout << "Root after deletion of (30, 40) is: ";
    cout << root->point[0] << ", " << root->point[1] << endl;

    return 0;
}

```

Output:

```

Root after deletion of (30, 40)
35, 45

```

Source:

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Advanced Data Structures

Treap (A Randomized Binary Search Tree)

Like **Red-Black** and **AVL** Trees, Treap is a Balanced Binary Search Tree, but not guaranteed to have height as $O(\log n)$. The idea is to use Randomization and Binary Heap property to maintain balance with high probability. The expected time complexity of search, insert and delete is $O(\log n)$.

treap

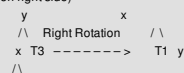
Every node of Treap maintains two values.

- 1) **Key** Follows standard BST ordering (left is smaller and right is greater)
- 2) **Priority** Randomly assigned value that follows Max-Heap property.

Basic Operation on Treap:

Like other self-balancing Binary Search Trees, Treap uses rotations to maintain Max-Heap property during insertion and deletion.

T1, T2 and T3 are subtrees of the tree rooted with y (on left side)
or x (on right side)



search(x)

Perform standard **BST Search** to find x.

Insert(x):

- 1) Create new node with key equals to x and value equals to a random value.
- 2) Perform standard **BST insert**.
- 3) Use rotations to make sure that inserted node's priority follows max heap property.

treapInsert

Delete(x):

- 1) If node to be deleted is a leaf, delete it.
- 2) Else replace node's priority with minus infinite ($-\infty$), and do appropriate rotations to bring the node down to a leaf.

treapDelete

Refer [Implementation of Treap Search, Insert and Delete](#) for more details.

References:

<https://en.wikipedia.org/wiki/Treap>

<https://courses.cs.washington.edu/courses/cse326/00wi/handouts/lecture19/sld017.htm>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner

Company Wise Coding Practice

Advanced Data Structure
Advanced Data Structures
Self-Balancing-BST

Ternary Search Tree

A ternary search tree is a special trie data structure where the child nodes of a standard trie are ordered as a binary search tree.

Representation of ternary search trees:

Unlike trie(standard) data structure where each node contains 26 pointers for its children, each node in a ternary search tree contains only 3 pointers:

1. The left pointer points to the node whose value is less than the value in the current node.
2. The equal pointer points to the node whose value is equal to the value in the current node.
3. The right pointer points to the node whose value is greater than the value in the current node.

Apart from above three pointers, each node has a field to indicate data(character in case of dictionary) and another field to mark end of a string.

So, more or less it is similar to BST which stores data based on some order. However, data in a ternary search tree is distributed over the nodes. e.g. It needs 4 nodes to store the word "Geek".

Below figure shows how exactly the words in a ternary search tree are stored?



One of the advantage of using ternary search trees over tries is that ternary search trees are a more space efficient (involve only three pointers per node as compared to 26 in standard tries). Further, ternary search trees can be used any time a hashtable would be used to store strings.

Tries are suitable when there is a proper distribution of words over the alphabets so that spaces are utilized most efficiently. Otherwise ternary search trees are better. Ternary search trees are efficient to use(in terms of space) when the strings to be stored share a common prefix.

Applications of ternary search trees:

1. Ternary search trees are efficient for queries like "Given a word, find the next word in dictionary(near-neighbor lookups)" or "Find all telephone numbers starting with 9342" or "typing few starting characters in a web browser displays all website names with this prefix"(Auto complete feature)".
2. Used in spell checks: Ternary search trees can be used as a dictionary to store all the words. Once the word is typed in an editor, the word can be parallelly searched in the ternary search tree to check for correct spelling.

Implementation:

Following is C implementation of ternary search tree. The operations implemented are, search, insert and traversal.

```
// C program to demonstrate Ternary Search Tree (TST) insert, traverse
// and search operations
#include <stdio.h>
#include <stdlib.h>
#define MAX 50

// A node of ternary search tree
struct Node
{
    char data;

    // True if this character is last character of one of the words
    unsigned isEndOfString: 1;

    struct Node *left, *eq, *right;
};

// A utility function to create a new ternary search tree node
struct Node* newNode(char data)
{
    struct Node* temp = (struct Node*) malloc(sizeof( struct Node ));
    temp->data = data;
    temp->isEndOfString = 0;
}
```

```

temp->left = temp->eq = temp->right = NULL;
return temp;
}

// Function to insert a new word in a Ternary Search Tree
void insert(struct Node* root, char *word)
{
    // Base Case: Tree is empty
    if (!(*root))
        *root = newNode(*word);

    // If current character of word is smaller than root's character,
    // then insert this word in left subtree of root
    if ((*word) < (*root->data))
        insert(&(*root->left), word);

    // If current character of word is greater than root's character,
    // then insert this word in right subtree of root
    else if ((*word) > (*root->data))
        insert(&(*root->right), word);

    // If current character of word is same as root's character,
    else
    {
        if (*(word+1))
            insert(&(*root->eq), word+1);

        // the last character of the word
        else
            (*root->isEndOfString) = 1;
    }
}

// A recursive function to traverse Ternary Search Tree
void traverseTSTUtil(struct Node* root, char* buffer, int depth)
{
    if (root)
    {
        // First traverse the left subtree
        traverseTSTUtil(root->left, buffer, depth);

        // Store the character of this node
        buffer[depth] = root->data;
        if (root->isEndOfString)
        {
            buffer[depth+1] = '\0';
            printf( "%s\n", buffer);
        }

        // Traverse the subtree using equal pointer (middle subtree)
        traverseTSTUtil(root->eq, buffer, depth + 1);

        // Finally Traverse the right subtree
        traverseTSTUtil(root->right, buffer, depth);
    }
}

// The main function to traverse a Ternary Search Tree.
// It mainly uses traverseTSTUtil()
void traverseTST(struct Node* root)
{
    char buffer[MAX];
    traverseTSTUtil(root, buffer, 0);
}

// Function to search a given word in TST
int searchTST(struct Node *root, char *word)
{
    if (!root)
        return 0;

    if (*word < (root->data))
        return searchTST(root->left, word);

    else if (*word > (root->data))
        return searchTST(root->right, word);

    else
    {
        if (*(word+1) == '\0')
            return root->isEndOfString;

        return searchTST(root->eq, word+1);
    }
}

// Driver program to test above functions
int main()
{
    struct Node *root = NULL;

    insert(&root, "cat");
    insert(&root, "cats");
    insert(&root, "bu");
    insert(&root, "bug");

    printf("Following is traversal of ternary search tree\n");
    traverseTST(root);

    printf("\nFollowing are search results for cats, bu and cat respectively\n");
    searchTST(root, "cats")? printf("Found\n"): printf("Not Found\n");
    searchTST(root, "bu")? printf("Found\n"): printf("Not Found\n");
    searchTST(root, "cat")? printf("Found\n"): printf("Not Found\n");

    return 0;
}

```

Output:

```

Following is traversal of ternary search tree
bug
cat
cats
up

```

Following are search results for cats, bu and cat respectively
Found
Not Found
Found

Time Complexity: The time complexity of the ternary search tree operations is similar to that of binary search tree. i.e. the insertion, deletion and search operations take time proportional to the height of the ternary search tree. The space is proportional to the length of the string to be stored.

Reference:

http://en.wikipedia.org/wiki/Ternary_search_tree

This article is compiled by **Aashish Barnwal** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Advance Data Structures
Advanced Data Structures

Interval Tree

Consider a situation where we have a set of intervals and we need following operations to be implemented efficiently.

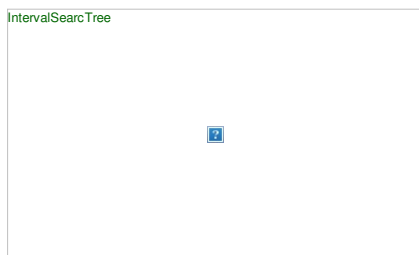
- 1) Add an interval
- 2) Remove an interval
- 3) Given an interval x , find if x overlaps with any of the existing intervals.

Interval Tree: The idea is to augment a self-balancing Binary Search Tree (BST) like **Red Black Tree**, **AVL Tree**, etc to maintain set of intervals so that all operations can be done in $O(\log n)$ time.

Every node of Interval Tree stores following information.

- a) i : An interval which is represented as a pair $[low, high]$
- b) **max**: Maximum $high$ value in subtree rooted with this node.

The low value of an interval is used as key to maintain order in BST. The insert and delete operations are same as insert and delete in self-balancing BST used.



The main operation is to search for an overlapping interval. Following is algorithm for searching an overlapping interval x in an Interval tree rooted with $root$.

```
Interval overlappingIntervalSearch(root, x)
1) If x overlaps with root's interval, return the root's interval.

2) If left child of root is not empty and the max in left child
is greater than x's low value, recur for left child

3) Else recur for right child.
```

How does the above algorithm work?

Let the interval to be searched be x . We need to prove this in for following two cases.

Case 1: When we go to right subtree, one of the following must be true.

- a) There is an overlap in right subtree: This is fine as we need to return one overlapping interval.
- b) There is no overlap in either subtree: We go to right subtree only when either left is NULL or maximum value in left is smaller than $x.low$. So the interval cannot be present in left subtree.

Case 2: When we go to left subtree, one of the following must be true.

- a) There is an overlap in left subtree: This is fine as we need to return one overlapping interval.
- b) There is no overlap in either subtree: This is the most important part. We need to consider following facts.
... We went to left subtree because $x.low$ in left subtree
.... max in left subtree is a high of one of the intervals let us say $[a, max]$ in left subtree.
.... Since x doesn't overlap with any node in left subtree $x.low$ must be smaller than 'a'.
.... All nodes in BST are ordered by low value, so all nodes in right subtree must have low value greater than 'a'.
.... From above two facts, we can say all intervals in right subtree have low value greater than $x.low$. So x cannot overlap with any interval in right subtree.

Implementation of Interval Tree:

Following is C++ implementation of Interval Tree. The implementation uses basic **insert operation** of BST to keep things simple. Ideally it should be **insertion of AVL Tree** or **insertion of Red-Black Tree**. **Deletion from BST** is left as an exercise.

```
#include <iostream>
using namespace std;

// Structure to represent an interval
struct Interval
{
    int low, high;
};

// Structure to represent a node in Interval Search Tree
struct ITNode
{
    Interval *i; // i could also be a normal variable
    int max;
    ITNode *left, *right;
};

// A utility function to create a new Interval Search Tree Node
ITNode * newNode(Interval i)
{
    ITNode *temp = new ITNode;
    temp->i = new Interval(i);
    temp->max = i.high;
    temp->left = temp->right = NULL;
};

// A utility function to insert a new Interval Search Tree Node
// This is similar to BST Insert. Here the low value of interval
// is used to maintain BST property
ITNode *insert(ITNode *root, Interval i)
{
    // Base case: Tree is empty, new node becomes root
```

```

if (root == NULL)
    return newNode(i);

// Get low value of interval at root
int l = root->l->low;

// If root's low value is smaller, then new interval goes to
// left subtree
if (i.low < l)
    root->left = insert(root->left, i);

// Else, new node goes to right subtree.
else
    root->right = insert(root->right, i);

// Update the max value of this ancestor if needed
if (root->max < i.high)
    root->max = i.high;

return root;
}

// A utility function to check if given two intervals overlap
bool doOverlap(Interval i1, Interval i2)
{
    if (i1.low <= i2.high && i2.low <= i1.high)
        return true;
    return false;
}

// The main function that searches a given interval i in a given
// Interval Tree.
Interval *overlapSearch(ITNode *root, Interval i)
{
    // Base Case, tree is empty
    if (root == NULL) return NULL;

    // If given interval overlaps with root
    if (doOverlap(*root->i, i))
        return root->i;

    // If left child of root is present and max of left child is
    // greater than or equal to given interval, then i may
    // overlap with an interval in left subtree
    if (root->left != NULL && root->left->max >= i.low)
        return overlapSearch(root->left, i);

    // Else interval can only overlap with right subtree
    return overlapSearch(root->right, i);
}

void inorder(ITNode *root)
{
    if (root == NULL) return;

    inorder(root->left);

    cout << "[" << root->i->low << ", " << root->i->high << "]"
        << " max = " << root->max << endl;

    inorder(root->right);
}

// Driver program to test above functions
int main()
{
    // Let us create interval tree shown in above figure
    Interval ints[] = {{15, 20}, {10, 30}, {17, 19},
        {5, 20}, {12, 15}, {30, 40}};
    int n = sizeof(ints)/sizeof(ints[0]);
    ITNode *root = NULL;
    for (int i = 0; i < n; i++)
        root = insert(root, ints[i]);

    cout << "Inorder traversal of constructed Interval Tree is\n";
    inorder(root);

    Interval x = {6, 7};

    cout << "\nSearching for interval [" << x.low << ", " << x.high << "]";
    Interval *res = overlapSearch(root, x);
    if (res == NULL)
        cout << "\nNo Overlapping Interval";
    else
        cout << "\nOverlaps with [" << res->low << ", " << res->high << "]";
    return 0;
}

```

Output:

```

Inorder traversal of constructed Interval Tree is
[5, 20] max = 20
[10, 30] max = 30
[12, 15] max = 15
[15, 20] max = 40
[17, 19] max = 40
[30, 40] max = 40

Searching for interval [6,7]
Overlaps with [5, 20]

```

Applications of Interval Tree:

Interval tree is mainly a geometric data structure and often used for windowing queries, for instance, to find all roads on a computerized map inside a rectangular viewport, or to find all visible elements inside a three-dimensional scene (Source [Wiki](#)).

Interval Tree vs Segment Tree

Both segment and interval trees store intervals. Segment tree is mainly optimized for queries for a given point, and interval trees are mainly optimized for overlapping queries for a given interval.

Exercise:

- 1) Implement delete operation for interval tree.
- 2) Extend the intervalSearch() to print all overlapping intervals instead of just one.

http://en.wikipedia.org/wiki/Interval_tree

<http://www.cse.unr.edu/~mgunes/cs302/IntervalTrees.pptx>

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

<https://www.youtube.com/watch?v=dQF0zyaym8A>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Advance Data Structures
Advanced Data Structures

Implement LRU Cache

How to implement LRU caching scheme? What data structures should be used?

We are given total possible page numbers that can be referred. We are also given cache (or memory) size (Number of page frames that cache can hold at a time). The LRU caching scheme is to remove the least recently used frame when the cache is full and a new page is referenced which is not there in cache. Please see the Galvin book for more details (see the LRU page replacement slide [here](#)).

We use two data structures to implement an LRU Cache.

1. **Queue** which is implemented using a doubly linked list. The maximum size of the queue will be equal to the total number of frames available (cache size). The most recently used pages will be near front end and least recently pages will be near rear end.
2. **A Hash** with page number as key and address of the corresponding queue node as value.

When a page is referenced, the required page may be in the memory. If it is in the memory, we need to detach the node of the list and bring it to the front of the queue.

If the required page is not in the memory, we bring that in memory. In simple words, we add a new node to the front of the queue and update the corresponding node address in the hash. If the queue is full, i.e. all the frames are full, we remove a node from the rear of queue, and add the new node to the front of queue.

Note: Initially no page is in the memory.

C implementation:

```
// A C program to show implementation of LRU cache
#include <stdio.h>
#include <stdlib.h>

// A Queue Node (Queue is implemented using Doubly Linked List)
typedef struct QNode
{
    struct QNode *prev, *next;
    unsigned pageNumber; // the page number stored in this QNode
} QNode;

// A Queue (A FIFO collection of Queue Nodes)
typedef struct Queue
{
    unsigned count; // Number of filled frames
    unsigned numberOfFrames; // total number of frames
    QNode *front, *rear;
} Queue;

// A hash (Collection of pointers to Queue Nodes)
typedef struct Hash
{
    int capacity; // how many pages can be there
    QNode* *array; // an array of queue nodes
} Hash;

// A utility function to create a new Queue Node. The queue Node
// will store the given 'pageNumber'
QNode* newQNode( unsigned pageNumber )
{
    // Allocate memory and assign 'pageNumber'
    QNode* temp = (QNode *) malloc( sizeof( QNode ) );
    temp->pageNumber = pageNumber;

    // Initialize prev and next as NULL
    temp->prev = temp->next = NULL;

    return temp;
}

// A utility function to create an empty Queue.
// The queue can have at most 'numberOfFrames' nodes
Queue* createQueue( int numberOfFrames )
{
    Queue* queue = (Queue *) malloc( sizeof( Queue ) );

    // The queue is empty
    queue->count = 0;
    queue->front = queue->rear = NULL;

    // Number of frames that can be stored in memory
    queue->numberOfFrames = numberOfFrames;

    return queue;
}

// A utility function to create an empty Hash of given capacity
Hash* createHash( int capacity )
{
    // Allocate memory for hash
    Hash* hash = (Hash *) malloc( sizeof( Hash ) );
    hash->capacity = capacity;

    // Create an array of pointers for refering queue nodes
    hash->array = (QNode **) malloc( hash->capacity * sizeof( QNode* ) );

    // Initialize all hash entries as empty
    int i;
    for( i = 0; i < hash->capacity; ++i )
        hash->array[i] = NULL;

    return hash;
}

// A function to check if there is slot available in memory
int AreAllFramesFull( Queue* queue )
{
    return queue->count == queue->numberOfFrames;
}
```

```

// A utility function to check if queue is empty
int isEmpty( Queue* queue )
{
    return queue->rear == NULL;
}

// A utility function to delete a frame from queue
void dequeue( Queue* queue )
{
    if( isEmpty( queue ) )
        return;

    // If this is the only node in list, then change front
    if (queue->front == queue->rear)
        queue->front = NULL;

    // Change rear and remove the previous rear
    QNode* temp = queue->rear;
    queue->rear = queue->rear->prev;

    if (queue->rear)
        queue->rear->next = NULL;

    free( temp );

    // decrement the number of full frames by 1
    queue->count--;
}

// A function to add a page with given 'pageNumber' to both queue
// and hash
void enqueue( Queue* queue, Hash* hash, unsigned pageNumber )
{
    // If all frames are full, remove the page at the rear
    if ( AreAllFramesFull ( queue ) )
    {
        // remove page from hash
        hash->array[ queue->rear->pageNumber ] = NULL;
        dequeue( queue );
    }

    // Create a new node with given page number,
    // And add the new node to the front of queue
    QNode* temp = newQNode( pageNumber );
    temp->next = queue->front;

    // If queue is empty, change both front and rear pointers
    if ( isEmpty( queue ) )
        queue->rear = queue->front = temp;
    else // Else change the front
    {
        queue->front->prev = temp;
        queue->front = temp;
    }

    // Add page entry to hash also
    hash->array[ pageNumber ] = temp;

    // increment number of full frames
    queue->count++;
}

// This function is called when a page with given 'pageNumber' is referenced
// from cache (or memory). There are two cases:
// 1. Frame is not there in memory, we bring it in memory and add to the front
// of queue
// 2. Frame is there in memory, we move the frame to front of queue
void ReferencePage( Queue* queue, Hash* hash, unsigned pageNumber )
{
    QNode* reqPage = hash->array[ pageNumber ];

    // the page is not in cache, bring it
    if ( reqPage == NULL )
        enqueue( queue, hash, pageNumber );

    // page is there and not at front, change pointer
    else if (reqPage != queue->front)
    {
        // Unlink requested page from its current location
        // in queue.
        reqPage->prev->next = reqPage->next;
        if (reqPage->next)
            reqPage->next->prev = reqPage->prev;

        // If the requested page is rear, then change rear
        // as this node will be moved to front
        if (reqPage == queue->rear)
        {
            queue->rear = reqPage->prev;
            queue->rear->next = NULL;
        }

        // Put the requested page before current front
        reqPage->next = queue->front;
        reqPage->prev = NULL;

        // Change prev of current front
        reqPage->next->prev = reqPage;

        // Change front to the requested page
        queue->front = reqPage;
    }
}

// Driver program to test above functions
int main()
{
    // Let cache can hold 4 pages
    Queue* q = createQueue( 4 );

    // Let 10 different pages can be requested (pages to be
    // referenced are numbered from 0 to 9
    Hash* hash = createHash( 10 );

    // Let us refer pages 1, 2, 3, 1, 4, 5

```



```

ReferencePage( q, hash, 1);
ReferencePage( q, hash, 2);
ReferencePage( q, hash, 3);
ReferencePage( q, hash, 1);
ReferencePage( q, hash, 4);
ReferencePage( q, hash, 5);

// Let us print cache frames after the above referenced pages
printf ("%d ", q->front->pageNumber);
printf ("%d ", q->front->next->pageNumber);
printf ("%d ", q->front->next->next->pageNumber);
printf ("%d ", q->front->next->next->next->pageNumber);

return 0;
}

```

Output:

```
5 4 1 3
```

This article is compiled by [Aashish Barnwal](#) and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Queue
Advance Data Structures
Advanced Data Structures
Queue

Sort numbers stored on different machines

Given N machines. Each machine contains some numbers in sorted form. But the amount of numbers, each machine has is not fixed. Output the numbers from all the machine in sorted non-decreasing form.

```

Example:
Machine M1 contains 3 numbers: {30, 40, 50}
Machine M2 contains 2 numbers: {35, 45}
Machine M3 contains 5 numbers: {10, 60, 70, 80, 100}

Output: {10, 30, 35, 40, 45, 50, 60, 70, 80, 100}

```

Representation of stream of numbers on each machine is considered as linked list. A Min Heap can be used to print all numbers in sorted order.

Following is the detailed process

1. Store the head pointers of the linked lists in a minHeap of size N where N is number of machines.
2. Extract the minimum item from the minHeap. Update the minHeap by replacing the head of the minHeap with the next number from the linked list or by replacing the head of the minHeap with the last number in the minHeap followed by decreasing the size of heap by 1.
3. Repeat the above step 2 until heap is not empty.

Below is C++ implementation of the above approach.

```

// A program to take numbers from different machines and print them in sorted order
#include <stdio.h>

// A Linked List node
struct ListNode
{
    int data;
    struct ListNode* next;
};

// A Min Heap Node
struct MinHeapNode
{
    ListNode* head;
};

// A Min Heao (Collection of Min Heap nodes)
struct MinHeap
{
    int count;
    int capacity;
    MinHeapNode* array;
};

// A function to create a Min Heap of given capacity
MinHeap* createMinHeap( int capacity )
{
    MinHeap* minHeap = new MinHeap;
    minHeap->capacity = capacity;
    minHeap->count = 0;
    minHeap->array = new MinHeapNode [minHeap->capacity];
    return minHeap;
}

/* A utility function to insert a new node at the beginning
of linked list */
void push( ListNode** head_ref, int new_data )
{
    /* allocate node */
    ListNode* new_node = new ListNode;

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

// A utility function to swap two min heap nodes. This function
// is needed in minHeapify
void swap( MinHeapNode* a, MinHeapNode* b )
{
    MinHeapNode temp = *a;
    *a = *b;
    *b = temp;
}

```

```

// The standard minHeapify function.
void minHeapify( MinHeap* minHeap, int idx )
{
    int left, right, smallest;
    left = 2 * idx + 1;
    right = 2 * idx + 2;
    smallest = idx;

    if ( left < minHeap->count &&
        minHeap->array[left].head->data <
        minHeap->array[smallest].head->data
    )
        smallest = left;

    if ( right < minHeap->count &&
        minHeap->array[right].head->data <
        minHeap->array[smallest].head->data
    )
        smallest = right;

    if( smallest != idx )
    {
        swap( &minHeap->array[smallest], &minHeap->array[idx] );
        minHeapify( minHeap, smallest );
    }
}

// A utility function to check whether a Min Heap is empty or not
int isEmpty( MinHeap* minHeap )
{
    return (minHeap->count == 0);
}

// A standard function to build a heap
void buildMinHeap( MinHeap* minHeap )
{
    int i, n;
    n = minHeap->count - 1;
    for( i = (n - 1) / 2; i >= 0; --i )
        minHeapify( minHeap, i );
}

// This function inserts array elements to heap and then calls
// buildHeap for heap property among nodes
void populateMinHeap( MinHeap* minHeap, ListNode* *array, int n )
{
    for( int i = 0; i < n; ++i )
        minHeap->array[ minHeap->count++ ].head = array[i];

    buildMinHeap( minHeap );
}

// Return minimum element from all linked lists
ListNode* extractMin( MinHeap* minHeap )
{
    if( isEmpty( minHeap ) )
        return NULL;

    // The root of heap will have minimum value
    MinHeapNode temp = minHeap->array[0];

    // Replace root either with next node of the same list.
    if( temp.head->next )
        minHeap->array[0].head = temp.head->next;
    else // If list empty, then reduce heap size
    {
        minHeap->array[0] = minHeap->array[ minHeap->count - 1 ];
        --minHeap->count;
    }

    minHeapify( minHeap, 0 );
    return temp.head;
}

// The main function that takes an array of lists from N machines
// and generates the sorted output
void externalSort( ListNode *array[], int N )
{
    // Create a min heap of size equal to number of machines
    MinHeap* minHeap = createMinHeap( N );

    // populate first item from all machines
    populateMinHeap( minHeap, array, N );

    while ( !isEmpty( minHeap ) )
    {
        ListNode* temp = extractMin( minHeap );
        printf( "%d ", temp->data );
    }
}

// Driver program to test above functions
int main()
{
    int N = 3; // Number of machines

    // an array of pointers storing the head nodes of the linked lists
    ListNode *array[N];

    // Create a Linked List 30->40->50 for first machine
    array[0] = NULL;
    push ( &array[0], 50 );
    push ( &array[0], 40 );
    push ( &array[0], 30 );

    // Create a Linked List 35->45 for second machine
    array[1] = NULL;
    push ( &array[1], 45 );
    push ( &array[1], 35 );

    // Create Linked List 10->60->70->80 for third machine
    array[2] = NULL;
    push ( &array[2], 100 );
    push ( &array[2], 80 );
    push ( &array[2], 70 );
    push ( &array[2], 60 );
}

```

```
push (&array[2], 10);

// Sort all elements
externalSort( array, N );

return 0;
}
```

Output:

```
10 30 35 40 45 50 60 70 80 100
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Heap
Sorting
Advance Data Structures
Advanced Data Structures

Find the k most frequent words from a file

Given a book of words. Assume you have enough main memory to accommodate all words. design a data structure to find top K maximum occurring words. The data structure should be dynamic so that new words can be added.

A simple solution is to **use Hashing**. Hash all words one by one in a hash table. If a word is already present, then increment its count. Finally, traverse through the hash table and return the k words with maximum counts.

We can **use Trie and Min Heap** to get the k most frequent words efficiently. The idea is to use Trie for searching existing words adding new words efficiently. Trie also stores count of occurrences of words. A Min Heap of size k is used to keep track of k most frequent words at any point of time(Use of Min Heap is same as we used it to find k largest elements in [this post](#)).

Trie and Min Heap are linked with each other by storing an additional field in Trie 'indexMinHeap' and a pointer 'trNode' in Min Heap. The value of 'indexMinHeap' is maintained as -1 for the words which are currently not in Min Heap (or currently not among the top k frequent words). For the words which are present in Min Heap, 'indexMinHeap' contains, index of the word in Min Heap. The pointer 'trNode' in Min Heap points to the leaf node corresponding to the word in Trie.

Following is the complete process to print k most frequent words from a file.

Read all words one by one. For every word, insert it into Trie. Increase the counter of the word, if already exists. Now, we need to insert this word in min heap also. For insertion in min heap, 3 cases arise:

1. The word is already present. We just increase the corresponding frequency value in min heap and call minHeapify() for the index obtained by "indexMinHeap" field in Trie. When the min heap nodes are being swapped, we change the corresponding minHeapIndex in the Trie. Remember each node of the min heap is also having pointer to Trie leaf node.

2. The minHeap is not full. we will insert the new word into min heap & update the root node in the min heap node & min heap index in Trie leaf node. Now, call buildMinHeap().

3. The min heap is full. Two sub-cases arise.

....**3.1** The frequency of the new word inserted is less than the frequency of the word stored in the head of min heap. Do nothing.

....**3.2** The frequency of the new word inserted is greater than the frequency of the word stored in the head of min heap. Replace & update the fields. Make sure to update the corresponding min heap index of the "word to be replaced" in Trie with -1 as the word is no longer in min heap.

4. Finally, Min Heap will have the k most frequent words of all words present in given file. So we just need to print all words present in Min Heap.

```
// A program to find k most frequent words in a file
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define MAX_CHARS 26
#define MAX_WORD_SIZE 30

// A Trie node
struct TrieNode
{
    bool isEnd; // indicates end of word
    unsigned frequency; // the number of occurrences of a word
    int indexMinHeap; // the index of the word in minHeap
    TrieNode* child[MAX_CHARS]; // represents 26 slots each for 'a' to 'z'.
};

// A Min Heap node
struct MinHeapNode
{
    TrieNode* root; // indicates the leaf node of TRIE
    unsigned frequency; // number of occurrences
    char* word; // the actual word stored
};

// A Min Heap
struct MinHeap
{
    unsigned capacity; // the total size a min heap
    int count; // indicates the number of slots filled.
    MinHeapNode* array; // represents the collection of minHeapNodes
};

// A utility function to create a new Trie node
TrieNode* newTrieNode()
{
    // Allocate memory for Trie Node
    TrieNode* trieNode = new TrieNode;

    // Initialize values for new node
    trieNode->isEnd = 0;
    trieNode->frequency = 0;
    trieNode->indexMinHeap = -1;
    for( int i = 0; i < MAX_CHARS; ++i )
        trieNode->child[i] = NULL;

    return trieNode;
}

// A utility function to create a Min Heap of given capacity
MinHeap* createMinHeap( int capacity )
{
    MinHeap* minHeap = new MinHeap;

    minHeap->capacity = capacity;
    minHeap->count = 0;

    // Allocate memory for array of min heap nodes
    minHeap->array = new MinHeapNode [ minHeap->capacity ];

    return minHeap;
}
```

```

// A utility function to swap two min heap nodes. This function
// is needed in minHeapify
void swapMinHeapNodes ( MinHeapNode* a, MinHeapNode* b )
{
    MinHeapNode temp = *a;
    *a = *b;
    *b = temp;
}

// This is the standard minHeapify function. It does one thing extra.
// It updates the minHapIndex in Trie when two nodes are swapped in
// in min heap
void minHeapify( MinHeap* minHeap, int idx )
{
    int left, right, smallest;

    left = 2 * idx + 1;
    right = 2 * idx + 2;
    smallest = idx;
    if ( left < minHeap->count &&
        minHeap->array[ left ]. frequency <
        minHeap->array[ smallest ]. frequency
    )
        smallest = left;

    if ( right < minHeap->count &&
        minHeap->array[ right ]. frequency <
        minHeap->array[ smallest ]. frequency
    )
        smallest = right;

    if ( smallest != idx )
    {
        // Update the corresponding index in Trie node.
        minHeap->array[ smallest ]. root->indexMinHeap = idx;
        minHeap->array[ idx ]. root->indexMinHeap = smallest;

        // Swap nodes in min heap
        swapMinHeapNodes ( &minHeap->array[ smallest ], &minHeap->array[ idx ] );

        minHeapify( minHeap, smallest );
    }
}

// A standard function to build a heap
void buildMinHeap( MinHeap* minHeap )
{
    int n, i;
    n = minHeap->count - 1;

    for( i = ( n - 1 ) / 2; i >= 0; --i )
        minHeapify( minHeap, i );
}

// Inserts a word to heap, the function handles the 3 cases explained above
void insertInMinHeap( MinHeap* minHeap, TrieNode** root, const char* word )
{
    // Case 1: the word is already present in minHeap
    if( (*root)->indexMinHeap != -1 )
    {
        ++( minHeap->array[ (*root)->indexMinHeap ]. frequency );

        // percolate down
        minHeapify( minHeap, (*root)->indexMinHeap );
    }

    // Case 2: Word is not present and heap is not full
    else if( minHeap->count < minHeap->capacity )
    {
        int count = minHeap->count;
        minHeap->array[ count ]. frequency = (*root)->frequency;
        minHeap->array[ count ]. word = new char [ strlen( word ) + 1 ];
        strcpy( minHeap->array[ count ], word, word );

        minHeap->array[ count ]. root = *root;
        (*root)->indexMinHeap = minHeap->count;

        ++( minHeap->count );
        buildMinHeap( minHeap );
    }

    // Case 3: Word is not present and heap is full. And frequency of word
    // is more than root. The root is the least frequent word in heap,
    // replace root with new word
    else if ( (*root)->frequency > minHeap->array[0]. frequency )
    {
        minHeap->array[ 0 ]. root->indexMinHeap = -1;
        minHeap->array[ 0 ]. root = *root;
        minHeap->array[ 0 ]. root->indexMinHeap = 0;
        minHeap->array[ 0 ]. frequency = (*root)->frequency;

        // delete previously allocated memory and
        delete [] minHeap->array[ 0 ]. word;
        minHeap->array[ 0 ]. word = new char [ strlen( word ) + 1 ];
        strcpy( minHeap->array[ 0 ], word, word );

        minHeapify ( minHeap, 0 );
    }
}

// Inserts a new word to both Trie and Heap
void insertUtil ( TrieNode** root, MinHeap* minHeap,
                const char* word, const char* dupWord )
{
    // Base Case
    if ( *root == NULL )
        *root = new TrieNode();

    // There are still more characters in word
    if ( *word != '\0' )
        insertUtil ( &(*root)->child[ tolower( *word ) - 97 ],
                    minHeap, word + 1, dupWord );
    else // The complete word is processed
    {

```

```

// word is already present, increase the frequency
if ( (*root)->isEnd )
    ++( (*root)->frequency );
else
{
    (*root)->isEnd = 1;
    (*root)->frequency = 1;
}

// Insert in min heap also
insertInMinHeap( minHeap, root, dupWord );
}
}

// add a word to Trie & min heap. A wrapper over the insertUtil
void insertTrieAndHeap(const char *word, TrieNode** root, MinHeap* minHeap)
{
    insertUtil( root, minHeap, word, word );
}

// A utility function to show results, The min heap
// contains k most frequent words so far, at any time
void displayMinHeap( MinHeap* minHeap )
{
    int i;

    // print top K word with frequency
    for( i = 0; i < minHeap->count; ++i )
    {
        printf( "%s : %d\n", minHeap->array[i].word,
                minHeap->array[i].frequency );
    }
}

// The main funtion that takes a file as input, add words to heap
// and Trie, finally shows result from heap
void printKMostFreq( FILE* fp, int k )
{
    // Create a Min Heap of Size k
    MinHeap* minHeap = createMinHeap( k );

    // Create an empty Trie
    TrieNode* root = NULL;

    // A buffer to store one word at a time
    char buffer[MAX_WORD_SIZE];

    // Read words one by one from file. Insert the word in Trie and Min Heap
    while( !scanf( fp, "%s", buffer ) != EOF )
        insertTrieAndHeap(buffer, &root, minHeap);

    // The Min Heap will have the k most frequent words, so print Min Heap nodes
    displayMinHeap( minHeap );
}

// Driver program to test above functions
int main()
{
    int k = 5;
    FILE *fp = fopen ("file.txt", "r");
    if (fp == NULL)
        printf ("File doesn't exist ");
    else
        printKMostFreq (fp, k);
    return 0;
}

```

Output:

```

your : 3
well : 3
and : 4
to : 4
Geeks : 6

```

The above output is for a file with following content.

```

Welcome to the world of Geeks
This portal has been created to provide well written well thought and well explained
solutions for selected questions If you like Geeks for Geeks and would like to contribute
here is your chance You can write article and mail your article to contribute at
geeksforgeeks.org See your article appearing on the Geeks for Geeks main page and help
thousands of other Geeks

```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Searching
Advance Data Structures

Given a sequence of words, print all anagrams together | Set 2

Given an array of words, print all anagrams together. For example, if the given array is {"cat", "dog", "tac", "god", "act"}, then output may be "cat tac act dog god".

We have discussed two different methods in the [previous post](#). In this post, a more efficient solution is discussed.

Trie data structure can be used for a more efficient solution. Insert the sorted order of each word in the trie. Since all the anagrams will end at the same leaf node. We can start a linked list at the leaf nodes where each node represents the index of the original array of words. Finally, traverse the Trie. While traversing the Trie, traverse each linked list one line at a time. Following are the detailed steps.

- 1) Create an empty Trie
- 2) One by one take all words of input sequence. Do following for each word
 - ...a) Copy the word to a buffer.
 - ...b) Sort the buffer
 - ...c) Insert the sorted buffer and index of this word to Trie. Each leaf node of Trie is head of a Index list. The Index list stores index of words in original sequence. If sorted buffer is already present, we insert index of this word to the index list.
- 3) Traverse Trie. While traversing, if you reach a leaf node, traverse the index list. And print all words using the index obtained from Index list.

```

// An efficient program to print all anagrams together

```

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define NO_OF_CHARS 26

// Structure to represent list node for indexes of words in
// the given sequence. The list nodes are used to connect
// anagrams at leaf nodes of Trie
struct IndexNode
{
    int index;
    struct IndexNode* next;
};

// Structure to represent a Trie Node
struct TrieNode
{
    bool isEnd; // indicates end of word
    struct TrieNode* child[NO_OF_CHARS]; // 26 slots each for 'a' to 'z'
    struct IndexNode* head; // head of the index list
};

// A utility function to create a new Trie node
struct TrieNode* newTrieNode()
{
    struct TrieNode* temp = new TrieNode;
    temp->isEnd = 0;
    temp->head = NULL;
    for (int i = 0; i < NO_OF_CHARS; ++i)
        temp->child[i] = NULL;
    return temp;
}

/* Following function is needed for library function qsort(). Refer
http://www.cplusplus.com/reference/cstdlib/qsort/ */
int compare(const void* a, const void* b)
{ return *(char*)a - *(char*)b; }

/* A utility function to create a new linked list node */
struct IndexNode* newIndexNode(int index)
{
    struct IndexNode* temp = new IndexNode;
    temp->index = index;
    temp->next = NULL;
    return temp;
}

// A utility function to insert a word to Trie
void insert(struct TrieNode** root, char* word, int index)
{
    // Base case
    if (*root == NULL)
        *root = newTrieNode();

    if (*word != '\0')
        insert(&(*root->child[tolower(*word) - 'a']), word+1, index);
    else // If end of the word reached
    {
        // Insert index of this word to end of index linked list
        if ((*root->isEnd)
        {
            IndexNode* pCrawl = (*root->head;
            while( pCrawl->next )
                pCrawl = pCrawl->next;
            pCrawl->next = newIndexNode(index);
        }
        else // If Index list is empty
        {
            (*root->isEnd = 1;
            (*root->head = newIndexNode(index);
        }
    }
}

// This function traverses the built trie. When a leaf node is reached,
// all words connected at that leaf node are anagrams. So it traverses
// the list at leaf node and uses stored index to print original words
void printAnagramsUtil(struct TrieNode* root, char* wordArr[])
{
    if (root == NULL)
        return;

    // If a leaf node is reached, print all anagrams using the indexes
    // stored in index linked list
    if (root->isEnd)
    {
        // traverse the list
        IndexNode* pCrawl = root->head;
        while (pCrawl != NULL)
        {
            printf("%s\n", wordArr[pCrawl->index]);
            pCrawl = pCrawl->next;
        }
    }

    for (int i = 0; i < NO_OF_CHARS; ++i)
        printAnagramsUtil(root->child[i], wordArr);
}

// The main function that prints all anagrams together. wordArr[] is input
// sequence of words.
void printAnagramsTogether(char* wordArr[], int size)
{
    // Create an empty Trie
    struct TrieNode* root = NULL;

    // Iterate through all input words
    for (int i = 0; i < size; ++i)
    {
        // Create a buffer for this word and copy the word to buffer
        int len = strlen(wordArr[i]);
        char* buffer = new char[len+1];
        strcpy(buffer, wordArr[i]);
    }
}

```

```
// Sort the buffer
qsort( (void*)buffer, strlen(buffer), sizeof(char), compare );

// Insert the sorted buffer and its original index to Trie
insert(&root, buffer, i);
}

// Traverse the built Trie and print all anagrams together
printAnagramsUtil(root, wordArr);
}

// Driver program to test above functions
int main()
{
    char* wordArr[] = {"cat", "dog", "tac", "god", "act", "gdo"};
    int size = sizeof(wordArr) / sizeof(wordArr[0]);
    printAnagramsTogether(wordArr, size);
    return 0;
}
```

Output:

```
cat
tac
act
dog
god
gdo
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Strings
Advance Data Structures
anagram

Tournament Tree (Winner Tree) and Binary Heap

Given a team of N players. How many minimum games are required to find second best player?

We can use adversary arguments based on tournament tree (Binary Heap).

Tournament tree is a form of min (max) heap which is a complete binary tree. Every external node represents a player and internal node represents winner. In a tournament tree every internal node contains winner and every leaf node contains one player.

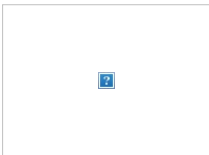
There will be $N - 1$ internal nodes in a binary tree with N leaf (external) nodes. For details see [this post](#) (put $n = 2$ in equation given in the post).

It is obvious that to select the best player among N players, $(N - 1)$ players to be eliminated, i.e. we need minimum of $(N - 1)$ games (comparisons). Mathematically we can prove it. In a binary tree $I = E - 1$, where I is number of internal nodes and E is number of external nodes. It means to find maximum or minimum element of an array, we need $N - 1$ (internal nodes) comparisons.

Second Best Player

The information explored during best player selection can be used to minimize the number of comparisons in tracing the next best players. For example, we can pick second best player in $(N + \log_2 N - 2)$ comparisons. For details read [this comment](#).

The following diagram displays a tournament tree (*winner tree*) as a max heap. Note that the concept of *loser tree* is different.



The above tree contains 4 leaf nodes that represent players and have 3 levels 0, 1 and 2. Initially 2 games are conducted at level 2, one between 5 and 3 and another one between 7 and 8. In the next move, one more game is conducted between 5 and 8 to conclude the final winner. Overall we need 3 comparisons. For second best player we need to trace the candidates participated with final winner, that leads to 7 as second best.

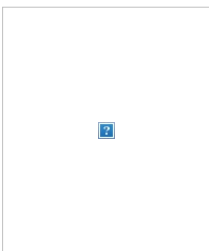
Median of Sorted Arrays

Tournament tree can effectively be used to find median of sorted arrays. Assume, given M sorted arrays of equal size L (for simplicity). We can attach all these sorted arrays to the tournament tree, one array per leaf. We need a tree of height **CEIL** ($\log_2 M$) to have atleast M external nodes.

Consider an example. Given 3 ($M = 3$) sorted integer arrays of maximum size 5 elements.

```
{ 2, 5, 7, 11, 15 } ---- Array1
{ 1, 3, 4 } ---- Array2
{ 6, 8, 12, 13, 14 } ---- Array3
```

What should be the height of tournament tree? We need to construct a tournament tree of height $\log_2 3 \approx 1.585 = 2$ rounded to next integer. A binary tree of height 2 will have 4 leaves to which we can attach the arrays as shown in the below figure.



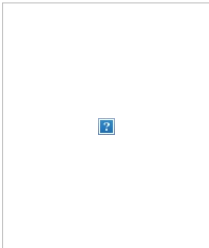
After the first tournament, the tree appears as below,



We can observe that the winner is from Array2. Hence the next element from Array2 will dive-in and games will be played along the winner path of previous tournament.

Note that infinity is used as sentinel element. Based on data being hold in nodes we can select the sentinel character. For example we usually store the pointers in nodes rather than keys, so NULL can serve as sentinel. If any of the array exhausts we will fill the corresponding leaf and upcoming internal nodes with sentinel.

After the second tournament, the tree appears as below,



The next winner is from Array1, so next element of Array1 array which is 5 will dive-in to the next round, and next tournament played along the path of 2.

The tournaments can be continued till we get median element which is $(5+3+5)/2 = 7$ th element. Note that there are even better algorithms for finding median of union of sorted arrays, for details see the related links given below.

In general with M sorted lists of size $L_1, L_2 \dots L_m$ requires time complexity of $O((L_1 + L_2 + \dots + L_m) * \log M)$ to merge all the arrays, and $O(m * \log M)$ time to find median, where m is median position.

Select smallest one million elements from one billion unsorted elements:

As a simple solution, we can sort the billion numbers and select first one million.

On a limited memory system sorting billion elements and picking the first one million seems to be impractical. We can use tournament tree approach. At any time only elements of tree to be in memory.

Split the large array (perhaps stored on disk) into smaller size arrays of size one million each (or even smaller that can be sorted by the machine). Sort these 1000 small size arrays and store them on disk as individual files. Construct a tournament tree which can have atleast 1000 leaf nodes (tree to be of height 10 since $2^9 < 1000 < 2^{10}$, if the individual file size is even smaller we will need more leaf nodes). Every leaf node will have an engine that picks next element from the sorted file stored on disk. We can play the tournament tree game to extract first one million elements.

Total cost = sorting 1000 lists of one million each + tree construction + tournaments

Implementation

We need to build the tree in bottom-up manner. All the leaf nodes filled first. Start at the left extreme of tree and fill along the breadth (i.e. from 2^{k-1} to $2^k - 1$ where k is depth of tree) and play the game. After practicing with few examples it will be easy to write code. Implementation is discussed in below code

[Second minimum element using minimum comparisons](#)

Related Posts

[Find the smallest and second smallest element in an array.](#)

[Second minimum element using minimum comparisons](#)

— by **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Heap
About Venki
Software Engineer
[View all posts by Venki](#) →

Decision Trees – Fake (Counterfeit) Coin Puzzle (12 Coin Puzzle)

Let us solve the classic "fake coin" puzzle using decision trees. There are the two different variants of the puzzle given below. I am providing description of both the puzzles below, try to solve on your own, assume $N = 8$.

Easy: Given a two pan fair balance and N identically looking coins, out of which only one coin is **lighter (or heavier)**. To figure out the odd coin, how many minimum number of weighing are required in the worst case?

Difficult: Given a two pan fair balance and N identically looking coins out of which only one coin **may be** defective. How can we trace which coin, if any, is odd one and also determine whether it is lighter or heavier in minimum number of trials in the worst case?

Let us start with relatively simple examples. After reading every problem try to solve on your own.

Problem 1: (Easy)

Given 5 coins out of which one coin is **lighter**. In the worst case, how many minimum number of weighing are required to figure out the odd coin?

Name the coins as 1, 2, 3, 4 and 5. We know that one coin is lighter. Considering best out come of balance, we can group the coins in two different ways, [(1, 2), (3, 4) and (5)], or [(12), (34) and (5)]. We can easily rule out groups like [(123) and (45)], as we will get obvious answer. Any other combination will fall into one of these two groups, like [(2)(45) and (13)], etc.

Consider the first group, pairs (1, 2) and (3, 4). We can check (1, 2), if they are equal we go ahead with (3, 4). We need two weighing in worst case. The same analogy can be applied when the coin is heavier.

With the second group, weigh (12) and (34). If they balance (5) is defective one, otherwise pick the lighter pair, and we need one more weighing to find odd one.

Both the combinations need two weighing in case of 5 coins with prior information of one coin is lighter.

Analysis: In general, if we know that the coin is heavy or light, we can trace the coin in $\log_3(N)$ trials (rounded to next integer). If we represent the outcome of balance as ternary tree, every leaf represent an outcome. Since any coin among N coins can be defective, we need to get a 3-ary tree having minimum of N leaves. A 3-ary tree at k -th level will have 3^k leaves and hence we need $3^k \geq N$.

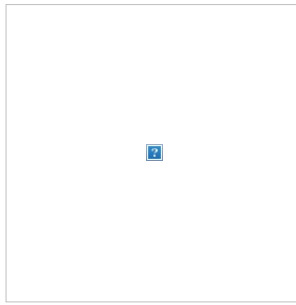
In other-words, in k trials we can examine upto 3^k coins, if we know whether the defective coin is heavier or lighter. Given that a coin is heavier, verify that 3 trials are sufficient to find the odd coin among 12 coins, because $3^2 < 12 < 3^3$.

Problem 2: (Difficult)

We are given 4 coins, out of which only one coin **may be** defective. We don't know, whether all coins are genuine or any defective one is present. How many number of weighing are required in worst case to figure out the odd coin, if present? We also need to tell whether it is heavier or lighter.

From the above analysis we may think that $k = 2$ trials are sufficient, since a two level 3-ary tree yields 9 leaves which is greater than $N = 4$ (read the problem once again). Note that it is impossible to solve above 4 coins problem in two weighing. The decision tree confirms the fact (try to draw).

We can group the coins in two different ways, [(12, 34)] or [(1, 2) and (3, 4)]. Let us consider the combination (12, 34), the corresponding decision tree is given below. Blue leaves are valid outcomes, and red leaves are impossible cases. We arrived at impossible cases due to the assumptions made earlier on the path.



The outcome can be $(12) < (34)$ i.e. we go on to left subtree or $(12) > (34)$ i.e. we go on to right subtree.

The left subtree is possible in two ways,

- A) Either 1 or 2 can be lighter OR
- B) Either 3 or 4 can be heavier.

Further on the left subtree, as second trial, we weigh $(1, 2)$ or $(3, 4)$. Let us consider $(3, 4)$ as the analogy for $(1, 2)$ is similar. The outcome of second trail can be three ways

- A) $(3) < (4)$ yielding 4 as defective heavier coin, OR
- B) $(3) > (4)$ yielding 3 as defective heavier coin OR
- C) $(3) = (4)$, yielding ambiguity. Here we need one more weighing to check a genuine coin against 1 or 2. In the figure I took $(3, 2)$ where 3 is confirmed as genuine. We can get $(3) > (2)$ in which 2 is lighter, or $(3) = (2)$ in which 1 is lighter. Note that it impossible to get $(3) < (2)$, it contradicts our assumption leaned to left side.

Similarly we can analyze the right subtree. We need two more weighings on right subtree as well.

Overall we need 3 weighings to trace the odd coin. Note that we are unable to utilize two outcomes of 3-ary trees. Also, the tree is not full tree, middle branch terminated after first weighing. Infact, we can get 27 leaves of 3 level full 3-ary tree, but only we got 11 leaves including impossible cases.

Analysis: Given N coins, all may be genuine or only one coin is defective. We need a decision tree with atleast $(2N + 1)$ leaves correspond to the outputs. Because there can be N leaves to be lighter, or N leaves to be heavier or one genuine case, on total $(2N + 1)$ leaves.

As explained earlier ternary tree at level k , can have utmost 3^k leaves and we need a tree with leaves of $3^k > (2N + 1)$.

In other words, we need atleast $k > \log_3(2N + 1)$ weighing to find the defective one.

Observe the above figure that not all the branches are generating leaves, i.e. we are missing valid outputs under some branches that leading to more number of trials. When possible, we should group the coins in such a way that every branch is going to yield valid output (in simple terms generate full 3-ary tree). Problem 4 describes this approach of 12 coins.

Problem 3: (Special case of two pan balance)

We are given 5 coins, a group of 4 coins out of which one coin is defective (we **don't know** whether it is heavier or lighter), and one coin is genuine. How many weighing are required in worst case to figure out the odd coin whether it is heavier or lighter?

Label the coins as 1, 2, 3, 4 and G (genuine). We now have some information on coin purity. We need to make use that in the groupings.

We can best group them as $\{(G1, 23) \text{ and } (4)\}$. Any other group can't generate full 3-ary tree, try yourself. The following diagram explains the procedure.



The middle case $(G1) = (23)$ is self explanatory, i.e. 1, 2, 3 are genuine and 4th coin can be figured out lighter or heavier in one more trial.

The left side of tree corresponds to the case $(G1) < (23)$. This is possible in two ways, either 1 should be lighter or either of $(2, 3)$ should be heavier. The former instance is obvious when next weighing $(2, 3)$ is balanced, yielding 1 as lighter. The later instance could be $(2) < (3)$ yielding 3 as heavier or $(2) > (3)$ yielding 2 as heavier. The leaf nodes on left branch are named to reflect these outcomes.

The right side of tree corresponds to the case $(G1) > (23)$. This is possible in two ways, either 1 is heavier or either of $(2, 3)$ should be lighter. The former instance is obvious when the next weighing $(2, 3)$ is balanced, yielding 1 as heavier. The later case could be $(2) < (3)$ yielding 2 as lighter coin, or $(2) > (3)$ yielding 3 as lighter.

In the above problem, under any possibility we need only two weighing. We are able to use all outcomes of two level full 3-ary tree. We started with $(N + 1) = 5$ coins where $N = 4$, we end up with $(2N + 1) = 9$ leaves. **Infact we should have 11 outcomes since we started with 5 coins, where are other 2 outcomes? These two outcomes can be declared at the root of tree itself (prior to first weighing), can you figure these two out comes?**

If we observe the figure, after the first weighing the problem reduced to "we know three coins, either one can be lighter (heavier) or one among other two can be heavier (lighter)". This can be solved in one weighing (read Problem 1).

Analysis: Given $(N + 1)$ coins, one is genuine and the rest N can be genuine or only one coin is defective. The required decision tree should result in minimum of $(2N + 1)$ leaves. Since the total possible outcomes are $(2(N + 1) + 1)$, number of weighing (trials) are given by the height of ternary tree, $k \geq \log_3[2(N + 1) + 1]$. Note the equality sign.

Rearranging k and N , we can weigh maximum of $N \leq (3^k - 3)/2$ coins in k trials.

Problem 4: (The classic 12 coin puzzle)

You are given two pan fair balance. You have 12 identically looking coins out of which one coin may be lighter or heavier. How can you find odd coin, if any, in minimum trials, also determine whether defective coin is lighter or heavier, in the worst case?

How do you want to group them? Bi-set or tri-set? Clearly we can discard the option of dividing into two equal groups. It can't lead to best tree. From the above two examples, we can ensure that the decision tree can be used in optimal way if we can reveal atleast one genuine coin. Remember to group coins such that the first weighing reveals atleast one genuine coin.

Let us name the coins as 1, 2, ..., 8, A, B, C and D. We can combine the coins into 3 groups, namely (1234) , (5678) and $(ABCD)$. Weigh (1234) and (5678) . You are encouraged to draw decision tree while reading the procedure. The outcome can be three ways,

1. $(1234) = (5678)$, both groups are equal. Defective coin may be in $(ABCD)$ group.
2. $(1234) < (5678)$, i.e. first group is less in weight than second group.
3. $(1234) > (5678)$, i.e. first group is more in weight than second group.

The output (1) can be solved in two more weighing as special case of two pan balance given in Problem 3. We know that groups (1234) and (5678) are genuine and defective coin may be in $(ABCD)$. Pick one genuine coin from any of weighed groups, and proceed with $(ABCD)$ as explained in Problem 3.

Outcomes (2) and (3) are special. In both the cases, we know that $(ABCD)$ is genuine. And also, we know a set of coins being lighter and a set of coins being heavier. We need to shuffle the weighed two groups in such a way that we end up with smaller height decision tree.

Consider the second outcome where $(1234) < (5678)$. It is possible when any coin among $(1, 2, 3, 4)$ is lighter or any coin among $(5, 6, 7, 8)$ is heavier. We revealed lighter or heavier possibility after first weighing. If we proceed as in Problem 1, we will not generate best decision tree. Let us shuffle coins as (1235) and $(4BCD)$ as new groups (there are different shuffles possible, they also lead to minimum weighing, can you try?). If we weigh these two groups again the outcome can be three ways, i) $(1235) < (4BCD)$ yielding one among 1, 2, 3 is lighter which is similar to Problem 1 explained above, we need one more weighing, ii) $(1235) = (4BCD)$ yielding one among 6, 7, 8 is heavier which is similar to Problem 1 explained above, we need one more weighing iii) $(1235) > (4BCD)$ yielding either 5 as heavier coin or 4 as lighter coin, at the expense of one more weighing.

Similar way we can also solve the right subtree (third outcome where $(1234) > (5678)$) in two more weighing.

We are able to solve the 12 coin puzzle in 3 weighing in the worst case.

Few Interesting Puzzles:

1. Solve Problem 4 with $N = 8$ and $N = 13$, How many minimum trials are required in each case?
2. Given a function `int weigh(A[], B[])` where A and B are arrays (need not be equal size). The function returns -1, 0 or 1. It returns 0 if sum of all elements in A and B are equal, -1 if $A < B$ and 1 if $A > B$. Given an array of 12 elements, all elements are equal except one. The odd element can be as that of others, smaller or greater than others. Write a program to find the odd element (if any) using `weigh()` minimum number of times.
3. You might have seen 3-pan balance in science labs during school days. Given a 3-pan balance (4 outcomes) and N coins, how many minimum trials are needed to figure out odd coin?

References:

Similar problem was provided in one of the exercises of the book "Introduction to Algorithms by Levitin". Specifically read section 5.5 and section 11.2 including exercises.

-- by **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
About Venki
Software Engineer
[View all posts by Venki](#) --

Spaghetti Stack

Spaghetti stack

A spaghetti stack is an N-ary tree data structure in which child nodes have pointers to the parent nodes (but not vice-versa)



Spaghetti stack structure is used in situations when records are dynamically pushed and popped onto a stack as execution progresses, but references to the popped records remain in use. Following are some applications of Spaghetti Stack.

Compilers for languages such as C create a spaghetti stack as it opens and closes symbol tables representing block scopes. When a new block scope is opened, a symbol table is pushed onto a stack. When the closing curly brace is encountered, the scope is closed and the symbol table is popped. But that symbol table is remembered, rather than destroyed. And of course it remembers its higher level "parent" symbol table and so on.

Spaghetti Stacks are also used to implement [Disjoint-set data structure](#).

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Sources:

http://en.wikipedia.org/wiki/Spaghetti_stack

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Stack
Advanced Data Structures

Data Structure for Dictionary and Spell Checker?

Which data structure can be used for efficiently building a word dictionary and Spell Checker?

The answer depends upon the functionalists required in Spell Checker and availability of memory. For example following are few possibilities.

Hashing is one simple option for this. We can put all words in a hash table. Refer [this](#) paper which compares hashing with self-balancing Binary Search Trees and Skip List, and shows that hashing performs better.

Hashing doesn't support operations like prefix search. Prefix search is something where a user types a prefix and your dictionary shows all words starting with that prefix. Hashing also doesn't support efficient printing of all words in dictionary in alphabetical order and nearest neighbor search.

If we want both operations, look up and prefix search, **Trie** is suited. With Trie, we can support all operations like insert, search, delete in $O(n)$ time where n is length of the word to be processed. Another advantage of Trie is, we can print all words in alphabetical order which is not possible with hashing.

The disadvantage of Trie is, it requires lots of space. If space is concern, then **Ternary Search Tree** can be preferred. In Ternary Search Tree, time complexity of search operation is $O(h)$ where h is height of the tree. Ternary Search Trees also supports other operations supported by Trie like prefix search, alphabetical order printing and nearest neighbor search.

If we want to support suggestions, like google shows "*did you mean ...*", then we need to find the closest word in dictionary. The closest word can be defined as the word that can be obtained with minimum number of character transformations (add, delete, replace). A Naive way is to take the given word and generate all words which are 1 distance (1 edit or 1 delete or 1 replace) away and one by one look them in dictionary. If nothing found, then look for all words which are 2 distant and so on. There are many complex algorithms for this. As per the [wiki page](#), The most successful algorithm to date is Andrew Golding and Dan Roth's Window-based spelling correction algorithm.

See [this](#) for a simple spell checker implementation.

This article is compiled by **Piyush**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Advance Data Structures
Advanced Data Structures

Cartesian Tree

A Cartesian tree is a tree data structure created from a set of data that obeys the following structural invariants:

1. The tree obeys in the min (or max) heap property -- each node is less (or greater) than its children.
2. An inorder traversal of the nodes yields the values in the same order in which they appear in the initial sequence.

Suppose we have an input array- {5,10,40,30,28}. Then the max-heap Cartesian Tree would be.

cartesianTree0



A min-heap Cartesian Tree of the above input array will be-

cartesianTree1



Note:

1. Cartesian Tree is not a height-balanced tree.
2. Cartesian tree of a sequence of distinct numbers is always unique.

Cartesian tree of a sequence of distinct numbers is always unique.

We will prove this using induction. As a base case, empty tree is always unique. For the inductive case, assume that for all trees containing $n' < n$ elements, there is a unique Cartesian tree for each sequence of n' nodes. Now take any sequence of n elements. Because a Cartesian tree is a min-heap, the smallest element of the sequence must be the root of the Cartesian tree. Because an inorder traversal of the elements must yield the input sequence, we know that all nodes to the left of the min element must be in its left subtree and similarly for the nodes to the right. Since the left and right subtree are both Cartesian trees with at most $n-1$ elements in them (since the min element is at the root), by the induction hypothesis there is a unique Cartesian tree that could be the left or right subtree. Since all our decisions were forced, we end up with a unique tree, completing the induction.

How to construct Cartesian Tree?

A $O(n^2)$ solution for construction of Cartesian Tree is discussed [here](#) (Note that the above program [here](#) constructs the "special binary tree" (which is nothing but a Cartesian tree)

A $O(n)$ Algorithm :

It's possible to build a Cartesian tree from a sequence of data in linear time. Beginning with the empty tree,

Scan the given sequence from left to right adding new nodes as follows:

1. Position the node as the right child of the rightmost node.
2. Scan upward from the node's parent up to the root of the tree until a node is found whose value is greater than the current value.
3. If such a node is found, set its right child to be the new node, and set the new node's left child to be the previous right child.
4. If no such node is found, set the new child to be the root, and set the new node's left child to be the previous tree.

```
// A O(n) C++ program to construct cartesian tree
// from a given array
#include<bits/stdc++.h>

/* A binary tree node has data, pointer to left
child and a pointer to right child */
struct Node
{
    int data;
    Node *left, *right;
};

/* This function is here just to test buildTree() */
void printInorder (Node* node)
{
    if (node == NULL)
        return;
    printInorder (node->left);
    cout << node->data << " ";
    printInorder (node->right);
}

// Recursively construct subtree under given root using
// leftChild[] and rightChild[]
Node * buildCartesianTreeUtil (int root, int arr[],
                              int parent[], int leftchild[], int rightchild[])
{
    if (root == -1)
        return NULL;

    // Create a new node with root's data
    Node *temp = new Node;
    temp->data = arr[root];

    // Recursively construct left and right subtrees
    temp->left = buildCartesianTreeUtil( leftchild[root],
        arr, parent, leftchild, rightchild );
    temp->right = buildCartesianTreeUtil( rightchild[root],
        arr, parent, leftchild, rightchild );

    return temp ;
}

// A function to create the Cartesian Tree in O(N) time
Node * buildCartesianTree (int arr[], int n)
{
    // Arrays to hold the index of parent, left-child,
    // right-child of each number in the input array
    int parent[n], leftchild[n], rightchild[n];

    // Initialize all array values as -1
    memset(parent, -1, sizeof(parent));
    memset(leftchild, -1, sizeof(leftchild));
    memset(rightchild, -1, sizeof(rightchild));
```

```

// 'root' and 'last' stores the index of the root and the
// last processed of the Cartesian Tree.
// Initially we take root of the Cartesian Tree as the
// first element of the input array. This can change
// according to the algorithm
int root = 0, last;

// Starting from the second element of the input array
// to the last on scan across the elements, adding them
// one at a time.
for (int i=1; i<=n-1; i++)
{
    last = i-1;
    righchild[i] = -1;

    // Scan upward from the node's parent up to
    // the root of the tree until a node is found
    // whose value is greater than the current one
    // This is the same as Step 2 mentioned in the
    // algorithm
    while (arr[last] <= arr[i] && last != root)
        last = parent[last];

    // arr[i] is the largest element yet; make it
    // new root
    if (arr[last] <= arr[i])
    {
        parent[root] = i;
        leftchild[i] = root;
        root = i;
    }

    // Just insert it
    else if (righchild[last] == -1)
    {
        righchild[last] = i;
        parent[i] = last;
        leftchild[i] = -1;
    }

    // Reconfigure links
    else
    {
        parent[righchild[last]] = i;
        leftchild[i] = righchild[last];
        righchild[last] = i;
        parent[i] = last;
    }
}

// Since the root of the Cartesian Tree has no
// parent, so we assign it -1
parent[root] = -1;

return (buildCartesianTreeUtil (root, arr, parent,
                                leftchild, righchild));
}

/* Driver program to test above functions */
int main()
{
    /* Assume that inorder traversal of following tree
    is given
    40
   /  \
  10   30
 /    \
5     28 */

    int arr[] = {5, 10, 40, 30, 28};
    int n = sizeof(arr)/sizeof(arr[0]);

    Node *root = buildCartesianTree(arr, n);

    /* Let us test the built tree by printing Inorder
    traversal */
    printf("Inorder traversal of the constructed tree : \n");
    printInorder(root);

    return(0);
}

```

Output:

```

Inorder traversal of the constructed tree :
5 10 40 30 28

```

Time Complexity :

At first look, the code seems to be taking $O(n^2)$ time as there are two loop in buildCartesianTree(). But actually, it takes linear time.

The inner while loop represents the process in which we scan the tree upwards in the search of finding a suitable place to insert the new element. A keen observation can show that in total, the whole while loop(i.e- over all values from $i = 1$ to $i < n$) takes $O(n)$ time and not every time. Hence, the overall time complexity is $O(n)$.

Auxiliary Space:

We declare a structure for every node as well as three extra arrays- leftchild[], righchild[], parent[] to hold the indices of left-child, right-child, parent of each value in the input array. Hence the overall $O(4*n) = O(n)$ extra space.

Application of Cartesian Tree

- **Cartesian Tree Sorting**
- A range minimum query on a sequence is equivalent to a lowest common ancestor query on the sequence's Cartesian tree. Hence, RMQ may be reduced to LCA using the sequence's Cartesian tree.
- **Treap, a balanced binary search tree structure**, is a Cartesian tree of (key,priority) pairs; it is heap-ordered according to the priority values, and an inorder traversal gives the keys in sorted order.
- **Suffix tree** of a string may be constructed from the suffix array and the longest common prefix array. The first step is to compute the Cartesian tree of the longest common prefix array.

References:

http://wcipeg.com/wiki/Cartesian_tree

This article is contributed by **Rachit Belwariar**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Cartesian Tree Sorting

Prerequisites : [Cartesian Tree](#)

Cartesian Sort is an Adaptive Sorting as it sorts the data faster if data is partially sorted. In fact, there are very few sorting algorithms that make use of this fact.

For example consider the array {5, 10, 40, 30, 28}. The input data is partially sorted too as only one swap between "40" and "28" results in a completely sorted order. See how Cartesian Tree Sort will take advantage of this fact below.

Below are steps used for sorting.

Step 1 : Build a (min-heap) [Cartesian Tree](#) from the given input sequence.

ctree1



Step 2 : Starting from the root of the built Cartesian Tree, we push the nodes in a priority queue.

Then we pop the node at the top of the priority queue and push the children of the popped node in the priority queue in a pre-order manner.

1. Pop the node at the top of the priority queue and add it to a list.
2. Push left child of the popped node first (if present).
3. Push right child of the popped node next (if present).

ctree1



ctree2



How to build (min-heap) Cartesian Tree?

Building min-heap is similar to building a (max-heap) Cartesian Tree (discussed in [previous post](#)), except the fact that now we scan upward from the node's parent up to the root of the tree until a node is found whose value is smaller (and not larger as in the case of a max-heap Cartesian Tree) than the current one and then accordingly reconfigure links to build the min-heap Cartesian tree.

Why not to use only priority queue?

One might wonder that using priority queue would anyway result in a sorted data if we simply insert the numbers of the input array one by one in the priority queue (i.e- without constructing the Cartesian tree).

But the time taken differs a lot.

Suppose we take the input array – {5, 10, 40, 30, 28}

If we simply insert the input array numbers one by one (without using a Cartesian tree), then we may have to waste a lot of operations in adjusting the queue order everytime we insert the numbers (just like a typical heap performs those operations when a new number is inserted, as priority queue is nothing but a heap).

Whereas, here we can see that using a Cartesian tree took only 5 operations (see the above two figures in which we are continuously pushing and popping the nodes of Cartesian tree), which is linear as there are 5 numbers in the input array also. So we see that the best case of Cartesian Tree sort is $O(n)$, a thing where heap-sort will take much more number of operations, because it doesn't make advantage of the fact that the input data is partially sorted.

Why pre-order traversal?

The answer to this is that since Cartesian Tree is basically a heap- data structure and hence follows all the properties of a heap. Thus the root node is always smaller than both of its children. Hence, we use a pre-order fashion popping-and-pushing as in this, the root node is always pushed earlier than its children inside the priority queue and since the root node is always less than both its child, so we don't have to do extra operations inside the priority queue.

Refer to the below figure for better understanding-

ctree3



```

// A C++ program to implement Cartesian Tree sort
// Note that in this program we will build a min-heap
// Cartesian Tree and not max-heap.
#include<bits/stdc++.h>
using namespace std;

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct Node
{
    int data;
    Node *left, *right;
};

// Creating a shortcut for int, Node* pair type
typedef pair<int, Node*> INPair;

// This function sorts by pushing and popping the
// Cartesian Tree nodes in a pre-order like fashion
void pQBasedTraversal(Node* root)
{
    // We will use a priority queue to sort the
    // partially-sorted data efficiently.
    // Unlike Heap, Cartesian tree makes use of
    // the fact that the data is partially sorted
    priority_queue<INPair, vector<INPair>, greater<INPair>> pQueue;
    pQueue.push(make_pair(root->data, root));

    // Resembles a pre-order traverse as first data
    // is printed then the left and then right child.
    while (!pQueue.empty())
    {
        INPair popped_pair = pQueue.top();
        printf("%d ", popped_pair.first);

        pQueue.pop();

        if (popped_pair.second->left != NULL)
            pQueue.push(make_pair(popped_pair.second->left->data,
                                   popped_pair.second->left));

        if (popped_pair.second->right != NULL)
            pQueue.push(make_pair(popped_pair.second->right->data,
                                   popped_pair.second->right));
    }

    return;
}

Node *buildCartesianTreeUtil(int root, int arr[],
                             int parent[], int leftchild[], int rightchild[])
{
    if (root == -1)
        return NULL;

    Node *temp = new Node;

    temp->data = arr[root];
    temp->left = buildCartesianTreeUtil(leftchild[root],
                                       arr, parent, leftchild, rightchild);

    temp->right = buildCartesianTreeUtil(rightchild[root],
                                       arr, parent, leftchild, rightchild);

    return temp;
}

// A function to create the Cartesian Tree in O(N) time
Node *buildCartesianTree(int arr[], int n)
{
    // Arrays to hold the index of parent, left-child,
    // right-child of each number in the input array
    int parent[n], leftchild[n], rightchild[n];

    // Initialize all array values as -1
    memset(parent, -1, sizeof(parent));
    memset(leftchild, -1, sizeof(leftchild));
    memset(rightchild, -1, sizeof(rightchild));

    // 'root' and 'last' stores the index of the root and the
    // last processed of the Cartesian Tree.
    // Initially we take root of the Cartesian Tree as the
    // first element of the input array. This can change
    // according to the algorithm
    int root = 0, last;

    // Starting from the second element of the input array
    // to the last on scan across the elements, adding them
    // one at a time.
    for (int i=1; i<n-1; i++)
    {
        last = i-1;
        rightchild[i] = -1;

        // Scan upward from the node's parent up to
        // the root of the tree until a node is found
        // whose value is smaller than the current one
        // This is the same as Step 2 mentioned in the
        // algorithm
        while (arr[last] >= arr[i] && last != root)
            last = parent[last];

        // arr[i] is the smallest element yet; make it
        // new root
        if (arr[last] >= arr[i])
        {
            parent[root] = i;
            leftchild[i] = root;
            root = i;
        }

        // Just insert it
        else if (rightchild[last] == -1)
        {

```

```

        rightchild[last] = i;
        parent[i] = last;
        leftchild[i] = -1;
    }

    // Reconfigure links
    else
    {
        parent[rightchild[last]] = i;
        leftchild[i] = rightchild[last];
        rightchild[last] = i;
        parent[i] = last;
    }
}

// Since the root of the Cartesian Tree has no
// parent, so we assign it -1
parent[root] = -1;

return (buildCartesianTreeUtil (root, arr, parent,
                                leftchild, rightchild));
}

// Sorts an input array
int printSortedArr(int arr[], int n)
{
    // Build a cartesian tree
    Node *root = buildCartesianTree(arr, n);

    printf("The sorted array is-\n");

    // Do pr-order traversal and insert
    // in priority queue
    pQBasedTraversal(root);
}

/* Driver program to test above functions */
int main()
{
    /* Given input array- {5,10,40,30,28},
    it's corresponding unique Cartesian Tree
    is-

    5
    \
    10
    \
    28
    /
    30
    /
    40
    */

    int arr[] = {5, 10, 40, 30, 28};
    int n = sizeof(arr)/sizeof(arr[0]);

    printSortedArr(arr, n);

    return(0);
}

```

Output :

```

The sorted array is-
5 10 28 30 40

```

Time Complexity : **O(n)** best-case behaviour (when the input data is partially sorted), **O(n log n)** worst-case behavior (when the input data is not partially sorted)

Auxiliary Space : We use a priority queue and a Cartesian tree data structure. Now, at any moment of time the size of the priority queue doesn't exceeds the size of the input array, as we are constantly pushing and popping the nodes. Hence we are using O(n) auxiliary space.

References :

https://en.wikipedia.org/wiki/Adaptive_sort

http://11011110.livejournal.com/283412.htmlhttp://gradbot.blogspot.in/2010/06/cartesian-tree-sort.htmlhttp://www.keithschwarz.com/interesting/code/?dir=cartesian-tree-sorthttps://en.wikipedia.org/wiki/Cartesian_tree#Application_in_sorting

This article is contributed by **Rachit Belwariar**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Sorting

Sparse Set

How to do the following operations efficiently if there are large number of queries for them.

1. Insertion
2. Deletion
3. Searching
4. Clearing/Removing all the elements.

One solution is to use a Self-Balancing Binary Search Tree like Red-Black Tree, AVL Tree, etc. Time complexity of this solution for insertion, deletion and searching is O(Log n).

We can also use Hashing. With hashing, time complexity of first three operations is O(1). But time complexity of the fourth operation is O(n).

We can also use bit-vector (or direct access table), but bit-vector also requires O(n) time for clearing.

Sparse Set outperforms all BST, Hashing and bit vector. We assume that we are given range of data (or maximum value an element can have) and maximum number of elements that can be stored in set. The idea is to maintain two arrays: sparse[] and dense[].

```

dense[] ==> Stores the actual elements
sparse[] ==> This is like bit-vector where
we use elements as index. Here
values are not binary, but
indexes of dense array.

```

```

maxVal ==> Maximum value this set can
store. Size of sparse[] is
equal to maxVal + 1.
capacity ==> Capacity of Set. Size of sparse
is equal to capacity.
n ==> Current number of elements in
Set.

```

insert(x): Let **x** be the element to be inserted. If **x** is greater than **maxVal** or **n** (current number of elements) is greater than equal to capacity, we return.

If none of the above conditions is true, we insert **x** in **dense[]** at index **n** (position after last element in a 0 based indexed array), increment **n** by one (Current number of elements) and store **n** (index of **x** in **dense[]**) at **sparse[x]**.

search(x): To search an element **x**, we use **x** as index in **sparse[]**. The value **sparse[x]** is used as index in **dense[]**. And if value of **dense[sparse[x]]** is equal to **x**, we return **dense[x]**. Else we return -1.

delete(x): To delete an element **x**, we replace it with last element in **dense[]** and update index of last element in **sparse[]**. Finally decrement **n** by 1.

clear(): Set **n** = 0.

print(): We can print all elements by simply traversing **dense[]**.

Illustration:

Let there be a set with two elements {3, 5}, maximum value as 10 and capacity as 4. The set would be represented as below.

Initially:

```

maxVal = 10 // Size of sparse
capacity = 4 // Size of dense
n = 2 // Current number of elements in set

```

```

// dense[] Stores actual elements
dense[] = {3, 5, _, _}

```

```

// Uses actual elements as index and stores
// indexes of dense[]
sparse[] = {_, _, 0, 1, _, _, _, _}

```

'_' means it can be any value and not used in sparse set

Insert 7:

```

n = 3
dense[] = {3, 5, 7, _}
sparse[] = {_, _, 0, 1, 1, 2, _, _}

```

Insert 4:

```

n = 4
dense[] = {3, 5, 7, 4}
sparse[] = {_, _, 0, 3, 1, 1, 2, _}

```

Delete 3:

```

n = 3
dense[] = {4, 5, 7, _}
sparse[] = {_, _, 0, 1, 1, 2, _}

```

Clear (Remove All):

```

n = 0
dense[] = {_, _, _, _}
sparse[] = {_, _, _, _, _, _, _}

```

Below is C++ implementation of above functions.

```

/* A C program to implement Sparse Set and its operations */
#include<bits/stdc++.h>
using namespace std;

// A structure to hold the three parameters required to
// represent a sparse set.
class SSet
{
    int *sparse; // To store indexes of actual elements
    int *dense; // To store actual set elements
    int n; // Current number of elements
    int capacity; // Capacity of set or size of dense[]
    int maxVal; /* Maximum value in set or size of
    sparse[] */

public:
    // Constructor
    SSet(int maxV, int cap)
    {
        sparse = new int[(maxV+1)];
        dense = new int[cap];
        capacity = cap;
        maxVal = maxV;
        n = 0; // No elements initially
    }

    // Destructor
    ~SSet()
    {
        delete[] sparse;
        delete[] dense;
    }

    // If element is present, returns index of
    // element in dense[]. Else returns -1.
    int search(int x);

    // Inserts a new element into set
    void insert(int x);

    // Deletes an element
    void deletion(int x);

    // Prints contents of set
    void print();

    // Removes all elements from set
    void clear() { n = 0; }

    // Finds intersection of this set with s

```



```

// and returns pointer to result.
SSet* intersection(SSet &s);

// A function to find union of two sets
// Time Complexity-O(n1+n2)
SSet *setUnion(SSet &s);
};

// If x is present in set, then returns index
// of it in dense[], else returns -1.
int SSet::search(int x)
{
    // Searched element must be in range
    if (x > maxVal)
        return -1;

    // The first condition verifies that 'x' is
    // within 'n' in this set and the second
    // condition tells us that it is present in
    // the data structure.
    if (sparse[x] < n && dense[sparse[x]] == x)
        return (sparse[x]);

    // Not found
    return -1;
}

// Inserts a new element into set
void SSet::insert(int x)
{
    // Corner cases, x must not be out of
    // range, dense[] should not be full and
    // x should not already be present
    if (x > maxVal)
        return;
    if (n >= capacity)
        return;
    if (search(x) != -1)
        return;

    // Inserting into array-dense[] at index 'n'.
    dense[n] = x;

    // Mapping it to sparse[] array.
    sparse[x] = n;

    // Increment count of elements in set
    n++;
}

// A function that deletes 'x' if present in this data
// structure, else it does nothing (just returns).
// By deleting 'x', we unset 'x' from this set.
void SSet::deletion(int x)
{
    // If x is not present
    if (search(x) == -1)
        return;

    int temp = dense[n-1]; // Take an element from end
    dense[sparse[x]] = temp; // Overwrite.
    sparse[temp] = sparse[x]; // Overwrite.

    // Since one element has been deleted, we
    // decrement 'n' by 1.
    n--;
}

// prints contents of set which are also content
// of dense[]
void SSet::print()
{
    for (int i=0; i<n; i++)
        printf("%d ", dense[i]);
    printf("\n");
}

// A function to find intersection of two sets
SSet* SSet::intersection(SSet &s)
{
    // Capacity and max value of result set
    int iCap  = min(n, s.n);
    int iMaxVal = max(s.maxVal, maxVal);

    // Create result set
    SSet *result = new SSet(iMaxVal, iCap);

    // Find the smaller of two sets
    // If this set is smaller
    if (n < s.n)
    {
        // Search every element of this set in 's'.
        // If found, add it to result
        for (int i = 0; i < n; i++)
            if (s.search(dense[i]) != -1)
                result->insert(dense[i]);
    }
    else
    {
        // Search every element of 's' in this set.
        // If found, add it to result
        for (int i = 0; i < s.n; i++)
            if (search(s.dense[i]) != -1)
                result->insert(s.dense[i]);
    }

    return result;
}

// A function to find union of two sets
// Time Complexity-O(n1+n2)
SSet* SSet::setUnion(SSet &s)
{
    // Find capacity and maximum value for result
    // set.
    int uCap  = s.n + n;

```

```

int uMaxVal = max(s.maxValue, maxVal);

// Create result set
SSet *result = new SSet(uMaxVal, uCap);

// Traverse the first set and insert all
// elements of it in result.
for (int i = 0; i < n; i++)
    result->insert(dense[i]);

// Traverse the second set and insert all
// elements of it in result (Note that sparse
// set doesn't insert an entry if it is already
// present)
for (int i = 0; i < s.n; i++)
    result->insert(s.dense[i]);

return result;
}

// Driver program
int main()
{
    // Create a set set1 with capacity 5 and max
    // value 100
    SSet s1(100, 5);

    // Insert elements into the set set1
    s1.insert(5);
    s1.insert(3);
    s1.insert(9);
    s1.insert(10);

    // Printing the elements in the data structure.
    printf("The elements in set1 are\n");
    s1.print();

    int index = s1.search(3);

    // 'index' variable stores the index of the number to
    // be searched.
    if (index != -1) // 3 exists
        printf("\n3 is found at index %d in set1\n", index);
    else // 3 doesn't exist
        printf("\n3 doesn't exists in set1\n");

    // Delete 9 and print set1
    s1.delete(9);
    s1.print();

    // Create a set with capacity 6 and max value
    // 1000
    SSet s2(1000, 6);

    // Insert elements into the set
    s2.insert(4);
    s2.insert(3);
    s2.insert(7);
    s2.insert(200);

    // Printing set 2.
    printf("\nThe elements in set2 are\n");
    s2.print();

    // Printing the intersection of the two sets
    SSet *intersect = s2.intersection(s1);
    printf("\nIntersection of set1 and set2\n");
    intersect->print();

    // Printing the union of the two sets
    SSet *unionset = s1.setUnion(s2);
    printf("\nUnion of set1 and set2\n");
    unionset->print();

    return 0;
}

```

Output :

```

The elements in set1 are
5 3 9 10

3 is found at index 1 in set1
5 3 10

The elements in set2 are-
4 3 7 200

Intersection of set1 and set2
3

Union of set1 and set2
5 3 10 4 7 200

```

Additional Operations:

The following are operations are also efficiently implemented using sparse set. It outperforms all the solutions discussed [here](#) and bit vector based solution, under the assumptions that range and maximum number of elements are known.

union():

- 1) Create an empty sparse set, say result.
- 2) Traverse the first set and insert all elements of it in result.
- 3) Traverse the second set and insert all elements of it in result (Note that sparse set doesn't insert an entry if it is already present)
- 4) Return result.

intersection():

- 1) Create an empty sparse set, say result.
- 2) Let the smaller of two given sets be first set and larger be second.
- 3) Consider the smaller set and search every element of it in second. If element is found, add it to result.
- 4) Return result.

A common use of this data structure is with register allocation algorithms in compilers, which have a fixed universe(the number of registers in the machine) and are updated and cleared frequently (just like- Q queries) during a single processing run.

References:

<http://research.swtch.com/sparse>

<http://codingplayground.blogspot.in/2009/03/sparse-sets-with-o1-insert-delete.html>

This article is contributed by **Rachit Belwariar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure

Centroid Decomposition of Tree

Background :

What is centroid of Tree?

Centroid of a Tree is a node which if removed from the tree would split it into a 'forest', such that any tree in the forest would have at most half the number of vertices in the original tree. Suppose there are n nodes in the tree. 'Subtree size' for a node is the size of the tree rooted at the node.

Let $S(v)$ be size of subtree rooted at node v

$$S(v) = 1 + \sum S(u)$$

Here u is a child to v (adjacent and at a depth one greater than the depth of v).

Centroid is a node v such that,

$$\max(n - S(v), S(u_1), S(u_2), \dots, S(u_m)) \leq n/2$$

where u_i is i 'th child to v .

Finding the centroid

Let T be an undirected tree with n nodes. Choose any arbitrary node v in the tree. If v satisfies the mathematical definition for the centroid, we have our centroid. Else, we know that our mathematical inequality did not hold, and from this we conclude that there exists some u adjacent to v such that $S(u) > n/2$. We make that u our new v and recurse.

centroidDecomposition1



We never revisit a node because when we decided to move away from it to a node with subtree size greater than $n/2$, we sort of declared that it now belongs to the component with nodes less than $n/2$, and we shall never find our centroid there.

In any case we are moving towards the centroid. Also, there are finitely many vertices in the tree. The process must stop, and it will, at the desired vertex.

Algorithm :

1. Select arbitrary node v
2. Start a DFS from v , and setup subtree sizes
3. Re-position to node v (or start at any arbitrary v that belongs to the tree)
4. Check mathematical condition of centroid for v
 1. If condition passed, return current node as centroid
 2. Else move to adjacent node with 'greatest' subtree size, and back to step 4

Theorem: Given a tree with n nodes, the centroid always exists.

Proof: Clear from our approach to the problem that we can always find a centroid using above steps.

Time Complexity

1. Select arbitrary node v : $O(1)$
2. DFS: $O(n)$
3. Reposition to v : $O(1)$
4. Find centroid: $O(n)$

Centroid Decomposition :

Finding the centroid for a tree is a part of what we are trying to achieve here. We need to think how can we organize the tree into a structure that decreases the complexity for answering certain 'type' of queries.

Algorithm

1. Make the centroid as the root of a new tree (which we will call as the 'centroid tree')
2. Recursively decompose the trees in the resulting forest
3. Make the centroids of these trees as children of the centroid which last split them.

The centroid tree has depth $O(\lg n)$, and can be constructed in $O(n \lg n)$, as we can find the centroid in $O(n)$.

Illustrative Example

Let us consider a tree with 16 nodes. The figure has subtree sizes already set up using a DFS from node 1.

centroidDecompo2



We start at node 1 and see if condition for centroid holds. Remember $S(v)$ is subtree size for v .

cd3



We make node 6 as the root of our centroid, and recurse on the 3 trees of the forest centroid split the original tree into.

NOTE: In the figure, subtrees generated by a centroid have been surrounded by a dotted line of the same color as the color of centroid.

cd4



We make the subsequently found centroids as the children to centroid that split them last, and obtain our centroid tree.

cd5



NOTE: The trees containing only a single element have the same element as their centroid. We haven't used color differentiation for such trees, and the leaf nodes represent them.

```
// C++ program for centroid decomposition of Tree
#include <bits/stdc++.h>
using namespace std;

#define MAXN 1025

vector<int> tree[MAXN];
vector<int> centroidTree[MAXN];
bool centroidMarked[MAXN];

/* method to add edge between to nodes of the undirected tree */
void addEdge(int u, int v)
{
    tree[u].push_back(v);
    tree[v].push_back(u);
}

/* method to setup subtree sizes and nodes in current tree */
void DFS(int src, bool visited[], int subtree_size[], int* n)
{
    /* mark node visited */
    visited[src] = true;

    /* increase count of nodes visited */
    *n += 1;

    /* initialize subtree size for current node */
    subtree_size[src] = 1;

    vector<int>::iterator it;

    /* recur on non-visited and non-centroid neighbours */
    for (it = tree[src].begin(); it != tree[src].end(); it++)
        if (!visited[*it] && !centroidMarked[*it])
        {
            DFS(*it, visited, subtree_size, n);
            subtree_size[src] += subtree_size[*it];
        }
}

int getCentroid(int src, bool visited[], int subtree_size[], int n)
{
    /* assume the current node to be centroid */
    bool is_centroid = true;

    /* mark it as visited */
    visited[src] = true;

    /* track heaviest child of node, to use in case node is
    not centroid */
    int heaviest_child = 0;

    vector<int>::iterator it;
```

```

/* iterate over all adjacent nodes which are children
(not visited) and not marked as centroid to some
subtree */
for (it = tree[src].begin(); it!=tree[src].end(); it++)
if (!visited[*it] && !centroidMarked[*it])
{
    /* If any adjacent node has more than n/2 nodes,
    * current node cannot be centroid */
    if (subtree_size[*it]>n/2)
        is_centroid=false;

    /* update heaviest child */
    if (heaviest_child==0 ||
        subtree_size[*it]>subtree_size[heaviest_child])
        heaviest_child = *it;
}

/* if current node is a centroid */
if (is_centroid && n-subtree_size[src]<=n/2)
    return src;

/* else recur on heaviest child */
return getCentroid(heaviest_child, visited, subtree_size, n);
}

/* function to get the centroid of tree rooted at src.
* tree may be the original one or may belong to the forest */
int getCentroid(int src)
{
    bool visited[MAXN];

    int subtree_size[MAXN];

    /* initialize auxiliary arrays */
    memset(visited, false, sizeof visited);
    memset(subtree_size, 0, sizeof subtree_size);

    /* variable to hold number of nodes in the current tree */
    int n = 0;

    /* DFS to set up subtree sizes and nodes in current tree */
    DFS(src, visited, subtree_size, &n);

    for (int i=1; i<MAXN; i++)
        visited[i] = false;

    int centroid = getCentroid(src, visited, subtree_size, n);

    centroidMarked[centroid]=true;

    return centroid;
}

/* function to generate centroid tree of tree rooted at src */
int decomposeTree(int root)
{
    //printf("decomposeTree(%d)\n", root);

    /* get centroid for current tree */
    int cend_tree = getCentroid(root);

    printf("%d ", cend_tree);

    vector<int>::iterator it;

    /* for every node adjacent to the found centroid
    * and not already marked as centroid */
    for (it=tree[cend_tree].begin(); it!=tree[cend_tree].end(); it++)
    {
        if (!centroidMarked[*it])
        {
            /* decompose subtree rooted at adjacent node */
            int cend_subtree = decomposeTree(*it);

            /* add edge between tree centroid and centroid of subtree */
            centroidTree[cend_tree].push_back(cend_subtree);
            centroidTree[cend_subtree].push_back(cend_tree);
        }
    }

    /* return centroid of tree */
    return cend_tree;
}

// driver function
int main()
{
    /* number of nodes in the tree */
    int n = 16;

    /* arguments in order: node u, node v
    * sequencing starts from 1 */
    addEdge(1, 4);
    addEdge(2, 4);
    addEdge(3, 4);
    addEdge(4, 5);
    addEdge(5, 6);
    addEdge(6, 7);
    addEdge(7, 8);
    addEdge(7, 9);
    addEdge(6, 10);
    addEdge(10, 11);
    addEdge(11, 12);
    addEdge(11, 13);
    addEdge(12, 14);
    addEdge(13, 15);
    addEdge(13, 16);

    /* generates centroid tree */
    decomposeTree(1);

    return 0;
}

```

Output :

6 4 1 2 3 5 7 8 9 11 10 12 14 13 15 16

Application:

Consider below example problem

Given a weighted tree with N nodes, find the minimum number of edges in a path of length K, or return -1 if such a path does not exist.

1
Brute force solution: For every node, perform DFS to find distance and number of edges to every other node

Time complexity: $O(N^2)$ Obviously inefficient because $N = 200000$

We can solve above problem in $O(N \log N)$ time using **Centroid Decomposition**.

1. Perform centroid decomposition to get a "tree of subtrees"
 2. Start at the root of the decomposition, solve the problem for each subtree as follows
 1. Solve the problem for each "child tree" of the current subtree.
 2. Perform DFS from the centroid on the current subtree to compute the minimum edge count for paths that include the centroid
 1. Two cases: centroid at the end or in the middle of path
- Use a timestamped array of size 1000000 to keep track of which distances from centroid are possible and the minimum edge count for that distance

Take the minimum of the above two

Time complexity of centroid decomposition based solution is $O(n \log n)$

Reference :

<http://www.ugrad.cs.ubc.ca/~cs490/2014W2/pdf/jason.pdf>

This article is contributed by **Yash Varyani**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner

Company Wise Coding Practice

Gomory-Hu Tree | Set 1 (Introduction)

Background :

In a flow network, an s-t cut is a cut that requires the source 's' and the sink 't' to be in different subsets, and it consists of edges going from the source's side to the sink's side. The capacity of an s-t cut is defined by the sum of capacity of each edge in the cut-set. (Source: [Wiki](#)). Given a two vertices, s and t, we can find [minimum s-t cut using max flow algorithm](#).

Since there are total $O(n^2)$ possible pairs, at first look it seems that there would be total $O(n^2)$ total minimum s-t cut values. But when we use Gomory-Hu Tree, we would see that there are total n-1 different cut values [A Tree with n vertices has n-1 edges]

Popular graph problems that can be solved using Gomory-Hu Tree :

1. Given a weighted and connected graph, find [minimum s-t cut](#) for all pairs of vertices. Or a problem like find minimum of all possible minimum s-t cuts.
2. [Minimum K-Cut problem](#) : Find minimum weight set of edges whose removal would partition the graph to k connected components. This is a NP-Hard problem, Gomory-Hu Tree provides an approximate solution for this problem.

What is Gomory-Hu Tree?

A Gomory-Hu Tree is defined for a flow graph with edge capacity function c. The tree has same set of vertices as input graph and has n-1 (n is number of vertices) edges. Edge capacity function c' is defined using following properties:

Equivalent flow tree : For any pair of vertices s and t, the minimum s-t cut in graph is equal to the smallest capacity of the edges on the path between s and t in Tree.

Cut property : a minimum s-t cut in Tree is also a minimum cut in Graph.G

For example, consider the following Graph and Corresponding Gomory-Hu Tree.

gomory1



Since there are $n-1$ edges in a tree with n nodes, we can conclude that there are at most $n-1$ different flow values in a flow network with n vertices.

We will be discussing construction of Tree in next post.

How to solve above problems using Gomory-Hu Tree is constructed?

The minimum weight edge in the tree is minimum of all $s-t$ cuts.

We can solve the k -cut problem using below steps.

- 1) Construct Gomory-Hu Tree.
- 2) Remove $k-1$ minimum weight (or lightest) edges from the Tree.
- 3) Return union of components obtained by above removal of edges.

Below diagram illustrates above algorithm.

GomoryHu



Note that the above solution may not always produce optimal result, but it is guaranteed to produce results within bounds of $(2-2/k)$.

References:

<https://www.corelab.ntua.gr/seminar/material/2008-2009/2008.10.20.Gomory-Hu%20trees%20and%20applications.slides.pdf>

https://courses.engr.illinois.edu/cs598csc/sp2009/lectures/lecture_7.pdf

<https://cseweb.ucsd.edu/classes/fa06/cse202/Gomory-Hu%20Tree.pdf>

This article is contributed by **Dheeraj Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Max-Flow

Given an array $A[]$ and a number x , check for pair in $A[]$ with sum as x

Write a C program that, given an array $A[]$ of n numbers and another number x , determines whether or not there exist two elements in S whose sum is exactly x .

We strongly recommend that you click here and practice it, before moving on to the solution.

METHOD 1 (Use Sorting)

Algorithm:

```
hasArrayTwoCandidates (A[], ar_size, sum)
1) Sort the array in non-decreasing order.
2) Initialize two index variables to find the candidate
   elements in the sorted array.
   (a) Initialize first to the leftmost index: l = 0
   (b) Initialize second the rightmost index: r = ar_size-1
3) Loop while l
```

Time Complexity: Depends on what sorting algorithm we use. If we use Merge Sort or Heap Sort then $(-)(n \log n)$ in worst case. If we use Quick Sort then $O(n^2)$ in worst case.

Auxiliary Space : Again, depends on sorting algorithm. For example auxiliary space is $O(n)$ for merge sort and $O(1)$ for Heap Sort.

Example:

Let Array be {1, 4, 45, 6, 10, -8} and sum to find be 16

Sort the array

$A = \{-8, 1, 4, 6, 10, 45\}$

Initialize $l = 0$, $r = 5$

$A[l] + A[r] (-8 + 45) > 16 \Rightarrow$ decrement r . Now $r = 10$

$A[l] + A[r] (-8 + 10)$ increment l . Now $l = 1$

$A[l] + A[r] (1 + 10)$ increment l . Now $l = 2$

$A[l] + A[r] (4 + 10)$ increment l . Now $l = 3$

$A[l] + A[r] (6 + 10) == 16 \Rightarrow$ Found candidates (return 1)

Note: If there are more than one pair having the given sum then this algorithm reports only one. Can be easily extended for this though.

Implementation:

C

```
# include <stdio.h>
# define bool int

void quickSort(int *, int, int);

bool hasArrayTwoCandidates(int A[], int arr_size, int sum)
{
    int l, r;

    /* Sort the elements */
    quickSort(A, 0, arr_size-1);

    /* Now look for the two candidates in the sorted
    array */
    l = 0;
    r = arr_size-1;
    while (l < r)
    {
        if(A[l] + A[r] == sum)
            return 1;
        else if(A[l] + A[r] < sum)
            l++;
        else // A[l] + A[r] > sum
            r--;
    }
    return 0;
}

/* Driver program to test above function */
int main()
{
    int A[] = {1, 4, 45, 6, 10, -8};
    int n = 16;
    int arr_size = 6;

    if( hasArrayTwoCandidates(A, arr_size, n))
        printf("Array has two elements with sum 16");
    else
        printf("Array doesn't have two elements with sum 16 ");

    getchar();
    return 0;
}

/* FOLLOWING FUNCTIONS ARE ONLY FOR SORTING
PURPOSE */
void exchange(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}

int partition(int A[], int si, int ei)
{
    int x = A[ei];
    int i = (si - 1);
    int j;

    for (j = si; j <= ei - 1; j++)
    {
        if(A[j] <= x)
        {
            i++;
            exchange(&A[i], &A[j]);
        }
    }
    exchange (&A[i + 1], &A[ei]);
    return (i + 1);
}

/* Implementation of Quick Sort
A[] --> Array to be sorted
si --> Starting index
ei --> Ending index
*/
void quickSort(int A[], int si, int ei)
{
    int pi; /* Partitioning index */
    if(si < ei)
    {
        pi = partition(A, si, ei);
        quickSort(A, si, pi - 1);
        quickSort(A, pi + 1, ei);
    }
}
```

Python


```
# Python program to check for the sum condition to be satisfied
def hasArrayTwoCandidates(A,arr_size,sum):
```

```
    # sort the array
    quickSort(A,0,arr_size-1)
    l = 0
    r = arr_size-1

    # traverse the array for the two elements
    while l<r:
        if (A[l] + A[r] == sum):
            return 1
        elif (A[l] + A[r] < sum):
            l += 1
        else:
            r -= 1
    return 0

# Implementation of Quick Sort
# A[] --> Array to be sorted
# si --> Starting index
# ei --> Ending index
def quickSort(A, si, ei):
    if si < ei:
        pi=partition(A,si,ei)
        quickSort(A,si,pi-1)
        quickSort(A,pi+1,ei)

# Utility function for partitioning the array(used in quick sort)
def partition(A, si, ei):
    x = A[ei]
    i = (si-1)
    for j in range(si,ei):
        if A[j] <= x:
            i += 1

    # This operation is used to swap two variables in python
    A[i], A[j] = A[j], A[i]

    A[i+1], A[ei] = A[ei], A[i+1]

    return i+1

# Driver program to test the functions
A = [1,4,45,6,10,-8]
n = 16
if (hasArrayTwoCandidates(A, len(A), n)):
    print("Array has two elements with the given sum")
else:
    print("Array doesn't have two elements with the given sum")

## This code is contributed by __Devesh Agrawal__
```

Output:

```
Array has two elements with the given sum
```

METHOD 2 (Use Hash Map)

Thanks to Bindu for suggesting this method and thanks to Shekhu for providing code.

This method works in O(n) time if range of numbers is known.

Let sum be the given sum and A[] be the array in which we need to find pair.

- 1) Initialize Binary Hash Map M[] = {0, 0, ...}
- 2) Do following for each element A[i] in A[]
 - (a) If M[x - A[i]] is set then print the pair (A[i], x - A[i])
 - (b) Set M[A[i]]

Implementation:

C/C++

```
#include <stdio.h>
#define MAX 100000

void printPairs(int arr[], int arr_size, int sum)
{
    int i, temp;
    bool binMap[MAX] = {0}; /*initialize hash map as 0*/

    for (i = 0; i < arr_size; i++)
    {
        temp = sum - arr[i];
        if (temp >= 0 && binMap[temp] == 1)
            printf("Pair with given sum %d is (%d, %d) \n",
                sum, arr[i], temp);
        binMap[arr[i]] = 1;
    }
}

/* Driver program to test above function */
int main()
{
    int A[] = {1, 4, 45, 6, 10, 8};
    int n = 16;
```

```

int arr_size = sizeof(A)/sizeof(A[0]);

printPairs(A, arr_size, n);

getchar();
return 0;
}

```

Java

```

// Java implementation using Hashing
import java.io.*;

class PairSum
{
    private static final int MAX = 100000; // Max size of Hashmap

    static void printpairs(int arr[],int sum)
    {
        // Declares and initializes the whole array as false
        boolean[] binmap = new boolean[MAX];

        for (int i=0; i<arr.length; ++i)
        {
            int temp = sum-arr[i];

            // checking for condition
            if (temp>=0 && binmap[temp])
            {
                System.out.println("Pair with given sum " +
                    sum + " is (" + arr[i] +
                    ", "+temp+")");
            }
            binmap[arr[i]] = true;
        }
    }

    // Main to test the above function
    public static void main (String[] args)
    {
        int A[] = {1, 4, 45, 6, 10, 8};
        int n = 16;
        printpairs(A, n);
    }
}

// This article is contributed by Aakash Hasija

```

Python

```

# Python program to find if there are two elements with given sum
CONST_MAX = 100000

# function to check for the given sum in the array
def printPairs(arr, arr_size, sum):

    # initialize hash map as 0
    binmap = [0]*CONST_MAX

    for i in range(0,arr_size):
        temp = sum-arr[i]
        if (temp>=0 and binmap[temp]==1):
            print "Pair with the given sum is", arr[i], "and", temp
            binmap[arr[i]]=1

# driver program to check the above function
A = [1,4,45,6,10,-8]
n = 16
printPairs(A, len(A), n)

# This code is contributed by __Devesh Agrawal__

```

Time Complexity: $O(n)$

Output:

```

Pair with given sum 16 is (10, 6)

```

Auxiliary Space: $O(R)$ where R is range of integers.

If range of numbers include negative numbers then also it works. All we have to do for negative numbers is to make everything positive by adding the absolute value of smallest negative integer to all numbers.

Please write comments if you find any of the above codes/algorithms incorrect, or find other ways to solve the same problem.

GATE CS Corner

Company Wise Coding Practice

Arrays
Hash
Amazon-Question
CareWale-Question
Hashing

Majority Element

Majority Element: A majority element in an array $A[]$ of size n is an element that appears more than $n/2$ times (and hence there is at most one such element).

Write a function which takes an array and emits the majority element (if it exists), otherwise prints NONE as follows:

I/P : 3 3 4 2 4 4 2 4 4
O/P : 4

I/P : 3 3 4 2 4 4 2 4
O/P : NONE

METHOD 1 (Basic)

The basic solution is to have two loops and keep track of maximum count for all different elements. If maximum count becomes greater than $n/2$ then break the loops and return the element having maximum count. If maximum count doesn't become more than $n/2$ then majority element doesn't exist.

Time Complexity: $O(n^2)$.

Auxiliary Space : $O(1)$.

METHOD 2 (Using Binary Search Tree)

Thanks to Sachin Midha for suggesting this solution. Node of the Binary Search Tree (used in this approach) will be as follows.

```
struct tree
{
    int element;
    int count;
}BST;
```

Insert elements in BST one by one and if an element is already present then increment the count of the node. At any stage, if count of a node becomes more than $n/2$ then return.

The method works well for the cases where $n/2+1$ occurrences of the majority element is present in the starting of the array, for example {1, 1, 1, 1, 1, 2, 3, 4}.

Time Complexity: If a binary search tree is used then time complexity will be $O(n^2)$. If a [self-balancing-binary-search tree](#) is used then $O(n \log n)$

Auxiliary Space: $O(n)$

METHOD 3 (Using Moore's Voting Algorithm)

This is a two step process.

1. Get an element occurring most of the time in the array. This phase will make sure that if there is a majority element then it will return that only.
2. Check if the element obtained from above step is majority element.

1. Finding a Candidate:

The algorithm for first phase that works in $O(n)$ is known as Moore's Voting Algorithm. Basic idea of the algorithm is if we cancel out each occurrence of an element e with all the other elements that are different from e then e will exist till end if it is a majority element.

```
findCandidate(a[], size)
1. Initialize index and count of majority element
   maj_index = 0, count = 1
2. Loop for i = 1 to size - 1
   (a) If a[maj_index] == a[i]
       count++
   (b) Else
```

```

count--;
(c) If count == 0
    maj_index = i;
    count = 1
3. Return a[maj_index]

```

Above algorithm loops through each element and maintains a count of a[maj_index]. If next element is same then increments the count, if next element is not same then decrements the count, and if the count reaches 0 then changes the maj_index to the current element and sets count to 1.

First Phase algorithm gives us a candidate element. In second phase we need to check if the candidate is really a majority element. Second phase is simple and can be easily done in O(n). We just need to check if count of the candidate element is greater than n/2.

Example:

A[] = 2, 2, 3, 5, 2, 2, 6

Initialize:

maj_index = 0, count = 1 → candidate '2'?

2, 2, 3, 5, 2, 2, 6

Same as a[maj_index] => count = 2

2, 2, 3, 5, 2, 2, 6

Different from a[maj_index] => count = 1

2, 2, 3, 5, 2, 2, 6

Different from a[maj_index] => count = 0

Since count = 0, change candidate for majority element to 5 => maj_index = 3, count = 1

2, 2, 3, 5, 2, 2, 6

Different from a[maj_index] => count = 0

Since count = 0, change candidate for majority element to 2 => maj_index = 4

2, 2, 3, 5, 2, 2, 6

Same as a[maj_index] => count = 2

2, 2, 3, 5, 2, 2, 6

Different from a[maj_index] => count = 1

Finally candidate for majority element is 2.

First step uses Moore's Voting Algorithm to get a candidate for majority element.

2. Check if the element obtained in step 1 is majority

```

printMajority(a[], size)
1. Find the candidate for majority
2. If candidate is majority. i.e., appears more than n/2 times.
   Print the candidate
3. Else
   Print "NONE"

```

Implementation of method 3:

C

```

/* Program for finding out majority element in an array */
#include<stdio.h>
#define bool int

int findCandidate(int *, int);
bool isMajority(int *, int, int);

/* Function to print Majority Element */
void printMajority(int a[], int size)
{
    /* Find the candidate for Majority */
    int cand = findCandidate(a, size);

    /* Print the candidate if it is Majority */
    if (isMajority(a, size, cand))
        printf("%d ", cand);
    else
        printf("No Majority Element");
}

/* Function to find the candidate for Majority */
int findCandidate(int a[], int size)
{
    int maj_index = 0, count = 1;
    int i;
    for (i = 1; i < size; i++)
    {
        if (a[maj_index] == a[i])
            count++;
        else
            count--;
        if (count == 0)
        {
            maj_index = i;
            count = 1;
        }
    }
    return a[maj_index];
}

/* Function to check if the candidate occurs more than n/2 times */
bool isMajority(int a[], int size, int cand)
{
    int i, count = 0;
    for (i = 0; i < size; i++)
        if (a[i] == cand)
            count++;
    if (count > size/2)
        return 1;
    else
        return 0;
}

/* Driver function to test above functions */
int main()
{
    int a[] = {1, 3, 3, 1, 2};
    int size = (sizeof(a))/sizeof(a[0]);
    printMajority(a, size);
}

```

```
getchar();
return 0;
}
```

Java

```
/* Program for finding out majority element in an array */

class MajorityElement
{
    /* Function to print Majority Element */
    void printMajority(int a[], int size)
    {
        /* Find the candidate for Majority*/
        int cand = findCandidate(a, size);

        /* Print the candidate if it is Majority*/
        if (isMajority(a, size, cand))
            System.out.println(" " + cand + " ");
        else
            System.out.println("No Majority Element");
    }

    /* Function to find the candidate for Majority */
    int findCandidate(int a[], int size)
    {
        int maj_index = 0, count = 1;
        int i;
        for (i = 1; i < size; i++)
        {
            if (a[maj_index] == a[i])
                count++;
            else
                count--;
            if (count == 0)
            {
                maj_index = i;
                count = 1;
            }
        }
        return a[maj_index];
    }

    /* Function to check if the candidate occurs more
    than n/2 times */
    boolean isMajority(int a[], int size, int cand)
    {
        int i, count = 0;
        for (i = 0; i < size; i++)
        {
            if (a[i] == cand)
                count++;
        }
        if (count > size / 2)
            return true;
        else
            return false;
    }

    /* Driver program to test the above functions */
    public static void main(String[] args)
    {
        MajorityElement majorelement = new MajorityElement();
        int a[] = new int[]{1, 3, 3, 1, 2};
        int size = a.length;
        majorelement.printMajority(a, size);
    }
}

// This code has been contributed by Mayank Jaiswal
```

Output:

No Majority Element

Time Complexity: $O(n)$

Auxiliary Space : $O(1)$

Now give a try to below question

Given an array of $2n$ elements of which n elements are same and the remaining n elements are all different. Write a C program to find out the value which is present n times in the array. There is no restriction on the elements in the array. They are random (In particular they not sequential).

GATE CS Corner Company Wise Coding Practice

Arrays
Majority Element
Moore's Voting Algorithm

Find the Number Occurring Odd Number of Times

Given an array of positive integers. All numbers occur even number of times except one number which occurs odd number of times. Find the number in $O(n)$ time & constant space.

Example:

I/P = [1, 2, 3, 2, 3, 1, 3]

O/P = 3

A **Simple Solution** is to run two nested loops. The outer loop picks all elements one by one and inner loop counts number of occurrences of the element picked by outer loop. Time complexity of this solution is $O(n^2)$.

A **Better Solution** is to use Hashing. Use array elements as key and their counts as value. Create an empty hash table. One by one traverse the given array elements and store counts. Time complexity of this solution is $O(n)$. But it requires extra space for hashing.

The **Best Solution** is to do bitwise XOR of all the elements. XOR of all elements gives us odd occurring element. Please note that XOR of two elements is 0 if both elements are same and XOR of a number x with 0 is x .

Below are implementations of this best approach.

Program:

C/C++

```
//C program to find the element occurring odd number of times

#include <stdio.h>
int getOddOccurance(int ar[], int ar_size)
{
    int i;
    int res = 0;
    for (i=0; i < ar_size; i++)
        res = res ^ ar[i];

    return res;
}

/* Driver function to test above function */
int main()
{
    int ar[] = {2, 3, 5, 4, 5, 2, 4, 3, 5, 2, 4, 4, 2};
    int n = sizeof(ar)/sizeof(ar[0]);
    printf("%d", getOddOccurance(ar, n));
    return 0;
}
```

Java

```
//Java program to find the element occurring odd number of times

class OddOccurance
{
    int getOddOccurance(int ar[], int ar_size)
    {
        int i;
        int res = 0;
        for (i = 0; i < ar_size; i++)
        {
            res = res ^ ar[i];
        }
        return res;
    }

    public static void main(String[] args)
    {
        OddOccurance occur = new OddOccurance();
        int ar[] = new int[] {2, 3, 5, 4, 5, 2, 4, 3, 5, 2, 4, 4, 2};
        int n = ar.length;
        System.out.println(occur.getOddOccurance(ar, n));
    }
}

// This code has been contributed by Mayank Jaiswal
```

Python

```
# Python program to find the element occurring odd number of times

def getOddOccurance(arr):

    # Initialize result
    res = 0

    # Traverse the array
    for element in arr:
        # XOR with the result
        res = res ^ element

    return res

# Test array
arr = [ 2, 3, 5, 4, 5, 2, 4, 3, 5, 2, 4, 4, 2]

print "%d" % getOddOccurance(arr)
```

Output:

5

Time Complexity: $O(n)$

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Arrays
Bit Magic
Bit Magic
XOR

Largest Sum Contiguous Subarray

Write an efficient C program to find the sum of contiguous subarray within a one-dimensional array of numbers which has the largest sum.

kadane-algorithm



Kadane's Algorithm:

Initialize:

```
max_so_far = 0
max_ending_here = 0
```

Loop for each element of the array

```
(a) max_ending_here = max_ending_here + a[i]
(b) if(max_ending_here
```

Explanation:

Simple idea of the Kadane's algorithm is to look for all positive contiguous segments of the array (max_ending_here is used for this). And keep track of maximum sum contiguous segment among all positive segments (max_so_far is used for this).

Lets take the example:

```
{-2, -3, 4, -1, -2, 1, 5, -3}
```

```
max_so_far = max_ending_here = 0
```

```
for i=0, a[0] = -2
```

```
max_ending_here = max_ending_here + (-2)
```

```
Set max_ending_here = 0 because max_ending_here
```

Program:

C++

```
// C++ program to print largest contiguous array sum
#include<iostream>
#include<climits>
using namespace std;

int maxSubArraySum(int a[], int size)
{
    int max_so_far = INT_MIN, max_ending_here = 0;

    for (int i = 0; i < size; i++)
    {
        max_ending_here = max_ending_here + a[i];
        if (max_so_far < max_ending_here)
            max_so_far = max_ending_here;

        if (max_ending_here < 0)
            max_ending_here = 0;
    }
    return max_so_far;
}

/*Driver program to test maxSubArraySum*/
int main()
{
    int a[] = {-2, -3, 4, -1, -2, 1, 5, -3};
```

```

int n = sizeof(a)/sizeof(a[0]);
int max_sum = maxSubArraySum(a, n);
cout << "Maximum contiguous sum is " << max_sum;
return 0;
}

```

Java

```

import java.io.*;
// Java program to print largest contiguous array sum
import java.util.*;

class Kadane
{
    public static void main (String[] args)
    {
        int [] a = {-2, -3, 4, -1, -2, 1, 5, -3};
        System.out.println("Maximum contiguous sum is " +
            maxSubArraySum(a));
    }

    static int maxSubArraySum(int a[])
    {
        int size = a.length;
        int max_so_far = Integer.MIN_VALUE, max_ending_here = 0;

        for (int i = 0; i < size; i++)
        {
            max_ending_here = max_ending_here + a[i];
            if (max_so_far < max_ending_here)
                max_so_far = max_ending_here;
            if (max_ending_here < 0)
                max_ending_here = 0;
        }
        return max_so_far;
    }
}

```

Python

```

# Python program to find maximum contiguous subarray

# Function to find the maximum contiguous subarray
from sys import maxint
def maxSubArraySum(a,size):

    max_so_far = -maxint - 1
    max_ending_here = 0

    for i in range(0, size):
        max_ending_here = max_ending_here + a[i]
        if (max_so_far < max_ending_here):
            max_so_far = max_ending_here

        if max_ending_here < 0:
            max_ending_here = 0
    return max_so_far

# Driver function to check the above function
a = [-13, -3, -25, -20, -3, -16, -23, -12, -5, -22, -15, -4, -7]
print "Maximum contiguous sum is", maxSubArraySum(a,len(a))

#This code is contributed by _Devesh Agrawal_

```

Output:

```
Maximum contiguous sum is 7
```

Above program can be optimized further, if we compare max_so_far with max_ending_here only if max_ending_here is greater than 0.

C++

```

int maxSubArraySum(int a[], int size)
{
    int max_so_far = 0, max_ending_here = 0;
    for (int i = 0; i < size; i++)
    {
        max_ending_here = max_ending_here + a[i];
        if (max_ending_here < 0)
            max_ending_here = 0;

        /* Do not compare for all elements. Compare only
        when max_ending_here > 0 */
        else if (max_so_far < max_ending_here)
            max_so_far = max_ending_here;
    }
    return max_so_far;
}

```


Python

```
def maxSubArraySum(a,size):

    max_so_far = 0
    max_ending_here = 0

    for i in range(0, size):
        max_ending_here = max_ending_here + a[i]
        if max_ending_here < 0:
            max_ending_here = 0

        # Do not compare for all elements. Compare only
        # when max_ending_here > 0
        elif (max_so_far < max_ending_here):
            max_so_far = max_ending_here

    return max_so_far
```

Time Complexity: $O(n)$

Algorithmic Paradigm: Dynamic Programming

Following is another simple implementation suggested by **Mohit Kumar**. The implementation handles the case when all numbers in array are negative.

C++

```
#include<iostream>
using namespace std;

int maxSubArraySum(int a[], int size)
{
    int max_so_far = a[0];
    int curr_max = a[0];

    for (int i = 1; i < size; i++)
    {
        curr_max = max(a[i], curr_max+a[i]);
        max_so_far = max(max_so_far, curr_max);
    }
    return max_so_far;
}

/* Driver program to test maxSubArraySum */
int main()
{
    int a[] = {-2, -3, 4, -1, -2, 1, 5, -3};
    int n = sizeof(a)/sizeof(a[0]);
    int max_sum = maxSubArraySum(a, n);
    cout << "Maximum contiguous sum is " << max_sum;
    return 0;
}
```

Python

```
# Python program to find maximum contiguous subarray

def maxSubArraySum(a,size):

    max_so_far=a[0]
    curr_max = a[0]

    for i in range(1,size):
        curr_max = max(a[i], curr_max + a[i])
        max_so_far = max(max_so_far,curr_max)

    return max_so_far

# Driver function to check the above function
a = [-2,-3,4,-1,-2,1,5,-3]
print"Maximum contiguous sum is" , maxSubArraySum(a,len(a))

#This code is contributed by _Devesh Agrawal_
```

Output:

```
Maximum contiguous sum is 7
```

To print the subarray with the maximum sum, we maintain indices whenever we get the maximum sum.

```
// C++ program to print largest contiguous array sum
#include<iostream>
```

```
#include<climits>
using namespace std;

int maxSubArraySum(int a[], int size)
{
    int max_so_far = INT_MIN, max_ending_here = 0,
        start = 0, end = 0, s=0;

    for (int i=0; i< size; i++)
    {
        max_ending_here += a[i];

        if (max_so_far < max_ending_here)
        {
            max_so_far = max_ending_here;
            start = s;
            end = i;
        }

        if (max_ending_here < 0)
        {
            max_ending_here = 0;
            s = i+1;
        }
    }
    cout << "Maximum contiguous sum is "
        << max_so_far << endl;
    cout << "Starting index " << start
        << endl << "Ending index " << end << endl;
}

/*Driver program to test maxSubArraySum*/
int main()
{
    int a[] = {-2, -3, 4, -1, -2, 1, 5, -3};
    int n = sizeof(a)/sizeof(a[0]);
    int max_sum = maxSubArraySum(a, n);
    return 0;
}
```

Output:

```
Maximum contiguous sum is 7
Starting index 2
Ending index 6
```

Now try below question

Given an array of integers (possibly some of the elements negative), write a C program to find out the "maximum product" possible by adding 'n' consecutive integers in the array. n

References:

http://en.wikipedia.org/wiki/Kadane%27s_Algorithm

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner
Company Wise Coding Practice

Find the Missing Number

You are given a list of $n-1$ integers and these integers are in the range of 1 to n . There are no duplicates in list. One of the integers is missing in the list. Write an efficient code to find the missing integer.

Example:

I/P [1, 2, 4, ,6, 3, 7, 8]
O/P 5

METHOD 1(Use sum formula)

Algorithm:

1. Get the sum of numbers
 $\text{total} = n*(n+1)/2$
2. Subtract all the numbers from sum and you will get the missing number.

```
#include<stdio.h>
```

```
/* getMissingNo takes array and size of array as arguments*/
int getMissingNo (int a[], int n)
{
    int i, total;
    total = (n+1)*(n+2)/2;
    for ( i = 0; i< n; i++)
        total -= a[i];
    return total;
}

/*program to test above function */
int main()
{
    int a[] = {1,2,4,5,6};
    int miss = getMissingNo(a,5);
    printf("%d", miss);
    getchar();
}
```

Time Complexity: $O(n)$

There can be overflow if n is large. In order to avoid Integer Overflow, we can pick one number from known numbers and subtract one number from given numbers. This way we won't have Integer Overflow ever. Thanks to Sahil Rally for suggesting this improvement.

METHOD 2(Use XOR)

- 1) XOR all the array elements, let the result of XOR be $X1$.
- 2) XOR all numbers from 1 to n , let XOR be $X2$.
- 3) XOR of $X1$ and $X2$ gives the missing number.

```
#include<stdio.h>
```

```
/* getMissingNo takes array and size of array as arguments*/
int getMissingNo(int a[], int n)
{
    int i;
    int x1 = a[0]; /* For xor of all the elements in array */
    int x2 = 1; /* For xor of all the elements from 1 to n+1 */

    for (i = 1; i< n; i++)
        x1 = x1^a[i];

    for ( i = 2; i <= n+1; i++)
        x2 = x2^i;

    return (x1^x2);
}

/*program to test above function */
int main()
{
    int a[] = {1, 2, 4, 5, 6};
    int miss = getMissingNo(a, 5);
    printf("%d", miss);
    getchar();
}
```

Time Complexity: $O(n)$

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Search an element in a sorted and rotated array

An element in a sorted array can be found in $O(\log n)$ time via binary search. But suppose we rotate an ascending order sorted array at some pivot unknown to you beforehand. So for instance, 1 2 3 4 5 might become 3 4 5 1 2. Devise a way to find an element in the rotated array in $O(\log n)$ time.

sortedPivotedArray



```
Input : arr[] = {5, 6, 7, 8, 9, 10, 1, 2, 3};
key = 3
Output : Found at index 8
```

```
Input : arr[] = {5, 6, 7, 8, 9, 10, 1, 2, 3};
key = 30
Output : Not found
```

```
Input : arr[] = {30, 40, 50, 10, 20}
key = 10
Output : Found at index 3
```

We strongly recommend that you click here and practice it, before moving on to the solution.

All solutions provided here assume that all elements in array are distinct.

The idea is to find the pivot point, divide the array in two sub-arrays and call binary search.

The main idea for finding pivot is – for a sorted (in increasing order) and pivoted array, pivot element is the only element for which next element to it is smaller than it.

Using above criteria and binary search methodology we can get pivot element in $O(\log n)$ time

```
Input arr[] = {3, 4, 5, 1, 2}
Element to Search = 1
1) Find out pivot point and divide the array in two
sub-arrays. (pivot = 2) /*Index of 5*/
2) Now call binary search for one of the two sub-arrays.
(a) If element is greater than 0th element then
search in left array
(b) Else Search in right array
(1 will go in else as 1 If element is found in selected sub-array then return index
Else return -1.
```

Implementation:

```
/* Program to search an element in a sorted and pivoted array */
#include <stdio.h>

int findPivot(int[], int, int);
int binarySearch(int[], int, int, int);

/* Searches an element key in a pivoted sorted array arr[]
of size n */
int pivotedBinarySearch(int arr[], int n, int key)
{
    int pivot = findPivot(arr, 0, n-1);

    // If we didn't find a pivot, then array is not rotated at all
    if (pivot == -1)
        return binarySearch(arr, 0, n-1, key);

    // If we found a pivot, then first compare with pivot and then
    // search in two subarrays around pivot
    if (arr[pivot] == key)
        return pivot;
    if (arr[0] <= key)
        return binarySearch(arr, 0, pivot-1, key);
    return binarySearch(arr, pivot+1, n-1, key);
}

/* Function to get pivot. For array 3, 4, 5, 6, 1, 2 it returns
3 (index of 6) */
int findPivot(int arr[], int low, int high)
{
    // base cases
    if (high < low) return -1;
    if (high == low) return low;

    int mid = (low + high)/2; /*low + (high - low)/2;*/
    if (mid < high && arr[mid] > arr[mid + 1])
        return mid;
    if (mid > low && arr[mid] < arr[mid - 1])
        return (mid-1);
    if (arr[low] >= arr[mid])
        return findPivot(arr, low, mid-1);
    return findPivot(arr, mid + 1, high);
}

/* Standard Binary Search function */
int binarySearch(int arr[], int low, int high, int key)
{
    if (high < low)
        return -1;
    int mid = (low + high)/2; /*low + (high - low)/2;*/
    if (key == arr[mid])
        return mid;
    if (key > arr[mid])
        return binarySearch(arr, (mid + 1), high, key);
    return binarySearch(arr, low, (mid - 1), key);
}

/* Driver program to check above functions */
int main()
{
    // Let us search 3 in below array
    int arr1[] = {5, 6, 7, 8, 9, 10, 1, 2, 3};
    int n = sizeof(arr1)/sizeof(arr1[0]);
    int key = 3;
    printf("Index: %d\n", pivotedBinarySearch(arr1, n, key));
}
```

```
    return 0;
}
```

Output:

Index of the element is 8

Time Complexity $O(\log n)$. Thanks to Ajay Mishra for initial solution.

Improved Solution:

We can search an element in one pass of Binary Search. The idea is to search

```
1) Find middle point mid = (l + h)/2
2) If key is present at middle point, return mid.
3) Else If arr[l..mid] is sorted
   a) If key to be searched lies in range from arr[l]
      to arr[mid], recur for arr[l..mid].
   b) Else recur for arr[mid+1..r]
4) Else (arr[mid+1..r] must be sorted)
   a) If key to be searched lies in range from arr[mid+1]
      to arr[r], recur for arr[mid+1..r].
   b) Else recur for arr[l..mid]
```

Below is C++ implementation of above idea.

```
// Search an element in sorted and rotated array using
// single pass of Binary Search
#include <bits/stdc++.h>
using namespace std;

// Returns index of key in arr[l..h] if key is present,
// otherwise returns -1
int search(int arr[], int l, int h, int key)
{
    if (l > h) return -1;

    int mid = (l+h)/2;
    if (arr[mid] == key) return mid;

    /* If arr[l..mid] is sorted */
    if (arr[l] <= arr[mid])
    {
        /* As this subarray is sorted, we can quickly
        check if key lies in half or other half */
        if (key >= arr[l] && key <= arr[mid])
            return search(arr, l, mid-1, key);

        return search(arr, mid+1, h, key);
    }

    /* If arr[l..mid] is not sorted, then arr[mid... r]
    must be sorted */
    if (key >= arr[mid] && key <= arr[h])
        return search(arr, mid+1, h, key);

    return search(arr, l, mid-1, key);
}

// Driver program
int main()
{
    int arr[] = {4, 5, 6, 7, 8, 9, 1, 2, 3};
    int n = sizeof(arr)/sizeof(arr[0]);
    int key = 6
    int i = search(arr, 0, n-1, key);
    if (i != -1) cout << "Index: " << i << endl;
    else cout << "Key not found\n";
}
```

Output:

Index: 2

Thanks to [Gaurav Ahirwar](#) for suggesting above solution.

How to handle duplicates?

It doesn't look possible to search in $O(\log n)$ time in all cases when duplicates are allowed. For example consider searching 0 in {2, 2, 2, 2, 2, 2, 2, 2, 0, 2} and {2, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2}. It doesn't look possible to decide whether to recur for left half or right half by doing constant number of comparisons at the middle.

Similar Articles:

[Find the minimum element in a sorted and rotated array](#)

[Given a sorted and rotated array, find if there is a pair with a given sum.](#)

Please write comments if you find any bug in above codes/algorithms, or find other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Searching
array
Binary-Search
rotation

Merge an array of size n into another array of size m+n

Asked by Binod

Question:

There are two sorted arrays. First one is of size m+n containing only m elements. Another one is of size n and contains n elements. Merge these two arrays into the first array of size m+n such that the output is sorted.

Input: array with m+n elements (mPlusN[]).

MergemPlusN



NA => Value is not filled/available in array mPlusN[]. There should be n such array blocks.

Input: array with n elements (N[]).

MergeN



Output: N[] merged into mPlusN[] (Modified mPlusN[])

MergemPlusN_Res



Algorithm:

Let first array be mPlusN[] and other array be N[]

- 1) Move m elements of mPlusN[] to end.
- 2) Start from nth element of mPlusN[] and 0th element of N[] and merge them into mPlusN[].

Implementation:

C/C++

```
#include <stdio.h>

/* Assuming -1 is filled for the places where element
is not available */
#define NA -1

/* Function to move m elements at the end of array mPlusN[] */
void moveToEnd(int mPlusN[], int size)
{
    int i = 0, j = size - 1;
    for (i = size-1; i >= 0; i--)
        if (mPlusN[i] != NA)
        {
            mPlusN[j] = mPlusN[i];
            j--;
        }
}

/* Merges array N[] of size n into array mPlusN[]
of size m+n */
int merge(int mPlusN[], int N[], int m, int n)
{
    int i = n; /* Current index of i/p part of mPlusN[] */
    int j = 0; /* Current index of N[] */
    int k = 0; /* Current index of output mPlusN[] */
    while (k < (m+n))
    {
        /* Take an element from mPlusN[] if
        a) value of the picked element is smaller and we have
        not reached end of it
        b) We have reached end of N[] */
        if ((i < (m+n) && mPlusN[i] <= N[j]) || (j == n))
        {
            mPlusN[k] = mPlusN[i];
            k++;
            i++;
        }
        else // Otherwise take element from N[]
        {
            mPlusN[k] = N[j];
            k++;
            j++;
        }
    }
}

/* Utility that prints out an array on a line */
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", arr[i]);

    printf("\n");
}

/* Driver function to test above functions */
int main()
{
    /* Initialize arrays */
    int mPlusN[] = {2, 8, NA, NA, NA, 13, NA, 15, 20};
    int N[] = {5, 7, 9, 25};
    int n = sizeof(N)/sizeof(N[0]);
    int m = sizeof(mPlusN)/sizeof(mPlusN[0]) - n;

    /* Move the m elements at the end of mPlusN */
    moveToEnd(mPlusN, m+n);

    /* Merge N[] into mPlusN[] */
    merge(mPlusN, N, m, n);

    /* Print the resultant mPlusN */
    printArray(mPlusN, m+n);

    return 0;
}
```

Java

```
class MergeArrays
{
    /* Function to move m elements at the end of array mPlusN[] */
    void moveToEnd(int mPlusN[], int size)
    {
        int i, j = size - 1;
        for (i = size - 1; i >= 0; i--)
        {
            if (mPlusN[i] != -1)
            {
                mPlusN[j] = mPlusN[i];
                j--;
            }
        }
    }

    /* Merges array N[] of size n into array mPlusN[]
    of size m+n */
    void merge(int mPlusN[], int N[], int m, int n)
    {
        int i = n;

        /* Current index of i/p part of mPlusN[] */
        int j = 0;

        /* Current index of N[] */
        int k = 0;

        /* Current index of output mPlusN[] */
        while (k < (m + n))
        {
            /* Take an element from mPlusN[] if
            a) value of the picked element is smaller and we have
            not reached end of it
            b) We have reached end of N[] */
            if ((i < (m + n) && mPlusN[i] <= N[j]) || (j == n))
            {
                mPlusN[k] = mPlusN[i];
                k++;
                i++;
            }
            else // Otherwise take element from N[]
            {
                mPlusN[k] = N[j];
                k++;
                j++;
            }
        }
    }

    /* Utility that prints out an array on a line */
    void printArray(int arr[], int size)
    {
        int i;
        for (i = 0; i < size; i++)
            System.out.print(arr[i] + " ");

        System.out.println("");
    }

    public static void main(String[] args)
    {
        MergeArrays mergearray = new MergeArrays();

        /* Initialize arrays */
        int mPlusN[] = {2, 8, -1, -1, -1, 13, -1, 15, 20};
        int N[] = {5, 7, 9, 25};
        int n = N.length;
        int m = mPlusN.length - n;

        /* Move the m elements at the end of mPlusN */
        mergearray.moveToEnd(mPlusN, m + n);

        /* Merge N[] into mPlusN[] */
        mergearray.merge(mPlusN, N, m, n);

        /* Print the resultant mPlusN */
        mergearray.printArray(mPlusN, m + n);
    }
}

// This code has been contributed by Mayank Jaiswal
```

Output:

```
2 5 7 8 9 13 15 20 25
```

Time Complexity: $O(m+n)$

Please write comment if you find any bug in the above program or a better way to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Arrays
array

Median of two sorted arrays

Question: There are 2 sorted arrays A and B of size n each. Write an algorithm to find the median of the array obtained after merging the above 2 arrays (i.e. array of length 2n). The complexity should be $O(\log(n))$

We strongly recommend that you click here and practice it, before moving on to the solution.

Median: In probability theory and statistics, a median is described as the number separating the higher half of a sample, a population, or a probability distribution, from the lower half. The median of a finite list of numbers can be found by arranging all the numbers from lowest value to highest value and picking the middle one.

For getting the median of input array { 12, 11, 15, 10, 20 }, first sort the array . We get { 10, 11, 12, 15, 20 } after sorting. Median is the middle element of the sorted array which is 12.

There are different conventions to take median of an array with even number of elements, one can take the mean of the two middle values, or first middle value, or second middle value.

Let us see different methods to get the median of two sorted arrays of size n each. Since size of the set for which we are looking for median is even (2n), we are taking average of middle two numbers in all below solutions.

Method 1 (Simply count while Merging)

Use merge procedure of merge sort. Keep track of count while comparing elements of two arrays. If count becomes n(For 2n elements), we have reached the median. Take the average of the elements at indexes n-1 and n in the merged array. See the below implementation.

Implementation:

```
// A Simple Merge based O(n) solution to find median of
// two sorted arrays
#include <stdio.h>

/* This function returns median of ar1[] and ar2[].
Assumptions in this function:
Both ar1[] and ar2[] are sorted arrays
Both have n elements */
int getMedian(int ar1[], int ar2[], int n)
{
    int i = 0; /* Current index of i/p array ar1[] */
    int j = 0; /* Current index of i/p array ar2[] */
    int count;
    int m1 = -1, m2 = -1;

    /* Since there are 2n elements, median will be average
    of elements at index n-1 and n in the array obtained after
    merging ar1 and ar2 */
    for (count = 0; count <= n; count++)
    {
        /*Below is to handle case where all elements of ar1[] are
        smaller than smallest(or first) element of ar2[]*/
        if (i == n)
        {
            m1 = m2;
            m2 = ar2[0];
            break;
        }

        /*Below is to handle case where all elements of ar2[] are
        smaller than smallest(or first) element of ar1[]*/
        else if (j == n)
        {
            m1 = m2;
            m2 = ar1[0];
            break;
        }

        if (ar1[i] < ar2[j])
        {
            m1 = m2; /* Store the prev median */
            m2 = ar1[i];
            i++;
        }
        else
        {
            m1 = m2; /* Store the prev median */
            m2 = ar2[j];
            j++;
        }
    }

    return (m1 + m2)/2;
}

/* Driver program to test above function */
int main()
{
    int ar1[] = { 1, 12, 15, 26, 38};
    int ar2[] = { 2, 13, 17, 30, 45};

    int n1 = sizeof(ar1)/sizeof(ar1[0]);
    int n2 = sizeof(ar2)/sizeof(ar2[0]);
    if (n1 == n2)
        printf("Median is %d", getMedian(ar1, ar2, n1));
    else
        printf("Doesn't work for arrays of unequal size");
    getchar();
    return 0;
}
```

Output

Median is 16

Time Complexity: O(n)

Method 2 (By comparing the medians of two arrays)

This method works by first getting medians of the two sorted arrays and then comparing them.

Let ar1 and ar2 be the input arrays.

Algorithm:

- 1) Calculate the medians m1 and m2 of the input arrays ar1[] and ar2[] respectively.
- 2) If m1 and m2 both are equal then we are done.
return m1 (or m2)
- 3) If m1 is greater than m2, then median is present in one of the below two subarrays.
 - a) From first element of ar1 to m1 (ar1[0..._{n/2}])
 - b) From m2 to last element of ar2 (ar2[_{n/2}...n-1])
- 4) If m2 is greater than m1, then median is present in one of the below two subarrays.
 - a) From m1 to last element of ar1 (ar1[_{n/2}...n-1])
 - b) From first element of ar2 to m2 (ar2[0..._{n/2}])
- 5) Repeat the above process until size of both the subarrays becomes 2.
- 6) If size of the two arrays is 2 then use below formula to get the median.
Median = (max(ar1[0], ar2[0]) + min(ar1[1], ar2[1]))/2

Example:

```
ar1[] = {1, 12, 15, 26, 38}
ar2[] = {2, 13, 17, 30, 45}
```

For above two arrays $m1 = 15$ and $m2 = 17$

For the above $ar1[]$ and $ar2[]$, $m1$ is smaller than $m2$. So median is present in one of the following two subarrays.

[15, 26, 38] and [2, 13, 17]

Let us repeat the process for above two subarrays:

$m1 = 26$ $m2 = 13$.

$m1$ is greater than $m2$. So the subarrays become

```
[15, 26] and [13, 17]
Now size is 2, so median = (max(ar1[0], ar2[0]) + min(ar1[1], ar2[1]))/2
                        = (max(15, 13) + min(26, 17))/2
                        = (15 + 17)/2
                        = 16
```

Implementation:

```
// A divide and conquer based efficient solution to find median
// of two sorted arrays of same size.
#include<bits/stdc++.h>
using namespace std;

int median(int l, int); /* to get median of a sorted array */

/* This function returns median of ar1[] and ar2[].
Assumptions in this function:
Both ar1[] and ar2[] are sorted arrays
Both have n elements */
int getMedian(int ar1[], int ar2[], int n)
{
    /* return -1 for invalid input */
    if (n <= 0)
        return -1;
    if (n == 1)
        return (ar1[0] + ar2[0])/2;
    if (n == 2)
        return (max(ar1[0], ar2[0]) + min(ar1[1], ar2[1])) / 2;

    int m1 = median(ar1, n); /* get the median of the first array */
    int m2 = median(ar2, n); /* get the median of the second array */

    /* If medians are equal then return either m1 or m2 */
    if (m1 == m2)
        return m1;

    /* if m1 < m2 then median must exist in ar1[m1....] and
    ar2[...m2] */
    if (m1 < m2)
    {
        if (n % 2 == 0)
            return getMedian(ar1 + n/2 - 1, ar2, n - n/2 + 1);
        return getMedian(ar1 + n/2, ar2, n - n/2);
    }

    /* if m1 > m2 then median must exist in ar1[...m1] and
    ar2[m2...] */
    if (n % 2 == 0)
        return getMedian(ar2 + n/2 - 1, ar1, n - n/2 + 1);
    return getMedian(ar2 + n/2, ar1, n - n/2);
}

/* Function to get median of a sorted array */
int median(int arr[], int n)
{
    if (n%2 == 0)
        return (arr[n/2] + arr[n/2-1])/2;
    else
        return arr[n/2];
}

/* Driver program to test above function */
int main()
{
    int ar1[] = {1, 2, 3, 6};
    int ar2[] = {4, 6, 8, 10};
    int n1 = sizeof(ar1)/sizeof(ar1[0]);
    int n2 = sizeof(ar2)/sizeof(ar2[0]);
    if (n1 == n2)
        printf("Median is %d", getMedian(ar1, ar2, n1));
    else
        printf("Doesn't work for arrays of unequal size");
    return 0;
}
```

Output :

Median is 5

Time Complexity: $O(\log n)$

Algorithmic Paradigm: Divide and Conquer

Median of two sorted arrays of different sizes

References:

<http://en.wikipedia.org/wiki/Median>

<http://ocw.alfaisal.edu/NR/rdonlyres/Electrical-Engineering-and-Computer-Science/6-046JFall-2005/30C68118-E436-4FE3-8C79-6BAFBB07D935/0/ps9sol.pdf> ds3etph5wn

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Divide and Conquer

Write a program to reverse an array or string

We strongly recommend that you click here and practice it, before moving on to the solution.

Iterative way:

1) Initialize start and end indexes.

start = 0, end = n-1

2) In a loop, swap arr[start] with arr[end] and change start and end as follows.

start = start + 1; end = end - 1

reverse-a-number



Another example to reverse a string:

reverse-a-string



C

```
// Iterative C program to reverse an array
#include<stdio.h>

/* Function to reverse arr[] from start to end */
void rreverseArray(int arr[], int start, int end)
{
    int temp;
    while (start < end)
    {
        temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
        start++;
        end--;
    }
}

/* Utility that prints out an array on a line */
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", arr[i]);

    printf("\n");
}

/* Driver function to test above functions */
int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6};
    printArray(arr, 6);
    rreverseArray(arr, 0, 5);
    printf("Reversed array is \n");
    printArray(arr, 6);
    return 0;
}
```

Java

```
// Java program to reverse an array
```

```
import java.io.*;

class ReverseArray {

    /* Function to reverse arr[] from start to end*/
    static void rverseArray(int arr[], int start, int end)
    {
        int temp;
        if (start >= end)
            return;
        temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
        rverseArray(arr, start+1, end-1);
    }

    /* Utility that prints out an array on a line */
    static void printArray(int arr[], int size)
    {
        int i;
        for (i=0; i < size; i++)
            System.out.print(arr[i] + " ");
        System.out.println("");
    }

    /*Driver function to check for above functions*/
    public static void main (String[] args) {
        int arr[] = { 1, 2, 3, 4, 5, 6};
        printArray(arr, 6);
        rverseArray(arr, 0, 5);
        System.out.println("Reversed array is ");
        printArray(arr, 6);
    }
}
/*This code is contributed by Devesh Agrawal*/
```

Output:

```
1 2 3 4 5 6
Reversed array is
6 5 4 3 2 1
```

Time Complexity: $O(n)$

Recursive Way:

- 1) Initialize start and end indexes
start = 0, end = n-1
- 2) Swap arr[start] with arr[end]
- 3) Recursively call reverse for rest of the array.

C

```
// Recursive C program to reverse an array
#include <stdio.h>

/* Function to reverse arr[] from start to end*/
void rverseArray(int arr[], int start, int end)
{
    int temp;
    if (start >= end)
        return;
    temp = arr[start];
    arr[start] = arr[end];
    arr[end] = temp;
    rverseArray(arr, start+1, end-1);
}

/* Utility that prints out an array on a line */
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", arr[i]);

    printf("\n");
}

/* Driver function to test above functions */
int main()
{
    int arr[] = { 1, 2, 3, 4, 5};
    printArray(arr, 5);
    rverseArray(arr, 0, 4);
    printf("Reversed array is \n");
    printArray(arr, 5);
    return 0;
}
```

Java

```
// Recursive Java Program to reverse an array
import java.io.*;

class ReverseArray {

    /* Function to reverse arr[] from start to end*/
    static void rverseArray(int arr[], int start, int end)
    {
        int temp;
        if (start >= end)
            return;
        temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
        rverseArray(arr, start+1, end-1);
    }

    /* Utility that prints out an array on a line */
```

```
static void printArray(int arr[], int size)
{
    for (int i=0; i < size; i++)
        System.out.print(arr[i] + " ");
    System.out.println("");
}

/*Driver function to check for above functions*/
public static void main (String[] args) {
    int arr[] = {1, 2, 3, 4, 5, 6};
    printArray(arr, 6);
    rverseArray(arr, 0, 5);
    System.out.println("Reversed array is ");
    printArray(arr, 6);
}
}
/*This article is contributed by Devesh Agrawal*/
```

Output:

```
1 2 3 4 5 6
Reversed array is
6 5 4 3 2 1
```

Time Complexity: O(n)

Please write comments if you find any bug in the above programs or other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Arrays
array

Program for array rotation

Write a function rotate(arr[], d, n) that rotates arr[] of size n by d elements.

Array

Rotation of the above array by 2 will make array

ArrayRotation1

We strongly recommend that you click here and practice it, before moving on to the solution.

METHOD 1 (Use temp array)

```
Input arr[] = [1, 2, 3, 4, 5, 6, 7], d = 2, n = 7
1) Store d elements in a temp array
   temp[] = [1, 2]
2) Shift rest of the arr[]
   arr[] = [3, 4, 5, 6, 7, 6, 7]
3) Store back the d elements
   arr[] = [3, 4, 5, 6, 7, 1, 2]
```

Time complexity O(n)

Auxiliary Space: O(d)

METHOD 2 (Rotate one by one)

```
leftRotate(arr[], d, n)
start
For i = 0 to i
To rotate by one, store arr[0] in a temporary variable temp, move arr[1] to arr[0], arr[2] to arr[1] ...and finally temp to arr[n-1]
```

Let us take the same example arr[] = [1, 2, 3, 4, 5, 6, 7], d = 2

Rotate arr[] by one 2 times

We get [2, 3, 4, 5, 6, 7, 1] after first rotation and [3, 4, 5, 6, 7, 1, 2] after second rotation.

Implementation:

C/C++

```
/*Function to left Rotate arr[] of size n by 1*/
void leftRotatebyOne(int arr[], int n);

/*Function to left rotate arr[] of size n by d*/
void leftRotate(int arr[], int d, int n)
{
    int i;
    for (i = 0; i < d; i++)
        leftRotatebyOne(arr, n);
}

void leftRotatebyOne(int arr[], int n)
{
    int i, temp;
    temp = arr[0];
    for (i = 0; i < n-1; i++)
        arr[i] = arr[i+1];
    arr[i] = temp;
}
```

```

}

/* utility function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for(i = 0; i < size; i++)
        printf("%d ", arr[i]);
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6, 7};
    leftRotate(arr, 2, 7);
    printArray(arr, 7);
    getchar();
    return 0;
}

```

Java

```

class RotateArray
{
    /*Function to left rotate arr[] of size n by d*/
    void leftRotate(int arr[], int d, int n)
    {
        int i;
        for (i = 0; i < d; i++)
            leftRotatebyOne(arr, n);
    }

    void leftRotatebyOne(int arr[], int n)
    {
        int i, temp;
        temp = arr[0];
        for (i = 0; i < n - 1; i++)
            arr[i] = arr[i + 1];
        arr[i] = temp;
    }

    /* utility function to print an array */
    void printArray(int arr[], int size)
    {
        int i;
        for (i = 0; i < size; i++)
            System.out.print(arr[i] + " ");
    }

    // Driver program to test above functions
    public static void main(String[] args)
    {
        RotateArray rotate = new RotateArray();
        int arr[] = {1, 2, 3, 4, 5, 6, 7};
        rotate.leftRotate(arr, 2, 7);
        rotate.printArray(arr, 7);
    }
}

// This code has been contributed by Mayank Jaiswal

```

Output:

```
3 4 5 6 7 1 2
```

Time complexity: $O(n \cdot d)$

Auxiliary Space: $O(1)$

METHOD 3 (A Juggling Algorithm)

This is an extension of method 2. Instead of moving one by one, divide the array in different sets

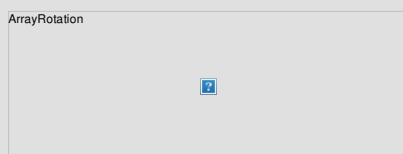
where number of sets is equal to GCD of n and d and move the elements within sets.

If GCD is 1 as is for the above example array ($n = 7$ and $d = 2$), then elements will be moved within one set only, we just start with $temp = arr[0]$ and keep moving $arr[i+d]$ to $arr[i]$ and finally store temp at the right place.

Here is an example for $n = 12$ and $d = 3$. GCD is 3 and

Let $arr[]$ be {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}

a) Elements are first moved in first set – (See below diagram for this movement)



$arr[]$ after this step \rightarrow {4 2 3 7 5 6 10 8 9 1 11 12}

b) Then in second set.

$arr[]$ after this step \rightarrow {4 5 3 7 8 6 10 11 9 1 2 12}

c) Finally in third set.

arr[] after this step --> {4 5 6 7 8 9 10 11 12 1 2 3}

Implementation:

C/C++

```
/* function to print an array */
void printArray(int arr[], int size);

/*Function to get gcd of a and b*/
int gcd(int a,int b);

/*Function to left rotate arr[] of siz n by d*/
void leftRotate(int arr[], int d, int n)
{
    int i, j, k, temp;
    for (i = 0; i < gcd(d, n); i++)
    {
        /* move i-th values of blocks */
        temp = arr[i];
        j = i;
        while(1)
        {
            k = j + d;
            if (k >= n)
                k = k - n;
            if (k == i)
                break;
            arr[j] = arr[k];
            j = k;
        }
        arr[j] = temp;
    }
}

/*UTILITY FUNCTIONS*/
/* function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for(i = 0; i < size; i++)
        printf("%d ", arr[i]);
}

/*Function to get gcd of a and b*/
int gcd(int a,int b)
{
    if(b==0)
        return a;
    else
        return gcd(b, a%b);
}

/* Driver program to test above functions */
int main()
{
    int arr[] = { 1, 2, 3, 4, 5, 6, 7};
    leftRotate(arr, 2, 7);
    printArray(arr, 7);
    getchar();
    return 0;
}
```

Java

```
class RotateArray
{
    /*Function to left rotate arr[] of siz n by d*/
    void leftRotate(int arr[], int d, int n)
    {
        int i, j, k, temp;
        for (i = 0; i < gcd(d, n); i++)
        {
            /* move i-th values of blocks */
            temp = arr[i];
            j = i;
            while (1 != 0)
            {
                k = j + d;
                if (k >= n)
                    k = k - n;
                if (k == i)
                    break;
                arr[j] = arr[k];
                j = k;
            }
            arr[j] = temp;
        }
    }

    /*UTILITY FUNCTIONS*/

    /* function to print an array */
    void printArray(int arr[], int size)
    {
        int i;
        for (i = 0; i < size; i++)
            System.out.print(arr[i] + " ");
    }
}
```

```

}

/*Function to get gcd of a and b*/
int gcd(int a, int b)
{
    if (b == 0)
        return a;
    else
        return gcd(b, a % b);
}

// Driver program to test above functions
public static void main(String[] args) {
    RotateArray rotate = new RotateArray();
    int arr[] = {1, 2, 3, 4, 5, 6, 7};
    rotate.LeftRotate(arr, 2, 7);
    rotate.printArray(arr, 7);
}
}

// This code has been contributed by Mayank Jaiswal

```

Output:

```
3 4 5 6 7 1 2
```

Time complexity: $O(n)$

Auxiliary Space: $O(1)$

Please see following posts for other methods of array rotation:

[Block swap algorithm for array rotation](#)

[Reversal algorithm for array rotation](#)

Please write comments if you find any bug in above programs/algorithms.

GATE CS Corner

Company Wise Coding Practice

Arrays
rotation

Reversal algorithm for array rotation

Write a function rotate(arr[], d, n) that rotates arr[] of size n by d elements.

Example:

```

Input: arr[] = [1, 2, 3, 4, 5, 6, 7]
d = 2
Output: arr[] = [3, 4, 5, 6, 7, 1, 2]

```

Array



Rotation of the above array by 2 will make array

ArrayRotation1



Method 4(The Reversal Algorithm)

Please read [this](#) for first three methods of array rotation.

Algorithm:

```

rotate(arr[], d, n)
reverse(arr[], 1, d);
reverse(arr[], d + 1, n);
reverse(arr[], 1, n);

```

Let AB are the two parts of the input array where A = arr[0..d-1] and B = arr[d..n-1]. The idea of the algorithm is:

Reverse A to get ArB. /* Ar is reverse of A */

Reverse B to get ArBr. /* Br is reverse of B */

Reverse all to get (ArBr) r = BA.

For arr[] = [1, 2, 3, 4, 5, 6, 7], d = 2 and n = 7

A = [1, 2] and B = [3, 4, 5, 6, 7]

Reverse A, we get ArB = [2, 1, 3, 4, 5, 6, 7]

Reverse B, we get ArBr = [2, 1, 7, 6, 5, 4, 3]
Reverse all, we get (ArBr)r = [3, 4, 5, 6, 7, 1, 2]

Implementation:

C/C++

```
// C/C++ program for reversal algorithm of array rotation
#include<stdio.h>

/*Utility function to print an array */
void printArray(int arr[], int size);

/* Utility function to reverse arr[] from start to end */
void rverseArray(int arr[], int start, int end);

/* Function to left rotate arr[] of size n by d */
void leftRotate(int arr[], int d, int n)
{
    rverseArray(arr, 0, d-1);
    rverseArray(arr, d, n-1);
    rverseArray(arr, 0, n-1);
}

/*UTILITY FUNCTIONS*/
/* function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", arr[i]);
}

/*Function to reverse arr[] from index start to end*/
void rverseArray(int arr[], int start, int end)
{
    int temp;
    while (start < end)
    {
        temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
        start++;
        end--;
    }
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6, 7};
    int n = sizeof(arr)/sizeof(arr[0]);
    int d = 2;
    leftRotate(arr, d, n);
    printArray(arr, n);
    return 0;
}
```

Java

```
// Java program for reversal algorithm of array rotation
import java.io.*;

class LeftRotate
{
    /* Function to left rotate arr[] of size n by d */
    static void leftRotate(int arr[], int d)
    {
        int n = arr.length;
        rverseArray(arr, 0, d-1);
        rverseArray(arr, d, n-1);
        rverseArray(arr, 0, n-1);
    }

    /*Function to reverse arr[] from index start to end*/
    static void rverseArray(int arr[], int start, int end)
    {
        int temp;
        while (start < end)
        {
            temp = arr[start];
            arr[start] = arr[end];
            arr[end] = temp;
            start++;
            end--;
        }
    }

    /*UTILITY FUNCTIONS*/
    /* function to print an array */
    static void printArray(int arr[])
    {
        for (int i = 0; i < arr.length; i++)
            System.out.print(arr[i] + " ");
    }

    /* Driver program to test above functions */
    public static void main (String[] args)
    {
        int arr[] = {1, 2, 3, 4, 5, 6, 7};
        leftRotate(arr, 2); // Rotate array by 2
        printArray(arr);
    }
}

/*This code is contributed by Devesh Agrawal*/
```

Python

```
# Python program for reversal algorithm of array rotation
```



```
# Function to reverse arr[] from index start to end
def rverseArray(arr, start, end):
    while (start < end):
        temp = arr[start]
        arr[start] = arr[end]
        arr[end] = temp
        start += 1
        end = end-1

# Function to left rotate arr[] of size n by d
def leftRotate(arr, d):
    n = len(arr)
    rverseArray(arr, 0, d-1)
    rverseArray(arr, d, n-1)
    rverseArray(arr, 0, n-1)

# Function to print an array
def printArray(arr):
    for i in range(0, len(arr)):
        print arr[i],

# Driver function to test above functions
arr = [1, 2, 3, 4, 5, 6, 7]
leftRotate(arr, 2) # Rotate array by 2
printArray(arr)

# This code is contributed by Devesh Agrawal
```

Output:

```
3 4 5 6 7 1 2
```

Time Complexity: O(n)

References:

<http://www.cs.bell-labs.com/cm/cs/pearls/s02b.pdf>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Arrays
rotation

Block swap algorithm for array rotation

Write a function rotate(arr[], d, n) that rotates arr[] of size n by d elements.

Array

Rotation of the above array by 2 will make array

ArrayRotation1

Algorithm:

```
Initialize A = arr[0..d-1] and B = arr[d..n-1]
1) Do following until size of A is equal to size of B

a) If A is shorter, divide B into B1 and B2 such that B2 is of same
length as A. Swap A and B2 to change AB1B2 into B2BA1. Now A
is at its final place, so recur on pieces of B.

b) If A is longer, divide A into A1 and A2 such that A1 is of same
length as B. Swap A1 and B to change A1A2B into BA1A2. Now B
is at its final place, so recur on pieces of A.

2) Finally when A and B are of equal size, block swap them.
```

Recursive Implementation:

```
#include<stdio.h>

/*Prototype for utility functions */
void printArray(int arr[], int size);
void swap(int arr[], int fi, int si, int d);

void leftRotate(int arr[], int d, int n)
{
    /* Return If number of elements to be rotated is
    zero or equal to array size */
    if(d == 0 || d == n)
        return;

    /* If number of elements to be rotated is exactly
    half of array size */
    if(n-d == d)
    {
        swap(arr, 0, n-d, d);
        return;
    }

    /* If A is shorter*/
    if(d < n-d)
    {
        swap(arr, 0, n-d, d);
        leftRotate(arr, d, n-d);
    }
    else /* If B is shorter*/
    {
        swap(arr, 0, d, n-d);
        leftRotate(arr+n-d, 2*d-n, d); /*This is tricky*/
    }
}

/*UTILITY FUNCTIONS*/
```

```

/* function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for(i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

/* This function swaps d elements starting at index fi
with d elements starting at index si */
void swap(int arr[], int fi, int si, int d)
{
    int i, temp;
    for(i = 0; i < d; i++)
    {
        temp = arr[fi + i];
        arr[fi + i] = arr[si + i];
        arr[si + i] = temp;
    }
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6, 7};
    leftRotate(arr, 2, 7);
    printArray(arr, 7);
    getchar();
    return 0;
}

```

Iterative Implementation:

Here is iterative implementation of the same algorithm. Same utility function swap() is used here.

```

void leftRotate(int arr[], int d, int n)
{
    int i, j;
    if(d == 0 || d == n)
        return;
    i = d;
    j = n - d;
    while (i != j)
    {
        if(i < j) /* A is shorter */
        {
            swap(arr, d-i, d+j-i, i);
            j -= i;
        }
        else /* B is shorter */
        {
            swap(arr, d-i, d, j);
            i -= j;
        }
        // printArray(arr, 7);
    }
    /* Finally, block swap A and B */
    swap(arr, d-i, d, i);
}

```

Time Complexity: O(n)

Please see following posts for other methods of array rotation:

<http://geeksforgeeks.org/?p=2398>

<http://geeksforgeeks.org/?p=2838>

References:

<http://www.cs.bell-labs.com/cm/cs/pearls/s02b.pdf>

Please write comments if you find any bug in the above programs/algorithms or want to share any additional information about the block swap algorithm.

GATE CS Corner Company Wise Coding Practice

Arrays
rotation

Maximum sum such that no two elements are adjacent

Given an array of positive numbers, find the maximum sum of a subsequence with the constraint that no 2 numbers in the sequence should be adjacent in the array. So 3 2 7 10 should return 13 (sum of 3 and 10) or 3 2 5 10 7 should return 15 (sum of 3, 5 and 7). Answer the question in most efficient way.

Examples :

Input : arr[] = {5, 5, 10, 100, 10, 5}
Output : 110

Input : arr[] = {1, 2, 3}
Output : 4

Input : arr[] = {1, 20, 3}
Output : 20

We strongly recommend that you click here and practice it, before moving on to the solution.

Algorithm:

Loop for all elements in arr[] and maintain two sums incl and excl where incl = Max sum including the previous element and excl = Max sum excluding the previous element.

Max sum excluding the current element will be max(incl, excl) and max sum including the current element will be excl + current element (Note that only excl is considered because elements cannot be adjacent).

At the end of the loop return max of incl and excl.

Example:

arr[] = {5, 5, 10, 40, 50, 35}

inc = 5
exc = 0

```

For i = 1 (current element is 5)
incl = (excl + arr[i]) = 5
excl = max(5, 0) = 5

For i = 2 (current element is 10)
incl = (excl + arr[i]) = 15
excl = max(5, 5) = 5

For i = 3 (current element is 40)
incl = (excl + arr[i]) = 45
excl = max(5, 15) = 15

For i = 4 (current element is 50)
incl = (excl + arr[i]) = 65
excl = max(45, 15) = 45

For i = 5 (current element is 35)
incl = (excl + arr[i]) = 80
excl = max(5, 15) = 65

```

And 35 is the last element. So, answer is $\max(\text{incl}, \text{excl}) = 80$

Thanks to [Debanjan](#) for providing code.

Implementation:

C/C++

```

#include<stdio.h>

/*Function to return max sum such that no two elements
are adjacent */
int FindMaxSum(int arr[], int n)
{
    int incl = arr[0];
    int excl = 0;
    int excl_new;
    int i;

    for (i = 1; i < n; i++)
    {
        /* current max excluding i */
        excl_new = (incl > excl) ? incl : excl;

        /* current max including i */
        incl = excl + arr[i];
        excl = excl_new;
    }

    /* return max of incl and excl */
    return ((incl > excl) ? incl : excl);
}

/* Driver program to test above function */
int main()
{
    int arr[] = {5, 5, 10, 100, 10, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("%d\n", FindMaxSum(arr, 6));
    return 0;
}

```

Java

```

class MaximumSum
{
    /*Function to return max sum such that no two elements
    are adjacent */
    int FindMaxSum(int arr[], int n)
    {
        int incl = arr[0];
        int excl = 0;
        int excl_new;
        int i;

        for (i = 1; i < n; i++)
        {
            /* current max excluding i */
            excl_new = (incl > excl) ? incl : excl;

            /* current max including i */
            incl = excl + arr[i];
            excl = excl_new;
        }

        /* return max of incl and excl */
        return ((incl > excl) ? incl : excl);
    }

    // Driver program to test above functions
    public static void main(String[] args)
    {
        MaximumSum sum = new MaximumSum();
        int arr[] = new int[] {5, 5, 10, 100, 10, 5};
        System.out.println(sum.FindMaxSum(arr, arr.length));
    }
}

// This code has been contributed by Mayank Jaiswal

```

Python

```

# Function to return max sum such that
# no two elements are adjacent
def find_max_sum(arr):
    incl = 0
    excl = 0

```

for i in arr:

```
# Current max excluding i (No ternary in
# Python)
new_excl = excl if excl>incl else incl

# Current max including i
incl = excl + i
excl = new_excl
```

```
# return max of incl and excl
return (excl if excl>incl else incl)
```

```
# Driver program to test above function
arr = [5, 5, 10, 100, 10, 5]
print find_max_sum(arr)
```

This code is contributed by Kalai Selvan

Output:

110

Time Complexity: O(n)

Now try the same problem for array with negative numbers also.

Please write comments if you find any bug in the above program/algorithm or other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Arrays
array

Leaders in an array

Write a program to print all the LEADERS in the array. An element is leader if it is greater than all the elements to its right side. And the rightmost element is always a leader. For example int the array {16, 17, 4, 3, 5, 2}, leaders are 17, 5 and 2.

Let the input array be arr[] and size of the array be size.

Method 1 (Simple)

Use two loops. The outer loop runs from 0 to size – 1 and one by one picks all elements from left to right. The inner loop compares the picked element to all the elements to its right side. If the picked element is greater than all the elements to its right side, then the picked element is the leader.

C++

```
#include<iostream>
using namespace std;

/*C++ Function to print leaders in an array */
void printLeaders(int arr[], int size)
{
    for (int i = 0; i < size; i++)
    {
        int j;
        for (j = i+1; j < size; j++)
        {
            if (arr[i] <= arr[j])
                break;
        }
        if (j == size) // the loop didn't break
            cout << arr[i] << " ";
    }
}

/* Driver program to test above function */
int main()
{
    int arr[] = {16, 17, 4, 3, 5, 2};
    int n = sizeof(arr)/sizeof(arr[0]);
    printLeaders(arr, n);
    return 0;
}
```

Java

```
class LeadersInArray
{
    /*Java Function to print leaders in an array */
    void printLeaders(int arr[], int size)
    {
        for (int i = 0; i < size; i++)
        {
            int j;
            for (j = i + 1; j < size; j++)
            {
                if (arr[i] <= arr[j])
                    break;
            }
            if (j == size) // the loop didn't break
                System.out.print(arr[i] + " ");
        }
    }

    /* Driver program to test above functions */
    public static void main(String[] args)
    {
        LeadersInArray lead = new LeadersInArray();
        int arr[] = new int[] {16, 17, 4, 3, 5, 2};
        int n = arr.length;
        lead.printLeaders(arr, n);
    }
}
```

Python

```
# Python Function to print leaders in array
```

```
def printLeaders(arr,size):
```

```
    for i in range(0, size):
        for j in range(i+1, size):
            if arr[i]<=arr[j]:
                break
        if j == size-1: # If loop didn't break
            print arr[i].
```

```
# Driver function
```

```
arr=[16, 17, 4, 3, 5, 2]
```

```
printLeaders(arr, len(arr))
```

```
# This code is contributed by __Devesh Agrawal__
```

Output:

```
17 5 2
```

Time Complexity: $O(n^2)$

Method 2 (Scan from right)

Scan all the elements from right to left in array and keep track of maximum till now. When maximum changes it's value, print it.

C++

```
#include <iostream>
using namespace std;
```

```
/* C++ Function to print leaders in an array */
```

```
void printLeaders(int arr[], int size)
```

```
{
    int max_from_right = arr[size-1];
```

```
    /* Rightmost element is always leader */
```

```
    cout << max_from_right << " ";
```

```
    for (int i = size-2; i >= 0; i--)
```

```
    {
```

```
        if (max_from_right < arr[i])
```

```
        {
```

```
            max_from_right = arr[i];
```

```
            cout << max_from_right << " ";
```

```
        }
```

```
    }
```

```
}
```

```
/* Driver program to test above function */
```

```
int main()
```

```
{
```

```
    int arr[] = {16, 17, 4, 3, 5, 2};
```

```
    int n = sizeof(arr)/sizeof(arr[0]);
```

```
    printLeaders(arr, n);
```

```
    return 0;
```

```
}
```

Java

```
class LeadersInArray
```

```
{
```

```
    /* Java Function to print leaders in an array */
```

```
    void printLeaders(int arr[], int size)
```

```
    {
```

```
        int max_from_right = arr[size-1];
```

```
        /* Rightmost element is always leader */
```

```
        System.out.print(max_from_right + " ");
```

```
        for (int i = size-2; i >= 0; i--)
```

```
        {
```

```
            if (max_from_right < arr[i])
```

```
            {
```

```
                max_from_right = arr[i];
```

```
                System.out.print(max_from_right + " ");
```

```
            }
```

```
        }
```

```
}
```

```
/* Driver program to test above functions */
```

```
public static void main(String[] args)
```

```
{
```

```
    LeadersInArray lead = new LeadersInArray();
```

```
    int arr[] = new int[]{16, 17, 4, 3, 5, 2};
```

```
    int n = arr.length;
```

```
    lead.printLeaders(arr, n);
```

```
}
```

```
}
```

Python

```
# Python function to print leaders in array
```

```
def printLeaders(arr, size):
```

```
    max_from_right = arr[size-1]
```

```
    print max_from_right,
```

```
    for i in range( size-2, 0, -1):
```

```
        if max_from_right < arr[i]:
```

```
            print arr[i],
```

```
            max_from_right = arr[i]
```

```
# Driver function
```

```
arr = [16, 17, 4, 3, 5, 2]
```

```
printLeaders(arr, len(arr))
```

This code contributed by _Devesh Agrawal__

Output

2 5 17

Time Complexity: $O(n)$

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Arrays
array

Sort elements by frequency | Set 1

Print the elements of an array in the decreasing frequency if 2 numbers have same frequency then print the one which came first.

Examples:

Input: arr[] = {2, 5, 2, 8, 5, 6, 8, 8}
Output: arr[] = {8, 8, 8, 2, 2, 5, 5, 6}

Input: arr[] = {2, 5, 2, 6, -1, 99999999, 5, 8, 8, 8}
Output: arr[] = {8, 8, 8, 2, 2, 5, 5, 6, -1, 99999999}

METHOD 1 (Use Sorting)

- 1) Use a sorting algorithm to sort the elements $O(n \log n)$
- 2) Scan the sorted array and construct a 2D array of element and count $O(n)$.
- 3) Sort the 2D array according to count $O(n \log n)$.

Example:

Input 2 5 2 8 5 6 8 8

After sorting we get
2 2 5 5 6 8 8 8

Now construct the 2D array as
2, 2
5, 2
6, 1
8, 3

Sort by count
8, 3
2, 2
5, 2
6, 1

How to maintain order of elements if frequency is same?

The above approach doesn't make sure order of elements if frequency is same. To handle this, we should use indexes in step 3, if two counts are same then we should first process (or print) the element with lower index. In step 1, we should store the indexes instead of elements.

Input 5 2 2 8 5 6 8 8

After sorting we get
Element 2 2 5 5 6 8 8 8
Index 1 2 0 4 5 3 6 7

Now construct the 2D array as
Index, Count
1, 2
0, 2
5, 1
3, 3

Sort by count (consider indexes in case of tie)
3, 3
0, 2
1, 2
5, 1

Print the elements using indexes in the above 2D array.

Below is C++ implementation of above approach.

```
// Sort elements by frequency. If two elements have same
// count, then put the elements that appears first
#include<bits/stdc++.h>
using namespace std;
```

```

// Used for sorting
struct ele
{
    int count, index, val;
};

// Used for sorting by value
bool mycomp(struct ele a, struct ele b) {
    return (a.val < b.val);
}

// Used for sorting by frequency. And if frequency is same,
// then by appearance
bool mycomp2(struct ele a, struct ele b) {
    if (a.count != b.count) return (a.count < b.count);
    else return a.index > b.index;
}

void sortByFrequency(int arr[], int n)
{
    struct ele element[n];
    for (int i = 0; i < n; i++)
    {
        element[i].index = i; /* Fill Indexes */
        element[i].count = 0; /* Initialize counts as 0 */
        element[i].val = arr[i]; /* Fill values in structure
                                elements */
    }

    /* Sort the structure elements according to value,
    we used stable sort so relative order is maintained. */
    stable_sort(element, element+n, mycomp);

    /* initialize count of first element as 1 */
    element[0].count = 1;

    /* Count occurrences of remaining elements */
    for (int i = 1; i < n; i++)
    {
        if (element[i].val == element[i-1].val)
        {
            element[i].count += element[i-1].count+1;

            /* Set count of previous element as -1 , we are
            doing this because we'll again sort on the
            basis of counts (if counts are equal than on
            the basis of index)*/
            element[i-1].count = -1;

            /* Retain the first index (Remember first index
            is always present in the first duplicate we
            used stable sort. */
            element[i].index = element[i-1].index;
        }

        /* Else if previous element is not equal to current
        so set the count to 1 */
        else element[i].count = 1;
    }

    /* Now we have counts and first index for each element so now
    sort on the basis of count and in case of tie use index
    to sort */
    stable_sort(element, element+n, mycomp2);
    for (int i = n-1, index=0; i >= 0; i--)
        if (element[i].count != -1)
            for (int j=0; j<element[i].count; j++)
                arr[index++] = element[i].val;
    }

// Driver program
int main()
{
    int arr[] = {2, 5, 2, 6, -1, 9999999, 5, 8, 8, 8};
    int n = sizeof(arr)/sizeof(arr[0]);

    sortByFrequency(arr, n);

    for (int i=0; i<n; i++)
        cout << arr[i] << " ";
    return 0;
}

```

Output:

```
8 8 8 2 2 5 6 -1 9999999
```

Thanks to [Gaurav Ahirwar](#) for providing above implementation.

METHOD 2(Use BST and Sorting)

1. Insert elements in BST one by one and if an element is already present then increment the count of the node. Node of the Binary Search Tree (used in this approach) will be as follows.

```

struct tree
{
    int element;
    int first_index /*To handle ties in counts*/
    int count;
}BST;

```

2.Store the first indexes and corresponding counts of BST in a 2D array.

3 Sort the 2D array according to counts (and use indexes in case of tie).

Time Complexity: O(nlogn) if a [Self Balancing Binary Search Tree](#) is used. This is implemented in [Set 2](#).

METHOD 3(Use Hashing and Sorting)

Using a hashing mechanism, we can store the elements (also first index) and their counts in a hash. Finally, sort the hash elements according to their counts.

Set 2:

[Sort elements by frequency | Set 2](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Count Inversions in an array | Set 1 (Using Merge Sort)

Inversion Count for an array indicates – how far (or close) the array is from being sorted. If array is already sorted then inversion count is 0. If array is sorted in reverse order that inversion count is the maximum.

Formally speaking, two elements $a[i]$ and $a[j]$ form an inversion if $a[i] > a[j]$ and $i < j$

Example:

The sequence 2, 4, 1, 3, 5 has three inversions (2, 1), (4, 1), (4, 3).

METHOD 1 (Simple)

For each element, count number of elements which are on right side of it and are smaller than it.

```
#include <bits/stdc++.h>
int getInvCount(int arr[], int n)
{
    int inv_count = 0;
    for (int i = 0; i < n - 1; i++)
        for (int j = i + 1; j < n; j++)
            if (arr[i] > arr[j])
                inv_count++;

    return inv_count;
}

/* Driver program to test above functions */
int main(int argc, char** args)
{
    int arr[] = {1, 20, 6, 4, 5};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Number of inversions are %d\n", getInvCount(arr, n));
    return 0;
}
```

Output:

Number of inversions are 5

Time Complexity: $O(n^2)$

METHOD 2(Enhance Merge Sort)

Suppose we know the number of inversions in the left half and right half of the array (let be $inv1$ and $inv2$), what kinds of inversions are not accounted for in $Inv1 + Inv2$? The answer is – the inversions we have to count during the merge step. Therefore, to get number of inversions, we need to add number of inversions in left subarray, right subarray and merge().

inv_count1



How to get number of inversions in merge()?

In merge process, let i is used for indexing left sub-array and j for right sub-array. At any step in merge(), if $a[i]$ is greater than $a[j]$, then there are $(mid - i)$ inversions. because left and right subarrays are sorted, so all the remaining elements in left-subarray ($a[i+1]$, $a[i+2]$... $a[mid]$) will be greater than $a[j]$

inv_count2



The complete picture:

inv_count3



Implementation:

```
#include <bits/stdc++.h>

int _mergeSort(int arr[], int temp[], int left, int right);
int merge(int arr[], int temp[], int left, int mid, int right);
```



```

/* This function sorts the input array and returns the
number of inversions in the array */
int mergeSort(int arr[], int array_size)
{
    int *temp = (int *)malloc(sizeof(int)*array_size);
    return _mergeSort(arr, temp, 0, array_size - 1);
}

/* An auxiliary recursive function that sorts the input array and
returns the number of inversions in the array. */
int _mergeSort(int arr[], int temp[], int left, int right)
{
    int mid, inv_count = 0;
    if (right > left)
    {
        /* Divide the array into two parts and call _mergeSortAndCountInv()
        for each of the parts */
        mid = (right + left)/2;

        /* Inversion count will be sum of inversions in left-part, right-part
        and number of inversions in merging */
        inv_count = _mergeSort(arr, temp, left, mid);
        inv_count += _mergeSort(arr, temp, mid+1, right);

        /*Merge the two parts*/
        inv_count += merge(arr, temp, left, mid+1, right);
    }
    return inv_count;
}

/* This funt merges two sorted arrays and returns inversion count in
the arrays.*/
int merge(int arr[], int temp[], int left, int mid, int right)
{
    int i, j, k;
    int inv_count = 0;

    i = left; /* i is index for left subarray*/
    j = mid; /* j is index for right subarray*/
    k = left; /* k is index for resultant merged subarray*/
    while ((i <= mid - 1) && (j <= right))
    {
        if (arr[i] <= arr[j])
        {
            temp[k++] = arr[i++];
        }
        else
        {
            temp[k++] = arr[j++];
        }

        /*this is tricky -- see above explanation/diagram for merge()*/
        inv_count = inv_count + (mid - i);
    }

    /* Copy the remaining elements of left subarray
    (if there are any) to temp*/
    while (i <= mid - 1)
        temp[k++] = arr[i++];

    /* Copy the remaining elements of right subarray
    (if there are any) to temp*/
    while (j <= right)
        temp[k++] = arr[j++];

    /*Copy back the merged elements to original array*/
    for (i=left; i <= right; i++)
        arr[i] = temp[i];

    return inv_count;
}

/* Driver program to test above functions */
int main(int argv, char** args)
{
    int arr[] = { 1, 20, 6, 4, 5};
    printf("Number of inversions are %d\n", mergeSort(arr, 5));
    getchar();
    return 0;
}

```

Output:

```

Number of inversions are 5

```

Note that above code modifies (or sorts) the input array. If we want to count only inversions then we need to create a copy of original array and call mergeSort() on copy.

Time Complexity: $O(n \log n)$

Algorithmic Paradigm: Divide and Conquer

You may like to see.

[Count inversions in an array | Set 2 \(Using Self-Balancing BST\)](#)

[Counting Inversions using Set in C++ STL](#)

[Count inversions in an array | Set 3 \(Using BIT\)](#)

References:

<http://www.cs.umd.edu/class/fall2009/cmsc451/lectures/Lec08-inversions.pdf>

<http://www.cp.eng.chula.ac.th/~piak/teaching/algo/algo2008/count-inv.htm>

Please write comments if you find any bug in the above program/algorithm or other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Divide and Conquer
Sorting
Divide and Conquer
Inversion
Merge Sort

Search in a row wise and column wise sorted matrix

Given an $n \times n$ matrix, where every row and column is sorted in increasing order. Given a number x , how to decide whether this x is in the matrix. The designed algorithm should have linear time complexity.

Thanks to [devendraiiit](#) for suggesting below approach.

- 1) Start with top right element
- 2) Loop: compare this element e with x
 -i) if they are equal then return its position
 - ...ii) $e > x$ then move it to left (if out of bound of matrix then break return false)
- 3) repeat the i), ii) and iii) till you find element or returned false

Implementation:

```
#include<stdio.h>

/* Searches the element x in mat[i][j]. If the element is found,
   then prints its position and returns true, otherwise prints
   "not found" and returns false */
int search(int mat[4][4], int n, int x)
{
    int i = 0, j = n-1; //set indexes for top right element
    while ( i < n && j >= 0 )
    {
        if ( mat[i][j] == x )
        {
            printf("n Found at %d, %d", i, j);
            return 1;
        }
        if ( mat[i][j] > x )
            j--;
        else // if mat[i][j] < x
            i++;
    }

    printf("n Element not found");
    return 0; // if ( i==n || j== -1 )
}

// driver program to test above function
int main()
{
    int mat[4][4] = { {10, 20, 30, 40},
                      {15, 25, 35, 45},
                      {27, 29, 37, 48},
                      {32, 33, 39, 50},
                    };
    search(mat, 4, 29);
    getchar();
    return 0;
}
```

Time Complexity: $O(n)$

The above approach will also work for $m \times n$ matrix (not only for $n \times n$). Complexity would be $O(m + n)$.

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Matrix

Print a given matrix in spiral form

Given a 2D array, print it in spiral form. See the following examples.

```
Input:
1  2  3  4
5  6  7  8
9 10 11 12
13 14 15 16

Output:
1 2 3 4 8 12 16 15 14 13 9 5 6 7 11 10

Input:
1  2  3  4  5  6
7  8  9 10 11 12
13 14 15 16 17 18

Output:
1 2 3 4 5 6 12 18 17 16 15 14 13 7 8 9 10 11
```

spiral-matrix



We strongly recommend that you click [here](#) and practice it, before moving on to the solution.

Solution:

```
/* This code is adopted from the solution given
   @ http://effprog.blogspot.com/2011/01/spiral-printing-of-two-dimensional.html */

#include <stdio.h>
#define R 3
#define C 6

void spiralPrint(int m, int n, int a[R][C])
{
    int i, k = 0, l = 0;

    /* k - starting row index
       m - ending row index
       l - starting column index
       n - ending column index
       i - iterator
    */

    while (k < m && l < n)
    {
        /* Print the first row from the remaining rows */
        for (i = l; i < n; ++i)
        {
            printf("%d ", a[k][i]);
        }
        k++;

        /* Print the last column from the remaining columns */
        for (i = k; i < m; ++i)
        {
            printf("%d ", a[i][n-1]);
        }
        n--;

        /* Print the last row from the remaining rows */
        if (k < m)
        {
            for (i = n-1; i >= l; --i)
            {
                printf("%d ", a[m-1][i]);
            }
            m--;
        }

        /* Print the first column from the remaining columns */
        if (l < n)
        {
            for (i = m-1; i >= k; --i)
            {
                printf("%d ", a[i][l]);
            }
            l++;
        }
    }
}

/* Driver program to test above functions */
int main()
{
    int a[R][C] = { {1, 2, 3, 4, 5, 6},
                    {7, 8, 9, 10, 11, 12},
                    {13, 14, 15, 16, 17, 18} };

    spiralPrint(R, C, a);
    return 0;
}

/* OUTPUT:
1 2 3 4 5 6 12 18 17 16 15 14 13 7 8 9 10 11
*/
```

Time Complexity: Time complexity of the above solution is $O(mn)$.

Please write comments if you find the above code incorrect, or find other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Matrix
Amazon-Question
pattern-printing
Snapdeal-Question
spiral

About wgpshashank

Shashank is Passionate About Computer Science, Problem Solving & Technology he graduated from Birla Institute of Technology Mesra. Design and Analysis of Algorithms, Application of Data Structures are his area of Interest & he Wants to Contribute to Computer Science. You can find him more active on his personal blog "Cracking The Code" <http://shashank7s.blogspot.com> Cheers !!!
[View all posts by wgpshashank](#) --

A Boolean Matrix Question

Given a boolean matrix `mat[M][N]` of size `M X N`, modify it such that if a matrix cell `mat[i][j]` is 1 (or true) then make all the cells of `i`th row and `j`th column as 1.

Example 1
The matrix
1 0
0 0
should be changed to following
1 1
1 0

Example 2
The matrix
0 0 0
0 0 1
should be changed to following
0 0 1
1 1 1

Example 3
The matrix
1 0 0 1
0 0 1 0
0 0 0 0
should be changed to following
1 1 1 1
1 1 1 1
1 0 1 1

We strongly recommend that you click here and practice it, before moving on to the solution.

Method 1 (Use two temporary arrays)

- 1) Create two temporary arrays `row[M]` and `col[N]`. Initialize all values of `row[]` and `col[]` as 0.
- 2) Traverse the input matrix `mat[M][N]`. If you see an entry `mat[i][j]` as true, then mark `row[i]` and `col[j]` as true.
- 3) Traverse the input matrix `mat[M][N]` again. For each entry `mat[i][j]`, check the values of `row[i]` and `col[j]`. If any of the two values (`row[i]` or `col[j]`) is true, then mark `mat[i][j]` as true.

Thanks to [Dixit Sethi](#) for suggesting this method.

```
#include <stdio.h>
#define R 3
#define C 4

void modifyMatrix(bool mat[R][C])
{
    bool row[R];
    bool col[C];

    int i, j;

    /* Initialize all values of row[] as 0 */
    for (i = 0; i < R; i++)
    {
        row[i] = 0;
    }

    /* Initialize all values of col[] as 0 */
    for (i = 0; i < C; i++)
    {
        col[i] = 0;
    }

    /* Store the rows and columns to be marked as 1 in row[] and col[]
    arrays respectively */
    for (i = 0; i < R; i++)
    {
        for (j = 0; j < C; j++)
        {
            if (mat[i][j] == 1)
            {
```

```

        row[i] = 1;
        col[j] = 1;
    }
}

/* Modify the input matrix mat[] using the above constructed row[] and
col[] arrays */
for (i = 0; i < R; i++)
{
    for (j = 0; j < C; j++)
    {
        if ( row[i] == 1 || col[j] == 1 )
        {
            mat[i][j] = 1;
        }
    }
}

/* A utility function to print a 2D matrix */
void printMatrix(bool mat[R][C])
{
    int i, j;
    for (i = 0; i < R; i++)
    {
        for (j = 0; j < C; j++)
        {
            printf("%d ", mat[i][j]);
        }
        printf("\n");
    }
}

/* Driver program to test above functions */
int main()
{
    bool mat[R][C] = { { 1, 0, 0, 1},
                        { 0, 0, 1, 0},
                        { 0, 0, 0, 0},
                        };

    printf("Input Matrix \n");
    printMatrix(mat);

    modifyMatrix(mat);

    printf("Matrix after modification \n");
    printMatrix(mat);

    return 0;
}

```

Output:

```

Input Matrix
1 0 0 1
0 0 1 0
0 0 0 0
Matrix after modification
1 1 1 1
1 1 1 1
1 0 1 1

```

Time Complexity: $O(M*N)$

Auxiliary Space: $O(M + N)$

Method 2 (A Space Optimized Version of Method 1)

This method is a space optimized version of above method 1. This method uses the first row and first column of the input matrix in place of the auxiliary arrays row[] and col[] of method 1. So what we do is: first take care of first row and column and store the info about these two in two flag variables rowFlag and colFlag. Once we have this info, we can use first row and first column as auxiliary arrays and apply method 1 for submatrix (matrix excluding first row and first column) of size $(M-1)*(N-1)$.

- 1) Scan the first row and set a variable rowFlag to indicate whether we need to set all 1s in first row or not.
- 2) Scan the first column and set a variable colFlag to indicate whether we need to set all 1s in first column or not.
- 3) Use first row and first column as the auxiliary arrays row[] and col[] respectively, consider the matrix as submatrix starting from second row and second column and apply method 1.
- 4) Finally, using rowFlag and colFlag, update first row and first column if needed.

Time Complexity: $O(M*N)$

Auxiliary Space: $O(1)$

Thanks to [Siddh](#) for suggesting this method.

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Matrix

Print unique rows in a given boolean matrix

Given a binary matrix, print all unique rows of the given matrix.

```

Input:
{0, 1, 0, 0, 1}
{1, 0, 1, 1, 0}
{0, 1, 0, 0, 1}
{1, 1, 1, 0, 0}

Output:
0 1 0 0 1
1 0 1 1 0
1 1 1 0 0

```

Method 1 (Simple)

A simple approach is to check each row with all processed rows. Print the first row. Now, starting from the second row, for each row, compare the row with already processed rows. If the row matches with any of the processed rows, don't print it. If the current row doesn't match with any row, print it.

Time complexity: $O(\text{ROW}^2 \times \text{COL})$

Auxiliary Space: $O(1)$

Method 2 (Use Binary Search Tree)

Find the decimal equivalent of each row and insert it into BST. Each node of the BST will contain two fields, one field for the decimal value, other for row number. Do not insert a node if it is duplicated. Finally, traverse the BST and print the corresponding rows.

Time complexity: $O(\text{ROW} \times \text{COL} + \text{ROW} \times \log(\text{ROW}))$

Auxiliary Space: $O(\text{ROW})$

This method will lead to Integer Overflow if number of columns is large.

Method 3 (Use Trie data structure)

Since the matrix is boolean, a variant of Trie data structure can be used where each node will be having two children one for 0 and other for 1. Insert each row in the Trie. If the row is already there, don't print the row. If row is not there in Trie, insert it in Trie and print it.

Below is C implementation of method 3.

```
// Given a binary matrix of M X N of integers, you need to return only unique rows of binary array
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define ROW 4
#define COL 5

// A Trie node
typedef struct Node
{
    bool isEndOfCol;
    struct Node *child[2]; // Only two children needed for 0 and 1
} Node;

// A utility function to allocate memory for a new Trie node
Node* newNode()
{
    Node* temp = (Node*)malloc(sizeof(Node));
    temp->isEndOfCol = 0;
    temp->child[0] = temp->child[1] = NULL;
    return temp;
}

// Inserts a new matrix row to Trie. If row is already
// present, then returns 0, otherwise inserts the row and
// return 1
bool insert( Node** root, int (*M)[COL], int row, int col )
{
    // base case
    if ( *root == NULL )
        *root = newNode();

    // Recur if there are more entries in this row
    if ( col < COL )
        return insert ( &(*root->child[ M[row][col] ] ), M, row, col+1 );

    else // If all entries of this row are processed
    {
        // unique row found, return 1
        if ( !(*root->isEndOfCol) )
            return (*root->isEndOfCol = 1);

        // duplicate row found, return 0
        return 0;
    }
}

// A utility function to print a row
void printRow( int (*M)[COL], int row )
{
    int i;
    for( i = 0; i < COL; ++i )
        printf( "%d ", M[row][i] );
    printf( "\n" );
}

// The main function that prints all unique rows in a
// given matrix.
void findUniqueRows( int (*M)[COL] )
{
    Node* root = NULL; // create an empty Trie
    int i;

    // Iterate through all rows
    for ( i = 0; i < ROW; ++i )
    {
        // insert row to TRIE
        if ( insert(&root, M, i, 0) )
            // unique row found, print it
            printRow( M, i );
    }
}

// Driver program to test above functions
int main()
{
    int M[ROW][COL] = {{0, 1, 0, 0, 1},
                       {1, 0, 1, 1, 0},
                       {0, 1, 0, 0, 1},
                       {1, 0, 1, 0, 0}
    };

    findUniqueRows( M );

    return 0;
}
```

Time complexity: $O(\text{ROW} \times \text{COL})$

Auxiliary Space: $O(\text{ROW} \times \text{COL})$

This method has better time complexity. Also, relative order of rows is maintained while printing.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Matrix
Advance Data Structures
Advanced Data Structures

Maximum size square sub-matrix with all 1s

Given a binary matrix, find out the maximum size square sub-matrix with all 1s.

For example, consider the below binary matrix.

maximum-size-square-sub-matrix-with-all-1s



We strongly recommend that you click here and practice it, before moving on to the solution.

Algorithm:

Let the given binary matrix be $M[R][C]$. The idea of the algorithm is to construct an auxiliary size matrix $S[][]$ in which each entry $S[i][j]$ represents size of the square sub-matrix with all 1s including $M[i][j]$ where $M[i][j]$ is the rightmost and bottommost entry in sub-matrix.

- 1) Construct a sum matrix $S[R][C]$ for the given $M[R][C]$.
 - a) Copy first row and first columns as it is from $M[][]$ to $S[][]$
 - b) For other entries, use following expressions to construct $S[][]$
If $M[i][j]$ is 1 then
 $S[i][j] = \min(S[i][j-1], S[i-1][j], S[i-1][j-1]) + 1$
Else /* If $M[i][j]$ is 0 */
 $S[i][j] = 0$
- 2) Find the maximum entry in $S[R][C]$
- 3) Using the value and coordinates of maximum entry in $S[i][j]$, print sub-matrix of $M[][]$

For the given $M[R][C]$ in above example, constructed $S[R][C]$ would be:

```
0 1 1 0 1
1 1 0 1 0
0 1 1 1 0
1 1 2 2 0
1 2 2 3 1
0 0 0 0 0
```

The value of maximum entry in above matrix is 3 and coordinates of the entry are (4, 3). Using the maximum value and its coordinates, we can find out the required sub-matrix.

```
#include<stdio.h>
#define bool int
#define R 6
#define C 5

void printMaxSubSquare(bool M[R][C])
{
    int i,j;
    int S[R][C];
    int max_of_s, max_i, max_j;

    /* Set first column of S[][] */
    for(i = 0; i < R; i++)
        S[i][0] = M[i][0];

    /* Set first row of S[][] */
    for(j = 0; j < C; j++)
        S[0][j] = M[0][j];

    /* Construct other entries of S[][] */
    for(i = 1; i < R; i++)
    {
        for(j = 1; j < C; j++)
        {
            if(M[i][j] == 1)
                S[i][j] = min(S[i][j-1], S[i-1][j], S[i-1][j-1]) + 1;
            else
                S[i][j] = 0;
        }
    }

    /* Find the maximum entry, and indexes of maximum entry
    in S[][] */
    max_of_s = S[0][0]; max_i = 0; max_j = 0;
    for(i = 0; i < R; i++)
    {
        for(j = 0; j < C; j++)
```

```

{
    if(max_of_s < S[i][j])
    {
        max_of_s = S[i][j];
        max_i = i;
        max_j = j;
    }
}

printf("\n Maximum size sub-matrix is: \n");
for(i = max_j; i > max_i - max_of_s; i--)
{
    for(j = max_j; j > max_j - max_of_s; j--)
    {
        printf("%d ", M[i][j]);
    }
    printf("\n");
}

/* UTILITY FUNCTIONS */
/* Function to get minimum of three values */
int min(int a, int b, int c)
{
    int m = a;
    if (m > b)
        m = b;
    if (m > c)
        m = c;
    return m;
}

/* Driver function to test above functions */
int main()
{
    bool M[R][C] = {{0, 1, 1, 0, 1},
                    {1, 1, 0, 1, 0},
                    {0, 1, 1, 1, 0},
                    {1, 1, 1, 1, 0},
                    {1, 1, 1, 1, 1},
                    {0, 0, 0, 0, 0}};

    printMaxSubSquare(M);
    getchar();
}

```

Time Complexity: $O(m \times n)$ where m is number of rows and n is number of columns in the given matrix.

Auxiliary Space: $O(m \times n)$ where m is number of rows and n is number of columns in the given matrix.

Algorithmic Paradigm: Dynamic Programming

Please write comments if you find any bug in above code/algorithm, or find other ways to solve the same problem

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Matrix
Dynamic Programming
Matrix

Print unique rows in a given boolean matrix

Given a binary matrix, print all unique rows of the given matrix.

```

Input:
{0, 1, 0, 0, 1}
{1, 0, 1, 1, 0}
{0, 1, 0, 0, 1}
{1, 1, 1, 0, 0}

Output:
0 1 0 0 1
1 0 1 1 0
1 1 1 0 0

```

Method 1 (Simple)

A simple approach is to check each row with all processed rows. Print the first row. Now, starting from the second row, for each row, compare the row with already processed rows. If the row matches with any of the processed rows, don't print it. If the current row doesn't match with any row, print it.

Time complexity: $O(\text{ROW}^2 \times \text{COL})$

Auxiliary Space: $O(1)$

Method 2 (Use Binary Search Tree)

Find the decimal equivalent of each row and insert it into BST. Each node of the BST will contain two fields, one field for the decimal value, other for row number. Do not insert a node if it is duplicated. Finally, traverse the BST and print the corresponding rows.

Time complexity: $O(\text{ROW} \times \text{COL} + \text{ROW} \times \log(\text{ROW}))$

Auxiliary Space: $O(\text{ROW})$

This method will lead to Integer Overflow if number of columns is large.

Method 3 (Use Trie data structure)

Since the matrix is boolean, a variant of Trie data structure can be used where each node will be having two children one for 0 and other for 1. Insert each row in the Trie. If the row is already there, don't print the row. If row is not there in Trie, insert it in Trie and print it.

Below is C implementation of method 3.

```

//Given a binary matrix of M X N of integers, you need to return only unique rows of binary array
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define ROW 4
#define COL 5

// A Trie node
typedef struct Node
{
    bool isEndOfCol;
    struct Node *child[2]; // Only two children needed for 0 and 1
} Node;

```



```

// A utility function to allocate memory for a new Trie node
Node* newNode()
{
    Node* temp = (Node *)malloc( sizeof( Node ) );
    temp->isEndOfCol = 0;
    temp->child[0] = temp->child[1] = NULL;
    return temp;
}

// Inserts a new matrix row to Trie. If row is already
// present, then returns 0, otherwise inserts the row and
// return 1
bool insert( Node** root, int (*M)[COL], int row, int col )
{
    // base case
    if ( *root == NULL )
        *root = newNode();

    // Recur if there are more entries in this row
    if ( col < COL )
        return insert ( &( (*root)->child[ M[row][col] ] ), M, row, col+1 );

    else // If all entries of this row are processed
    {
        // unique row found, return 1
        if ( ! ( (*root)->isEndOfCol ) )
            return ( (*root)->isEndOfCol = 1;

        // duplicate row found, return 0
        return 0;
    }
}

// A utility function to print a row
void printRow( int (*M)[COL], int row )
{
    int i;
    for( i = 0; i < COL; ++i )
        printf( "%d ", M[row][i] );
    printf( "\n" );
}

// The main function that prints all unique rows in a
// given matrix.
void findUniqueRows( int (*M)[COL] )
{
    Node* root = NULL; // create an empty Trie
    int i;

    // Iterate through all rows
    for ( i = 0; i < ROW; ++i )
        // insert row to TRIE
        if ( insert(&root, M, i, 0) )
            // unique row found, print it
            printRow( M, i );
}

// Driver program to test above functions
int main()
{
    int M[ROW][COL] = {{0, 1, 0, 0, 1},
                        {1, 0, 1, 1, 0},
                        {0, 1, 0, 0, 1},
                        {1, 0, 1, 0, 0}
    };

    findUniqueRows( M );

    return 0;
}

```

Time complexity: $O(\text{ROW} \times \text{COL})$

Auxiliary Space: $O(\text{ROW} \times \text{COL})$

This method has better time complexity. Also, relative order of rows is maintained while printing.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Matrix
Advance Data Structures
Advanced Data Structures

Inplace (Fixed space) M x N size matrix transpose | Updated

About four months of gap (missing GFG), a new post. Given an $M \times N$ matrix, transpose the matrix without auxiliary memory. It is easy to transpose matrix using an auxiliary array. If the matrix is symmetric in size, we can transpose the matrix inplace by mirroring the 2D array across its diagonal (try yourself). How to transpose an arbitrary size matrix inplace? See the following matrix,

```

a b c   a d g j
d e f ==> b e h k

```

```
g h i   c f i l
j k l
```

As per 2D numbering in C/C++, corresponding location mapping looks like,

```
Org element New
0  a  0
1  b  4
2  c  8
3  d  1
4  e  5
5  f  9
6  g  2
7  h  6
8  i  10
9  j  3
10 k  7
11 l  11
```

Note that the first and last elements stay in their original location. We can easily see the transformation forms few permutation cycles.

- 1->4->5->9->3->1 – Total 5 elements form the cycle
- 2->8->10->7->6->2 – Another 5 elements form the cycle
- 0 – Self cycle
- 11 – Self cycle

From the above example, we can easily devise an algorithm to move the elements along these cycles. *How can we generate permutation cycles?* Number of elements in both the matrices are constant, given by $N = R * C$, where R is row count and C is column count. An element at location ol (old location in $R \times C$ matrix), moved to nl (new location in $C \times R$ matrix). We need to establish relation between ol , nl , R and C . Assume $ol = A[or][oc]$. In C/C++ we can calculate the element address as,

```
ol = or x C + oc (ignore base reference for simplicity)
```

It is to be moved to new location nl in the transposed matrix, say $nl = A[nr][nc]$, or in C/C++ terms

```
nl = nr x R + nc (R - column count, C is row count as the matrix is transposed)
```

Observe, $nr = oc$ and $nc = or$, so replacing these for nl ,

```
nl = oc x R + or -----> [eq 1]
```

after solving for relation between ol and nl , we get

```
ol = or x C + oc
ol x R = or x C x R + oc x R
= or x N + oc x R (from the fact R * C = N)
= or x N + (nl - or) --- from [eq 1]
= or x (N-1) + nl
```

OR,

```
nl = ol x R - or x (N-1)
```

Note that the values of nl and ol never go beyond $N-1$, so considering modulo division on both the sides by $(N-1)$, we get the following based on properties of congruence,

```
nl mod (N-1) = (ol x R - or x (N-1)) mod (N-1)
= (ol x R) mod (N-1) - or x (N-1) mod (N-1)
= ol x R mod (N-1), since second term evaluates to zero
nl = (ol x R) mod (N-1), since nl is always less than N-1
```

A curious reader might have observed the significance of above relation. Every location is scaled by a factor of R (row size). It is obvious from the matrix that every location is displaced by scaled factor of R . The actual multiplier depends on congruence class of $(N-1)$, i.e. the multiplier can be both -ve and +ve value of the congruent class. Hence every location transformation is simple modulo division. These modulo divisions form cyclic permutations. We need some book keeping information to keep track of already moved elements. Here is code for inplace matrix transformation,

```
// Program for in-place matrix transpose
#include <stdio.h>
#include <iostream>
#include <bitset>
#define HASH_SIZE 128

using namespace std;

// A utility function to print a 2D array of size nr x nc and base address A
void Print2DArray(int *A, int nr, int nc)
{
    for(int r = 0; r < nr; r++)
    {
        for(int c = 0; c < nc; c++)
            printf("%4d", *(A + r*nc + c));

        printf("\n");
    }

    printf("\n\n");
}

// Non-square matrix transpose of matrix of size r x c and base address A
void MatrixInplaceTranspose(int *A, int r, int c)
{
    int size = r*c - 1;
    int t; // holds element to be replaced, eventually becomes next element to move
    int next; // location of t to be moved
    int cycleBegin; // holds start of cycle
    int i; // iterator
    bitset<HASH_SIZE> b; // hash to mark moved elements

    b.reset();
    b[0] = b[size] = 1;
    i = 1; // Note that A[0] and A[size-1] won't move
    while (i < size)
    {
        cycleBegin = i;
        t = A[i];
        do
        {
            // Input matrix [r x c]
            // Output matrix
            // i_new = (i*r)% (N-1)
            next = (i*r)%size;
            swap(A[next], t);

            // Program for in-place matrix transpose
            #include <stdio.h>
            #include <iostream>
            #include <bitset>
            #define HASH_SIZE 128

            using namespace std;

            // A utility function to print a 2D array of size nr x nc and base address A
            void Print2DArray(int *A, int nr, int nc)
            {
                for(int r = 0; r < nr; r++)
                {
                    for(int c = 0; c < nc; c++)
                        printf("%4d", *(A + r*nc + c));

                    printf("\n");
                }

                printf("\n\n");
            }

            // Non-square matrix transpose of matrix of size r x c and base address A
            void MatrixInplaceTranspose(int *A, int r, int c)
            {
                int size = r*c - 1;
                int t; // holds element to be replaced, eventually becomes next element to move
                int next; // location of t to be moved
                int cycleBegin; // holds start of cycle
                int i; // iterator
                bitset<HASH_SIZE> b; // hash to mark moved elements

                b.reset();
                b[0] = b[size] = 1;
                i = 1; // Note that A[0] and A[size-1] won't move
                while (i < size)
                {
                    cycleBegin = i;
                    t = A[i];
                    do
                    {
                        // Input matrix [r x c]
                        // Output matrix
                        // i_new = (i*r)% (N-1)
                        next = (i*r)%size;
                        swap(A[next], t);
```

```

        b[i] = 1;
        i = next;
    }
    while (i != cycleBegin);

    // Get Next Move (what about querying random location?)
    for (i = 1; i < size && b[i]; i++)
        ;
    cout << endl;
}

// Driver program to test above function
int main(void)
{
    int r = 5, c = 6;
    int size = r*c;
    int *A = new int[size];

    for(int i = 0; i < size; i++)
        A[i] = i+1;

    Print2DArray(A, r, c);
    MatrixInplaceTranspose(A, r, c);
    Print2DArray(A, c, r);

    delete[] A;

    return 0;
}

```

Output:

```

1  2  3  4  5  6
7  8  9 10 11 12
13 14 15 16 17 18
19 20 21 22 23 24
25 26 27 28 29 30

1  7 13 19 25
2  8 14 20 26
3  9 15 21 27
4 10 16 22 28
5 11 17 23 29
6 12 18 24 30

```

Extension: 17 – March – 2013 Some [readers](#) identified similarity between the matrix transpose and [string transformation](#). Without much theory I am presenting the problem and solution. In given array of elements like [a1b2c3d4e5f6g7h8i9j1k2l3m4]. Convert it to [abcde fghijklm1234567891234]. The program should run inplace. What we need is an inplace transpose. Given below is code.

```

#include <stdio.h>
#include <iostream>
#include <bitset>
#define HASH_SIZE 128

using namespace std;

typedef char data_t;

void Print2DArray(char A[], int nr, int nc) {
    int size = nr*nc;
    for(int i = 0; i < size; i++)
        printf("%4c", *(A + i));

    printf("\n");
}

void MatrixTransposeInplaceArrangement(data_t A[], int r, int c) {
    int size = r*c - 1;
    data_t t; // holds element to be replaced, eventually becomes next element to move
    int next; // location of T to be moved
    int cycleBegin; // holds start of cycle
    int i; // iterator
    bitset<HASH_SIZE> b; // hash to mark moved elements

    b.reset();
    b[0] = b[size] = 1;
    i = 1; // Note that A[0] and A[size-1] won't move
    while (i < size) {
        cycleBegin = i;
        t = A[i];
        do {
            // Input matrix [r x c]
            // Output matrix
            // i_new = (i*r)%size
            next = (i*r)%size;
            swap(A[next], t);
            b[i] = 1;
            i = next;
        } while (i != cycleBegin);

        // Get Next Move (what about querying random location?)
        for(i = 1; i < size && b[i]; i++)
            ;
        cout << endl;
    }
}

void Fill(data_t buf[], int size) {
    // Fill abcd ...
    for(int i = 0; i < size; i++)
        buf[i] = 'a'+i;

    // Fill 0123 ...
    buf += size;
    for(int i = 0; i < size; i++)
        buf[i] = '0'+i;
}

void TestCase_01(void) {
    int r = 2, c = 10;
    int size = r*c;
    data_t *A = new data_t[size];

    Fill(A, c);
}

```

```

Print2DArray(A, r, c), cout << endl;
MatrixTransposeInplaceArrangement(A, r, c);
Print2DArray(A, c, r), cout << endl;

delete[] A;
}

int main() {
    TestCase_01();

    return 0;
}

```

+++++

Update 09-July-2016: Notes on space complexity and storage order.

After long time, it happened to review this post. Some readers pointed valid questions on how can it be in-place (?) when we are using bitset as marker (*hash* in code). Apologies for incorrect perception by looking at the article heading or content. While preparing the initial content, I was thinking of naive implementation using auxiliary space of atleast $O(MN)$ needed to transpose rectangular matrix. The program presented above is using constant space as bitset size is fixed at compile time. However, to support arbitrary size of matrices we need bitset size atleast $O(MN)$ size. One can use a *HashMap* (amortized $O(1)$ complexity) for marking finished locations, yet *HashMap*'s worst case complexity can be $O(N)$ or $O(\log N)$ based on implementation. *HashMap* space cost also increases based on items inserted. *Please note that **in-place** was used w.r.t. matrix space.*

Also, it was assumed that the matrix will be stored in row major ordering (contiguous locations in memory). The reader can derive the formulae, if the matrix is represented in column major order by the programming language (e.g. Fortran/Julia).

Thanks to the readers who pointed these two gaps.

+++++

The post is incomplete without mentioning two links.

1. Aashish covered good theory behind cycle leader algorithm. See his post on [string transformation](#).
2. As usual, [Sambasiva](#) demonstrated his exceptional skills in recursion to the [problem](#). Ensure to understand his solution.

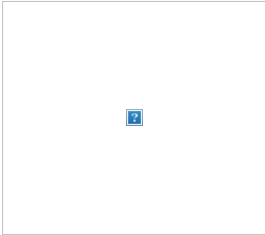
— Venki. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Matrix
About Venki
Software Engineer
[View all posts by Venki](#) →

Dynamic Programming | Set 27 (Maximum sum rectangle in a 2D matrix)

Given a 2D array, find the maximum sum subarray in it. For example, in the following 2D array, the maximum sum subarray is highlighted with blue rectangle and sum of this subarray is 29.



This problem is mainly an extension of [Largest Sum Contiguous Subarray for 1D array](#).

The **naive solution** for this problem is to check every possible rectangle in given 2D array. This solution requires 4 nested loops and time complexity of this solution would be $O(n^4)$.

Kadane's algorithm for 1D array can be used to reduce the time complexity to $O(n^3)$. The idea is to fix the left and right columns one by one and find the maximum sum contiguous rows for every left and right column pair. We basically find top and bottom row numbers (which have maximum sum) for every fixed left and right column pair. To find the top and bottom row numbers, calculate sum of elements in every row from left to right and store these sums in an array say *temp[]*. So *temp[i]* indicates sum of elements from left to right in row *i*. If we apply Kadane's 1D algorithm on *temp[]*, and get the maximum sum subarray of *temp*, this maximum sum would be the maximum possible sum with left and right as boundary columns. To get the overall maximum sum, we compare this sum with the maximum sum so far.

C

```

// Program to find maximum sum subarray in a given 2D array
#include <stdio.h>
#include <string.h>
#include <limits.h>
#define ROW 4
#define COL 5

// Implementation of Kadane's algorithm for 1D array. The function
// returns the maximum sum and stores starting and ending indexes of the
// maximum sum subarray at addresses pointed by start and finish pointers
// respectively.
int kadane(int* arr, int* start, int* finish, int n)
{
    // initialize sum, maxSum and
    int sum = 0, maxSum = INT_MIN, i;

    // Just some initial value to check for all negative values case
    *finish = -1;

    // local variable
    int local_start = 0;

    for (i = 0; i < n; ++i)
    {
        sum += arr[i];
        if (sum < 0)
        {
            sum = 0;
            local_start = i + 1;
        }
        else if (sum > maxSum)
        {
            maxSum = sum;
            *start = local_start;
            *finish = i;
        }
    }
}

// There is at-least one non-negative number

```

```

if (*finish != -1)
    return maxSum;

// Special Case: When all numbers in arr[] are negative
maxSum = arr[0];
*start = *finish = 0;

// Find the maximum element in array
for (i = 1; i < n; i++)
{
    if (arr[i] > maxSum)
    {
        maxSum = arr[i];
        *start = *finish = i;
    }
}
return maxSum;
}

// The main function that finds maximum sum rectangle in M[][]
void findMaxSum(int M[][COL])
{
    // Variables to store the final output
    int maxSum = INT_MIN, finalLeft, finalRight, finalTop, finalBottom;

    int left, right, i;
    int temp[ROW], sum, start, finish;

    // Set the left column
    for (left = 0; left < COL; ++left)
    {
        // Initialize all elements of temp as 0
        memset(temp, 0, sizeof(temp));

        // Set the right column for the left column set by outer loop
        for (right = left; right < COL; ++right)
        {
            // Calculate sum between current left and right for every row 'i'
            for (i = 0; i < ROW; ++i)
                temp[i] += M[i][right];

            // Find the maximum sum subarray in temp[]. The kadane()
            // function also sets values of start and finish. So 'sum' is
            // sum of rectangle between (start, left) and (finish, right)
            // which is the maximum sum with boundary columns strictly as
            // left and right.
            sum = kadane(temp, &start, &finish, ROW);

            // Compare sum with maximum sum so far. If sum is more, then
            // update maxSum and other output values
            if (sum > maxSum)
            {
                maxSum = sum;
                finalLeft = left;
                finalRight = right;
                finalTop = start;
                finalBottom = finish;
            }
        }
    }

    // Print final values
    printf("(Top, Left) (%d, %d)\n", finalTop, finalLeft);
    printf("(Bottom, Right) (%d, %d)\n", finalBottom, finalRight);
    printf("Max sum is: %d\n", maxSum);
}

// Driver program to test above functions
int main()
{
    int M[ROW][COL] = {{1, 2, -1, -4, -20},
                        {-8, -3, 4, 2, 1},
                        {3, 8, 10, 1, 3},
                        {-4, -1, 1, 7, -6}
                        };

    findMaxSum(M);

    return 0;
}

```

Java

```

import java.util.*;
import java.lang.*;
import java.io.*;

/**
 * Given a 2D array, find the maximum sum subarray in it
 */
class Ideone
{
    public static void main (String[] args) throws java.lang.Exception
    {
        findMaxSubMatrix(new int[][] {
            {1, 2, -1, -4, -20},
            {-8, -3, 4, 2, 1},
            {3, 8, 10, 1, 3},
            {-4, -1, 1, 7, -6}
        });
    }

    /**
     * To find maxSum in 1d array
     *
     * return {maxSum, left, right}
     */
    public static int[] kadane(int[] a) {
        //result[0] == maxSum, result[1] == start, result[2] == end;
        int[] result = new int[] {Integer.MIN_VALUE, 0, -1};
        int currentSum = 0;
        int localStart = 0;
    }
}

```

```

for (int i = 0; i < a.length; i++) {
    currentSum += a[i];
    if (currentSum < 0) {
        currentSum = 0;
        localStart = i + 1;
    } else if (currentSum > result[0]) {
        result[0] = currentSum;
        result[1] = localStart;
        result[2] = i;
    }
}

//all numbers in a are negative
if (result[2] == -1) {
    result[0] = 0;
    for (int i = 0; i < a.length; i++) {
        if (a[i] > result[0]) {
            result[0] = a[i];
            result[1] = i;
            result[2] = i;
        }
    }
}

return result;
}

/**
 * To find and print maxSum, (left, top),(right, bottom)
 */
public static void findMaxSubMatrix(int[][] a) {
    int cols = a[0].length;
    int rows = a.length;
    int[] currentResult;
    int maxSum = Integer.MIN_VALUE;
    int left = 0;
    int top = 0;
    int right = 0;
    int bottom = 0;

    for (int leftCol = 0; leftCol < cols; leftCol++) {
        int[] tmp = new int[rows];

        for (int rightCol = leftCol; rightCol < cols; rightCol++) {

            for (int i = 0; i < rows; i++) {
                tmp[i] += a[i][rightCol];
            }
            currentResult = kadane(tmp);
            if (currentResult[0] > maxSum) {
                maxSum = currentResult[0];
                left = leftCol;
                top = currentResult[1];
                right = rightCol;
                bottom = currentResult[2];
            }
        }
    }

    System.out.println("MaxSum: " + maxSum +
        ", range: [(" + left + ", " + top +
        ")(" + right + ", " + bottom + ")"]);
}
}
// Thanks to Ilia Savin for contributing this code.

```

Output:

```

(Top, Left) (1, 1)
(Bottom, Right) (3, 3)
Max sum is: 29

```

Time Complexity: $O(n^3)$

This article is compiled by [Aashish Barnwal](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Matrix
Dynamic Programming
Matrix

Divide and Conquer | Set 5 (Strassen's Matrix Multiplication)

Given two square matrices A and B of size $n \times n$ each, find their multiplication matrix.

Naive Method

Following is a simple way to multiply two matrices.

```

void multiply(int A[][N], int B[][N], int C[][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            C[i][j] = 0;
            for (int k = 0; k < N; k++)
            {
                C[i][j] += A[i][k]*B[k][j];
            }
        }
    }
}

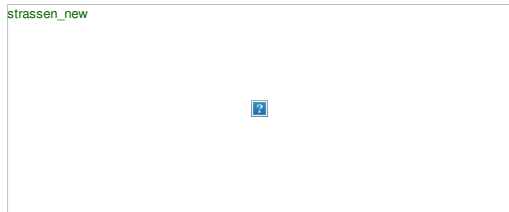
```

Time Complexity of above method is $O(N^3)$.

Divide and Conquer

Following is simple Divide and Conquer method to multiply two square matrices.

- 1) Divide matrices A and B in 4 sub-matrices of size $N/2 \times N/2$ as shown in the below diagram.
- 2) Calculate following values recursively: $ae + bg$, $af + bh$, $ce + dg$ and $cf + dh$.



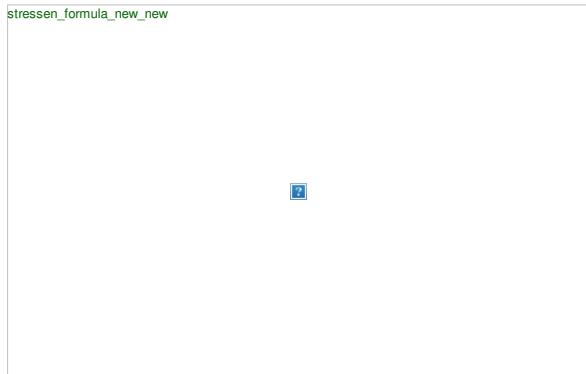
In the above method, we do 8 multiplications for matrices of size $N/2 \times N/2$ and 4 additions. Addition of two matrices takes $O(N^2)$ time. So the time complexity can be written as

$$T(N) = 8T(N/2) + O(N^2)$$

From [Master's Theorem](#), time complexity of above method is $O(N^3)$ which is unfortunately same as the above naive method.

Simple Divide and Conquer also leads to $O(N^3)$, can there be a better way?

In the above divide and conquer method, the main component for high time complexity is 8 recursive calls. The idea of **Strassen's method** is to reduce the number of recursive calls to 7. Strassen's method is similar to above simple divide and conquer method in the sense that this method also divide matrices to sub-matrices of size $N/2 \times N/2$ as shown in the above diagram, but in Strassen's method, the four sub-matrices of result are calculated using following formulae.



Time Complexity of Strassen's Method

Addition and Subtraction of two matrices takes $O(N^2)$ time. So time complexity can be written as

$$T(N) = 7T(N/2) + O(N^2)$$

From [Master's Theorem](#), time complexity of above method is $O(N^{\log_2 7})$ which is approximately $O(N^{2.8074})$

Generally Strassen's Method is not preferred for practical applications for following reasons.

- 1) The constants used in Strassen's method are high and for a typical application Naive method works better.
- 2) For Sparse matrices, there are better methods especially designed for them.
- 3) The submatrices in recursion take extra space.
- 4) Because of the limited precision of computer arithmetic on noninteger values, larger errors accumulate in Strassen's algorithm than in Naive Method (Source: [CLRS Book](#))

Easy way to remember Strassen's Matrix Equation

References:

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest
<https://www.youtube.com/watch?v=LOLebQ8nKHA>
<https://www.youtube.com/watch?v=QXY4RskLQcI>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Divide and Conquer
Divide and Conquer

Create a matrix with alternating rectangles of O and X

Write a code which inputs two numbers m and n and creates a matrix of size m x n (m rows and n columns) in which every elements is either X or 0. The Xs and 0s must be filled alternatively, the matrix should have outermost rectangle of Xs, then a rectangle of 0s, then a rectangle of Xs, and so on.

Examples:

```
Input: m = 3, n = 3
Output: Following matrix
X X X
X 0 X
X X X
```

```
Input: m = 4, n = 5
Output: Following matrix
X X X X X
X 0 0 0 X
X 0 0 0 X
X X X X X
```

```
Input: m = 5, n = 5
Output: Following matrix
X X X X X
X 0 0 0 X
X 0 X 0 X
X 0 0 0 X
X X X X X
```

```
Input: m = 6, n = 7
Output: Following matrix
X X X X X X
X 0 0 0 0 X
X 0 X X 0 X
X 0 X X 0 X
X 0 0 0 0 X
```

XXXXXXXX

We strongly recommend to minimize the browser and try this yourself first.

This question was asked in campus recruitment of Shreepartners Gurgaon. I followed the following approach.

- 1) Use the [code for Printing Matrix in Spiral form](#).
- 2) Instead of printing the array, inserted the element 'X' or '0' alternatively in the array.

Following is C implementation of the above approach.

```
#include <stdio.h>

// Function to print alternating rectangles of 0 and X
void fill0X(int m, int n)
{
    /* k - starting row index
       m - ending row index
       l - starting column index
       n - ending column index
       i - iterator */
    int i, k = 0, l = 0;

    // Store given number of rows and columns for later use
    int r = m, c = n;

    // A 2D array to store the output to be printed
    char a[m][n]; // Initialize the character to be stored in a[]

    // Fill characters in a[] in spiral form. Every iteration fills
    // one rectangle of either Xs or Os
    while (k < m && l < n)
    {
        /* Fill the first row from the remaining rows */
        for (i = l; i < n; ++i)
            a[k][i] = 'X';
        k++;

        /* Fill the last column from the remaining columns */
        for (i = k; i < m; ++i)
            a[i][n-1] = 'X';
        n--;

        /* Fill the last row from the remaining rows */
        if (k < m)
        {
            for (i = n-1; i >= l; --i)
                a[m-1][i] = 'X';
            m--;
        }

        /* Print the first column from the remaining columns */
        if (l < n)
        {
            for (i = m-1; i >= k; --i)
                a[i][l] = 'X';
            l++;
        }

        // Flip character for next iteration
        x = (x == '0') ? 'X' : '0';
    }

    // Print the filled matrix
    for (i = 0; i < r; i++)
    {
        for (int j = 0; j < c; j++)
            printf("%c ", a[i][j]);
        printf("\n");
    }
}

/* Driver program to test above functions */
int main()
{
    puts("Output for m = 5, n = 6");
    fill0X(5, 6);

    puts("\nOutput for m = 4, n = 4");
    fill0X(4, 4);

    puts("\nOutput for m = 3, n = 4");
    fill0X(3, 4);

    return 0;
}
```

Output:

```
Output for m = 5, n = 6
XXXXXX
X0000X
X0XX0X
X0000X
XXXXXX

Output for m = 4, n = 4
XXXX
X00X
X00X
XXXX

Output for m = 3, n = 4
XXXX
X00X
XXXX
```

Time Complexity: $O(mn)$

Auxiliary Space: $O(mn)$

Please suggest if someone has a better solution which is more efficient in terms of space and time.

This article is contributed by **Deepak Bisht**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Print all elements in sorted order from row and column wise sorted matrix

Given an $n \times n$ matrix, where every row and column is sorted in non-decreasing order. Print all elements of matrix in sorted order.

Example:

```
Input: mat[][] = { {10, 20, 30, 40},
                  {15, 25, 35, 45},
                  {27, 29, 37, 48},
                  {32, 33, 39, 50},
                  };
```

```
Output:
Elements of matrix in sorted order
10 15 20 25 27 29 30 32 33 35 37 39 40 45 48 50
```

We strongly recommend to minimize the browser and try this yourself first.

We can use **Young Tableau** to solve the above problem. The idea is to consider given 2D array as Young Tableau and call extract minimum $O(N)$

```
// A C++ program to Print all elements in sorted order from row and
// column wise sorted matrix
#include<iostream>
#include<climits>
using namespace std;

#define INF INT_MAX
#define N 4

// A utility function to youngify a Young Tableau. This is different
// from standard youngify. It assumes that the value at mat[0][0] is
// infinite.
void youngify(int mat[][N], int i, int j)
{
    // Find the values at down and right sides of mat[i][j]
    int downVal = (i+1 < N)? mat[i+1][j]: INF;
    int rightVal = (j+1 < N)? mat[i][j+1]: INF;

    // If mat[i][j] is the down right corner element, return
    if (downVal==INF && rightVal==INF)
        return;

    // Move the smaller of two values (downVal and rightVal) to
    // mat[i][j] and recur for smaller value
    if (downVal < rightVal)
    {
        mat[i][j] = downVal;
        mat[i+1][j] = INF;
        youngify(mat, i+1, j);
    }
    else
    {
        mat[i][j] = rightVal;
        mat[i][j+1] = INF;
        youngify(mat, i, j+1);
    }
}

// A utility function to extract minimum element from Young tableau
int extractMin(int mat[][N])
{
    int ret = mat[0][0];
    mat[0][0] = INF;
    youngify(mat, 0, 0);
    return ret;
}

// This function uses extractMin() to print elements in sorted order
void printSorted(int mat[][N])
{
    cout << "Elements of matrix in sorted order \n";
    for (int i=0; i<N*N; i++)
        cout << extractMin(mat) << " ";
}

// driver program to test above function
int main()
{
    int mat[N][N] = { {10, 20, 30, 40},
                      {15, 25, 35, 45},
                      {27, 29, 37, 48},
                      {32, 33, 39, 50},
                      };
    printSorted(mat);
    return 0;
}
```

Output:

```
Elements of matrix in sorted order
10 15 20 25 27 29 30 32 33 35 37 39 40 45 48 50
```

Time complexity of extract minimum is $O(N)$ and it is called $O(N^2)$ times. Therefore the overall time complexity is $O(N^3)$.

A **better solution** is to use the [approach used for merging k sorted arrays](#). The idea is to use a Min Heap of size N which stores elements of first column. Then do extract minimum. In extract minimum, replace the minimum element with the next element of the row from which the element is extracted. Time complexity of this solution is $O(N^2 \log N)$.

```
// C++ program to merge k sorted arrays of size n each.
#include<iostream>
#include<climits>
using namespace std;

#define N 4

// A min heap node
struct MinHeapNode
{
```

```

int element; // The element to be stored
int i; // index of the row from which the element is taken
int j; // index of the next element to be picked from row
};

// Prototype of a utility function to swap two min heap nodes
void swap(MinHeapNode *x, MinHeapNode *y);

// A class for Min Heap
class MinHeap
{
    MinHeapNode *harr; // pointer to array of elements in heap
    int heap_size; // size of min heap
public:
    // Constructor: creates a min heap of given size
    MinHeap(MinHeapNode a[], int size);

    // to heapify a subtree with root at given index
    void MinHeapify(int i);

    // to get index of left child of node at index i
    int left(int i) { return (2*i + 1); }

    // to get index of right child of node at index i
    int right(int i) { return (2*i + 2); }

    // to get the root
    MinHeapNode getMin() { return harr[0]; }

    // to replace root with new node x and heapify() new root
    void replaceMin(MinHeapNode x) { harr[0] = x; MinHeapify(0); }
};

// This function prints elements of a given matrix in non-decreasing
// order. It assumes that mat[][] is sorted row wise sorted.
void printSorted(int mat[][N])
{
    // Create a min heap with k heap nodes. Every heap node
    // has first element of an array
    MinHeapNode *harr = new MinHeapNode[N];
    for (int i = 0; i < N; i++)
    {
        harr[i].element = mat[i][0]; // Store the first element
        harr[i].i = i; // index of row
        harr[i].j = 1; // Index of next element to be stored from row
    }
    MinHeap hp(harr, N); // Create the min heap

    // Now one by one get the minimum element from min
    // heap and replace it with next element of its array
    for (int count = 0; count < N*N; count++)
    {
        // Get the minimum element and store it in output
        MinHeapNode root = hp.getMin();

        cout << root.element << " ";

        // Find the next element that will replace current
        // root of heap. The next element belongs to same
        // array as the current root.
        if (root.j < N)
        {
            root.element = mat[root.i][root.j];
            root.j += 1;
        }
        // If root was the last element of its array
        else root.element = INT_MAX; //INT_MAX is for infinite

        // Replace root with next element of array
        hp.replaceMin(root);
    }
}

// FOLLOWING ARE IMPLEMENTATIONS OF STANDARD MIN HEAP METHODS
// FROM CORMEN BOOK
// Constructor: Builds a heap from a given array a[] of given size
MinHeap::MinHeap(MinHeapNode a[], int size)
{
    heap_size = size;
    harr = a; // store address of array
    int l = (heap_size - 1)/2;
    while (l >= 0)
    {
        MinHeapify(l);
        l--;
    }
}

// A recursive method to heapify a subtree with root at given index
// This method assumes that the subtrees are already heapified
void MinHeap::MinHeapify(int i)
{
    int l = left(i);
    int r = right(i);
    int smallest = i;
    if (l < heap_size && harr[l].element < harr[i].element)
        smallest = l;
    if (r < heap_size && harr[r].element < harr[smallest].element)
        smallest = r;
    if (smallest != i)
    {
        swap(&harr[i], &harr[smallest]);
        MinHeapify(smallest);
    }
}

// A utility function to swap two elements
void swap(MinHeapNode *x, MinHeapNode *y)
{
    MinHeapNode temp = *x; *x = *y; *y = temp;
}

// driver program to test above function
int main()
{
    int mat[N][N] = { {10, 20, 30, 40},

```

```

        {15, 25, 35, 45},
        {27, 29, 37, 48},
        {32, 33, 39, 50},
    };
    printSorted(mat);
    return 0;
}

```

Output:

```
10 15 20 25 27 29 30 32 33 35 37 39 40 45 48 50
```

Exercise:

Above solutions work for a square matrix. Extend the above solutions to work for an M*N rectangular matrix.

This article is contributed by **Varun**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Heap
Matrix
Matrix

Given an n x n square matrix, find sum of all sub-squares of size k x k

Given an n x n square matrix, find sum of all sub-squares of size k x k where k is smaller than or equal to n.

Examples

Input:
n = 5, k = 3
arr[][] = { {1, 1, 1, 1, 1},
 {2, 2, 2, 2, 2},
 {3, 3, 3, 3, 3},
 {4, 4, 4, 4, 4},
 {5, 5, 5, 5, 5},
 };
Output:
18 18 18
27 27 27
36 36 36

Input:
n = 3, k = 2
arr[][] = { {1, 2, 3},
 {4, 5, 6},
 {7, 8, 9},
 };
Output:
12 16
24 28

A **Simple Solution** is to one by one pick starting point (leftmost-topmost corner) of all possible sub-squares. Once the starting point is picked, calculate sum of sub-square starting with the picked starting point.

Following is C++ implementation of this idea.

```

// A simple C++ program to find sum of all subsquares of size k x k
#include <iostream>
using namespace std;

// Size of given matrix
#define n 5

// A simple function to find sum of all sub-squares of size k x k
// in a given square matrix of size n x n
void printSumSimple(int mat[][n], int k)
{
    // k must be smaller than or equal to n
    if (k > n) return;

    // row number of first cell in current sub-square of size k x k
    for (int i=0; i<n-k+1; i++)
    {
        // column of first cell in current sub-square of size k x k
        for (int j=0; j<n-k+1; j++)
        {
            // Calculate and print sum of current sub-square
            int sum = 0;
            for (int p=i; p<k+i; p++)
                for (int q=j; q<k+j; q++)
                    sum += mat[p][q];
            cout << sum << " ";
        }

        // Line separator for sub-squares starting with next row
        cout << endl;
    }
}

// Driver program to test above function
int main()
{
    int mat[n][n] = {{1, 1, 1, 1, 1},
                     {2, 2, 2, 2, 2},
                     {3, 3, 3, 3, 3},
                     {4, 4, 4, 4, 4},
                     {5, 5, 5, 5, 5},
                     };
    int k = 3;
    printSumSimple(mat, k);
    return 0;
}

```

Output:

```
18 18 18
27 27 27
36 36 36
```

Time complexity of above solution is $O(k^2n^2)$. We can solve this problem in $O(n^2)$ time using a **Tricky Solution**. The idea is to preprocess the given square matrix. In the preprocessing step, calculate sum of all vertical

strips of size $k \times 1$ in a temporary square matrix `stripSum[][]`. Once we have sum of all vertical strips, we can calculate sum of first sub-square in a row as sum of first k strips in that row, and for remaining sub-squares, we can calculate sum in $O(1)$ time by removing the leftmost strip of previous subsquare and adding the rightmost strip of new square.

Following is C++ implementation of this idea.

```
// An efficient C++ program to find sum of all subsquares of size k x k
#include <iostream>
using namespace std;

// Size of given matrix
#define n 5

// A O(n^2) function to find sum of all sub-squares of size k x k
// in a given square matrix of size n x n
void printSumTricky(int mat[][n], int k)
{
    // k must be smaller than or equal to n
    if (k > n) return;

    // 1: PREPROCESSING
    // To store sums of all strips of size k x 1
    int stripSum[n][n];

    // Go column by column
    for (int j=0; j<n; j++)
    {
        // Calculate sum of first k x 1 rectangle in this column
        int sum = 0;
        for (int i=0; i<k; i++)
            sum += mat[i][j];
        stripSum[0][j] = sum;

        // Calculate sum of remaining rectangles
        for (int i=1; i<n-k+1; i++)
        {
            sum += (mat[i+k-1][j] - mat[i-1][j]);
            stripSum[i][j] = sum;
        }
    }

    // 2: CALCULATE SUM of Sub-Squares using stripSum[][]
    for (int i=0; i<n-k+1; i++)
    {
        // Calculate and print sum of first subsquare in this row
        int sum = 0;
        for (int j = 0; j<k; j++)
            sum += stripSum[j][i];
        cout << sum << " ";

        // Calculate sum of remaining squares in current row by
        // removing the leftmost strip of previous sub-square and
        // adding a new strip
        for (int j=1; j<n-k+1; j++)
        {
            sum += (stripSum[j][i+k-1] - stripSum[j-1][i]);
            cout << sum << " ";
        }

        cout << endl;
    }
}

// Driver program to test above function
int main()
{
    int mat[][n] = {{1, 1, 1, 1, 1},
                    {2, 2, 2, 2, 2},
                    {3, 3, 3, 3, 3},
                    {4, 4, 4, 4, 4},
                    {5, 5, 5, 5, 5},
                    };
    int k = 3;
    printSumTricky(mat, k);
    return 0;
}
```

Output:

```
18 18 18
27 27 27
36 36 36
```

This article is contributed by **Rahul Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Matrix
Matrix

Count number of islands where every island is row-wise and column-wise separated

Given a rectangular matrix which has only two possible values 'X' and 'O'. The values 'X' always appear in form of rectangular islands and these islands are always row-wise and column-wise separated by at least one line of 'O's. Note that islands can only be diagonally adjacent. Count the number of islands in the given matrix.

Examples:

```
mat[M][N] = {{'O', 'O', 'O'},
              {'X', 'X', 'O'},
              {'X', 'X', 'O'},
              {'O', 'O', 'X'},
              {'O', 'O', 'X'},
              {'X', 'X', 'O'}
              };

Output: Number of islands is 3

mat[M][N] = {{'X', 'O', 'O', 'O', 'O', 'O'},
              {'X', 'O', 'X', 'X', 'X', 'X'},
              {'O', 'O', 'O', 'O', 'O', 'O'},
              {'X', 'X', 'X', 'O', 'X', 'X'},
              {'X', 'X', 'X', 'O', 'X', 'X'},
              {'O', 'O', 'O', 'O', 'X', 'X'},
              };

```

Output: Number of islands is 4

We strongly recommend to minimize your browser and try this yourself first.

The idea is to count all top-leftmost corners of given matrix. We can check if a 'X' is top left or not by checking following conditions.

- 1) A 'X' is top of rectangle if the cell just above it is a 'O'
- 2) A 'X' is leftmost of rectangle if the cell just left of it is a 'O'

Note that we must check for both conditions as there may be more than one top cells and more than one leftmost cells in a rectangular island. Below is C++ implementation of above idea.

```
// A C++ program to count the number of rectangular
// islands where every island is separated by a line
#include<iostream>
using namespace std;

// Size of given matrix is M X N
#define M 6
#define N 3

// This function takes a matrix of 'X' and 'O'
// and returns the number of rectangular islands
// of 'X' where no two islands are row-wise or
// column-wise adjacent, the islands may be diagonally
// adjacent
int countIslands(int mat[][N])
{
    int count = 0; // Initialize result

    // Traverse the input matrix
    for (int i=0; i<M; i++)
    {
        for (int j=0; j<N; j++)
        {
            // If current cell is 'X', then check
            // whether this is top-leftmost of a
            // rectangle. If yes, then increment count
            if (mat[i][j] == 'X')
            {
                if ((i == 0 || mat[i-1][j] == 'O') &&
                    (j == 0 || mat[i][j-1] == 'O'))
                    count++;
            }
        }
    }

    return count;
}

// Driver program to test above function
int main()
{
    int mat[M][N] = {{ 'O', 'O', 'O' },
                     { 'X', 'X', 'O' },
                     { 'X', 'X', 'O' },
                     { 'O', 'O', 'X' },
                     { 'O', 'O', 'X' },
                     { 'X', 'X', 'O' } };

    cout << "Number of rectangular islands is "
    << countIslands(mat);
    return 0;
}
```

Output:

Number of rectangular islands is 3

Time complexity of this solution is $O(MN)$.

This article is contributed by **Udit Gupta**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

GATE CS Corner Company Wise Coding Practice

Matrix

Find a common element in all rows of a given row-wise sorted matrix

Given a matrix where every row is sorted in increasing order. Write a function that finds and returns a common element in all rows. If there is no common element, then returns -1.

Example:

```
Input: mat[4][5] = { { 1, 2, 3, 4, 5 },
                    { 2, 4, 5, 8, 10 },
                    { 3, 5, 7, 9, 11 },
                    { 1, 3, 5, 7, 9 },
                    };
Output: 5
```

A **$O(m \cdot n)$ simple solution** is to take every element of first row and search it in all other rows, till we find a common element. Time complexity of this solution is $O(m \cdot n \cdot n)$ where m is number of rows and n is number of columns in given matrix. This can be improved to $O(m \cdot n \cdot \log n)$ if we use [Binary Search](#) instead of linear search.

We can solve this problem in **$O(mn)$ time** using the approach similar to merge of [Merge Sort](#). The idea is to start from the last column of every row. If elements at all last columns are same, then we found the common element. Otherwise we find the minimum of all last columns. Once we find a minimum element, we know that all other elements in last columns cannot be a common element, so we reduce last column index for all rows except for the row which has minimum value. We keep repeating these steps till either all elements at current last column don't become same, or a last column index reaches 0.

Below is C implementation of above idea.

```
// A C program to find a common element in all rows of a
// row wise sorted array
#include<stdio.h>

// Specify number of rows and columns
#define M 4
#define N 5

// Returns common element in all rows of mat[M][N]. If there is no
// common element, then -1 is returned
int findCommon(int mat[M][N])
{
    // An array to store indexes of current last column
```

```

int column[M];
int min_row; // To store index of row whose current
             // last element is minimum

// Initialize current last element of all rows
int i;
for (i=0; i<M; i++)
    column[i] = N-1;

min_row = 0; // Initialize min_row as first row

// Keep finding min_row in current last column, till either
// all elements of last column become same or we hit first column.
while (column[min_row] >= 0)
{
    // Find minimum in current last column
    for (i=0; i<M; i++)
    {
        if (mat[i][column[i]] < mat[min_row][column[min_row]])
            min_row = i;
    }

    // eq_count is count of elements equal to minimum in current last
    // column.
    int eq_count = 0;

    // Travers current last column elements again to update it
    for (i=0; i<M; i++)
    {
        // Decrease last column index of a row whose value is more
        // than minimum.
        if (mat[i][column[i]] > mat[min_row][column[min_row]])
        {
            if (column[i] == 0)
                return -1;

            column[i] -= 1; // Reduce last column index by 1
        }
        else
            eq_count++;
    }

    // If equal count becomes M, return the value
    if (eq_count == M)
        return mat[min_row][column[min_row]];
    }
    return -1;
}

// driver program to test above function
int main()
{
    int mat[M][N] = { { 1, 2, 3, 4, 5},
                      { 2, 4, 5, 8, 10},
                      { 3, 5, 7, 9, 11},
                      { 1, 3, 5, 7, 9},
                      };
    int result = findCommon(mat);
    if (result == -1)
        printf("No common element");
    else
        printf("Common element is %d", result);
    return 0;
}

```

Output:

```
Common element is 5
```

Explanation for working of above code

Let us understand working of above code for following example.

Initially entries in last column array are N-1, i.e., {4, 4, 4, 4}

```

{1, 2, 3, 4, 5},
{2, 4, 5, 8, 10},
{3, 5, 7, 9, 11},
{1, 3, 5, 7, 9},

```

The value of min_row is 0, so values of last column index for rows with value greater than 5 is reduced by one. So column[] becomes {4, 3, 3, 3}.

```

{1, 2, 3, 4, 5},
{2, 4, 5, 8, 10},
{3, 5, 7, 9, 11},
{1, 3, 5, 7, 9},

```

The value of min_row remains 0 and value of last column index for rows with value greater than 5 is reduced by one. So column[] becomes {4, 2, 2, 2}.

```

{1, 2, 3, 4, 5},
{2, 4, 5, 8, 10},
{3, 5, 7, 9, 11},
{1, 3, 5, 7, 9},

```

The value of min_row remains 0 and value of last column index for rows with value greater than 5 is reduced by one. So column[] becomes {4, 2, 1, 2}.

```

{1, 2, 3, 4, 5},
{2, 4, 5, 8, 10},
{3, 5, 7, 9, 11},
{1, 3, 5, 7, 9},

```

Now all values in current last columns of all rows is same, so 5 is returned.

A Hashing Based Solution

We can also use hashing. This solution works even if the rows are not sorted. It can be used to print all common elements.

```

Step1: Create a Hash Table with all key as distinct elements
of row 1. Value for all these will be 0.

```

```

Step2:

```

```

For i = 1 to M-1

```

```

For j = 0 to N-1

```

```

If (mat[i][j]) is already present in Hash Table)

```

```

If (And this is not a repetition in current row.

```

```

    This can be checked by comparing HashTable value with
    row number)
    Update the value of this key in HashTable with current

```

row number

Step3: Iterate over HashTable and print all those keys for which value = M

Time complexity of the above hashing based solution is $O(MN)$ under the assumption that search and insert in HashTable take $O(1)$ time. Thanks to Nishant for suggesting this solution in a comment below.

Exercise: Given n sorted arrays of size m each, find all common elements in all arrays in $O(mn)$ time.

This article is contributed by **Anand Agrawal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Matrix
Matrox

Commonly Asked Data Structure Interview Questions | Set 1

What is a Data Structure?

A data structure is a way of organizing the data so that the data can be used efficiently. Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks. For example, B-trees are particularly well-suited for implementation of databases, while compiler implementations usually use hash tables to look up identifiers. (Source: [Wiki Page](#))

What are linear and non linear data Structures?

- **Linear:** A data structure is said to be linear if its elements form a sequence or a linear list. Examples: Array, Linked List, Stacks and Queues
- **Non-Linear:** A data structure is said to be linear if traversal of nodes is nonlinear in nature. Example: Graph and Trees.

What are the various operations that can be performed on different Data Structures?

- **Insertion** – Add a new data item in the given collection of data items.
- **Deletion** – Delete an existing data item from the given collection of data items.
- **Traversal** – Access each data item exactly once so that it can be processed.
- **Searching** – Find out the location of the data item if it exists in the given collection of data items.
- **Sorting** – Arranging the data items in some order i.e. in ascending or descending order in case of numerical data and in dictionary order in case of alphanumeric data.

How is an Array different from Linked List?

- The size of the arrays is fixed, Linked Lists are Dynamic in size.
- Inserting and deleting a new element in an array of elements is expensive, Whereas both insertion and deletion can easily be done in Linked Lists.
- Random access is not allowed in Linked Listed.
- Extra memory space for a pointer is required with each element of the Linked list.
- Arrays have better cache locality that can make a pretty big difference in performance.

What is Stack and where it can be used?

Stack is a linear data structure which the order LIFO (Last In First Out) or FILO (First In Last Out) for accessing elements. Basic operations of stack are : **Push, Pop, Peek**

Applications of Stack:

1. [Infix to Postfix Conversion using Stack](#)
2. [Evaluation of Postfix Expression](#)
3. [Reverse a String using Stack](#)
4. [Implement two stacks in an array](#)
5. [Check for balanced parentheses in an expression](#)

What is a Queue, how it is different from stack and how is it implemented?

Queue is a linear structure which follows the order is **First In First Out (FIFO)** to access elements. Mainly the following are basic operations on queue: **Enqueue, Dequeue, Front, Rear**

The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added. Both Queues and Stacks can be implemented using Arrays and Linked Lists.

What are Infix, prefix, Postfix notations?

- **Infix notation:** $X + Y$ – Operators are written in-between their operands. This is the usual way we write expressions. An expression such as

$A * (B + C) / D$

- **Postfix notation (also known as “Reverse Polish notation”):** $X Y +$ Operators are written after their operands. The infix expression given above is equivalent to

$A B C + * D /$

- **Prefix notation (also known as “Polish notation”):** $+ X Y$ Operators are written before their operands. The expressions given above are equivalent to

$/ * A + B C D$

Converting between these notations: [Click here](#)

What is a Linked List and What are its types?

A linked list is a linear data structure (like arrays) where each element is a separate object. Each element (that is node) of a list is comprising of two items – the data and a reference to the next node. Types of Linked List :

1. **Singly Linked List** : In this type of linked list, every node stores address or reference of next node in list and the last node has next address or reference as NULL. For example $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow \text{NULL}$
2. **Doubly Linked List** : Here, there are two references associated with each node, One of the reference points to the next node and one to the previous node. Eg. $\text{NULL} \leftarrow 1 \leftrightarrow 2 \leftrightarrow 3 \rightarrow \text{NULL}$
3. **Circular Linked List** : Circular linked list is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list. Eg. $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$ [The next pointer of last node is pointing to the first]

Which data structures are used for BFS and DFS of a graph?

- **Queue is used for BFS**
- Stack is used for DFS. DFS can also be implemented using recursion (Note that recursion also uses function call stack).

Can doubly linked be implemented using a single pointer variable in every node?

Doubly linked list can be implemented using a single pointer. See [XOR Linked List – A Memory Efficient Doubly Linked List](#)

How to implement a stack using queue?

A stack can be implemented using two queues. Let stack to be implemented be 's' and queues used to implement be 'q1' and 'q2'. Stack 's' can be implemented in two ways:

- Method 1 (By making push operation costly)
- Method 2 (By making pop operation costly) See [Implement Stack using Queues](#)

How to implement a queue using stack?

A queue can be implemented using two stacks. Let queue to be implemented be q and stacks used to implement q be stack1 and stack2. q can be implemented in two ways:

- Method 1 (By making enqueue operation costly)
- Method 2 (By making dequeue operation costly) See [Implement Queue using Stacks](#)

Which Data Structure Should be used for implementing LRU cache?

We use two data structures to implement an LRU Cache.

1. **Queue** which is implemented using a doubly linked list. The maximum size of the queue will be equal to the total number of frames available (cache size). The most recently used pages will be near front end and least recently pages will be near rear end.
2. **A Hash** with page number as key and address of the corresponding queue node as value. See [How to implement LRU caching scheme? What data structures should be used?](#)

How to check if a given Binary Tree is BST or not?

If inorder traversal of a binary tree is sorted, then the binary tree is BST. The idea is to simply do inorder traversal and while traversing keep track of previous key value. If current key value is greater, then continue, else return false. See [A program to check if a binary tree is BST or not](#) for more details.

Linked List Questions

- [Linked List Insertion](#)
- [Linked List Deletion](#)
- [middle of a given linked list](#)
- [Nth node from the end of a Linked List](#)

Tree Traversal Questions

- [Inorder](#)
- [Preorder and Postorder Traversals](#)
- [Level order traversal](#)
- [Height of Binary Tree](#)

Convert a DLL to Binary Tree in-place

See [In-place conversion of Sorted DLL to Balanced BST](#)

Convert Binary Tree to DLL in-place

See [Convert a given Binary Tree to Doubly Linked List | Set 1](#), [Convert a given Binary Tree to Doubly Linked List | Set 2](#)

Delete a given node in a singly linked list

Given only a pointer to a node to be deleted in a singly linked list, how do you delete it?

Reverse a Linked List

Write a function to reverse a linked list

Detect Loop in a Linked List

Write a C function to detect loop in a linked list.

Which data structure is used for dictionary and spell checker?

[Data Structure for Dictionary and Spell Checker?](#)

You may also like

- [Practice Quizzes on Data Structures](#)
- [Last Minute Notes – DS](#)
- [Common Interview Puzzles](#)
- [Amazon's most asked interview questions](#)
- [Microsoft's most asked interview questions](#)
- [Accenture's most asked Interview Questions](#)
- [Commonly Asked OOP Interview Questions](#)
- [Commonly Asked C++ Interview Questions,](#)
- [Commonly Asked C Programming Interview Questions | Set 1](#)
- [Commonly Asked C Programming Interview Questions | Set 2](#)
- [Commonly asked DBMS interview questions | Set 1](#)
- [Commonly Asked Operating Systems Interview Questions | Set 1](#)
- [Commonly Asked Data Structure Interview Questions](#)
- [Commonly Asked Algorithm Interview Questions](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.

A data structure for n elements and O(1) operations

Propose a data structure for the following:

The data structure would hold elements from 0 to n-1. There is no order on the elements (no ascending/descending order requirement)

The complexity of the operations should be as follows:

- * Insertion of an element – O(1)
- * Deletion of an element – O(1)
- * Finding an element – O(1)

We strongly recommend to minimize the browser and try this yourself first.

A boolean array works here. Array will have value 'true' at ith index if i is present, and 'false' if absent.

Initialization:

We create an array of size n and initialize all elements as absent.

```
void initialize(int n)
{
    bool A[n];
    for (int i = 0; i < n; i++)
        A[i] = 0; // or A[i] = {false};
}
```

Insertion of an element:

```
void insert(unsigned i)
{
    /* set the value at index i to true */
    A[i] = 1; // Or A[i] = true;
}
```

Deletion of an element:

```
void delete(unsigned i)
{
    /* make the value at index i to 0 */
    A[i] = 0; // Or A[i] = false;
}
```

Finding an element:

```
// Returns true if 'i' is present, else false
bool find(unsigned i)
{
    return A[i];
}
```

As an exercise, change the data structure so that it holds values from 1 to n instead of 0 to n-1.

This article is contributed by **Sachin**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.

Category: Data Structures

Expression Tree

Expression tree is a binary tree in which each internal node corresponds to operator and each leaf node corresponds to operand so for example expression tree for $3 + ((5+9)*2)$ would be:

expressiontree



Inorder traversal of expression tree produces infix version of given postfix expression (same with preorder traversal it gives prefix expression)

Evaluating the expression represented by expression tree:

```
Let t be the expression tree
If t is not null then
    If t.value is operand then
        Return t.value
    A = solve(t.left)
    B = solve(t.right)

    // calculate applies operator t.value'
    // on A and B, and returns value
    Return calculate(A, B, t.value)
```

Construction of Expression Tree:

Now For constructing expression tree we use a stack. We loop through input expression and do following for every character.

- 1) If character is operand push that into stack
- 2) If character is operator pop two values from stack make them its child and push current node again.

At the end only element of stack will be root of expression tree.

Below is the implementation :

C/C++

```
// C++ program for expression tree
#include<bits/stdc++.h>
using namespace std;

// An expression tree node
struct et
{
    char value;
    et* left, *right;
};

// A utility function to check if 'c'
// is an operator
bool isOperator(char c)
{
    if (c == '+' || c == '-' ||
        c == '*' || c == '/' ||
        c == '^')
        return true;
    return false;
}

// Utility function to do inorder traversal
void inorder(et *t)
{
    if(t)
    {
        inorder(t->left);
        printf("%c ", t->value);
        inorder(t->right);
    }
}

// A utility function to create a new node
et* newNode(int v)
{
    et *temp = new et;
    temp->left = temp->right = NULL;
    temp->value = v;
    return temp;
};

// Returns root of constructed tree for given
// postfix expression
et* constructTree(char postfix[])
{
    stack<et*> st;
    et *t, *t1, *t2;

    // Traverse through every character of
    // input expression
    for (int i=0; i<strlen(postfix); i++)
    {
        // If operand, simply push into stack
        if (!isOperator(postfix[i]))
        {
            t = newNode(postfix[i]);
            st.push(t);
        }
        else // operator
        {
            t = newNode(postfix[i]);

            // Pop two top nodes
            t1 = st.top(); // Store top
            st.pop(); // Remove top
            t2 = st.top();
            st.pop();

            // make them children
            t->right = t1;
            t->left = t2;

            // Add this subexpression to stack
            st.push(t);
        }
    }

    // only element will be root of expression
    // tree
    t = st.top();
    st.pop();

    return t;
}

// Driver program to test above
int main()
{
    char postfix[] = "ab+e*g*-";
    et* t = constructTree(postfix);
    printf("Infix expression is \n");
    inorder(t);
    return 0;
}
```

Java

```
// Java program to construct an expression tree

import java.util.Stack;

// Java program for expression tree
class Node {

    char value;
    Node left, right;

    Node(char item) {
        value = item;
    }
}
```

```

        left = right = null;
    }
}

class ExpressionTree {

    // A utility function to check if 'c'
    // is an operator

    boolean isOperator(char c) {
        if (c == '+' || c == '-' || c == '*' || c == '/')
            return true;
        else
            return false;
    }

    // Utility function to do inorder traversal
    void inorder(Node t) {
        if (t != null) {
            inorder(t.left);
            System.out.print(t.value + " ");
            inorder(t.right);
        }
    }

    // Returns root of constructed tree for given
    // postfix expression
    Node constructTree(char postfix[]) {
        Stack<Node> st = new Stack();
        Node t, t1, t2;

        // Traverse through every character of
        // input expression
        for (int i = 0; i < postfix.length; i++) {

            // If operand, simply push into stack
            if (!isOperator(postfix[i])) {
                t = new Node(postfix[i]);
                st.push(t);
            } else // operator
            {
                t = new Node(postfix[i]);

                // Pop two top nodes
                // Store top
                t1 = st.pop(); // Remove top
                t2 = st.pop();

                // make them children
                t.right = t1;
                t.left = t2;

                // System.out.println(t1 + "" + t2);
                // Add this subexpression to stack
                st.push(t);
            }
        }

        // only element will be root of expression
        // tree
        t = st.peek();
        st.pop();

        return t;
    }

    public static void main(String args[]) {

        ExpressionTree et = new ExpressionTree();
        String postfix = "ab+efg*.";
        char[] charArray = postfix.toCharArray();
        Node root = et.constructTree(charArray);
        System.out.println("infix expression is");
        et.inorder(root);

    }
}

// This code has been contributed by Mayank Jaiswal

```

Python

```

# Python program for expression tree

# An expression tree node
class Et:

    # Constructor to create a node
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

# A utility function to check if 'c'
# is an operator
def isOperator(c):
    if (c == '+' or c == '-' or c == '*'
        or c == '/ or c == '^'):
        return True
    else:
        return False

# A utility function to do inorder traversal
def inorder(t):
    if t is not None:
        inorder(t.left)
        print t.value,
        inorder(t.right)

# Returns root of constructed tree for

```

```

# given postfix expression
def constructTree(postfix):
    stack = []

    # Traverse through every character of input expression
    for char in postfix :

        # if operand, simply push into stack
        if not isOperator(char):
            t = Et(char)
            stack.append(t)

        # Operator
        else:

            # Pop two top nodes
            t = Et(char)
            t1 = stack.pop()
            t2 = stack.pop()

            # make them children
            t.right = t1
            t.left = t2

            # Add this subexpression to stack
            stack.append(t)

    # Only element will be the root of expression tree
    t = stack.pop()

    return t

# Driver program to test above
postfix = "ab+ef*g*-"
r = constructTree(postfix)
print "Infix expression is"
inorder(r)

```

Output:

```

infix expression is
a + b - e * f * g

```

This article is contributed by [Utkarsh Trivedi](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner **Company Wise Coding Practice**

Trees