

Data Structures and Algorithms

-MAYANK PATEL



What is Data Structure?

A **data structure** is a particular way of organizing data in a computer so that it can be used effectively.

TYPES OF DATA STRUCTURES

LINEAR DATA STRUCTURES:

Arrays

String

Linked List

Stack

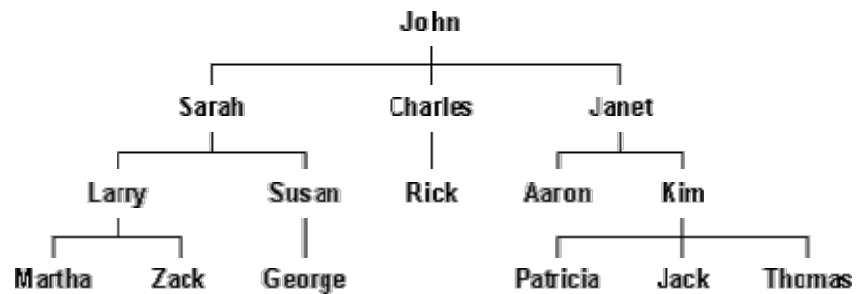
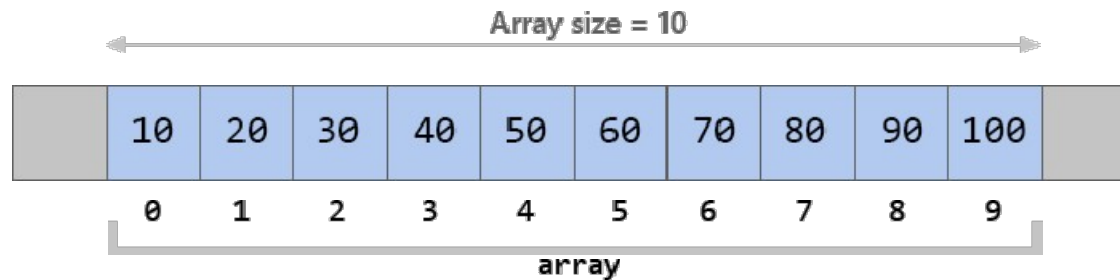
Queue

NON-LINEAR DATA STRUCTURES:

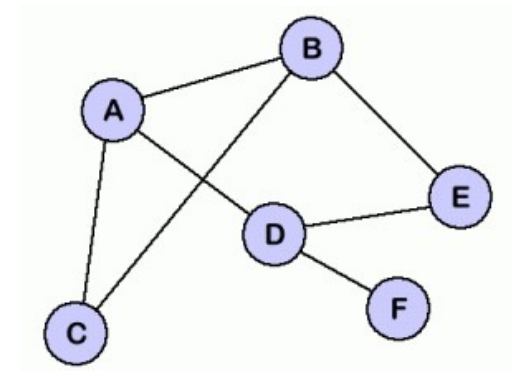
Trees

Graphs

Examples of some Data Structures



Family represented using a Tree DS



A network of friends represented using Graphs(Facebook)

Advantages of using Data Structures.

1. Efficient
2. Reusable
3. Support Abstarction

Some common operations on DS

1. Insertion
2. Deletion
3. Traversal
4. Search

Algorithms

An algorithm is a process of well defined steps used for solving a problem.

In DSA we can say:

They are used to manipulate the Data Structures or gain some useful information through them.



Analysis of an Algorithm

1. Space - Space **complexity** of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the length of the input.
2. Time -Time **complexity** of an algorithm quantifies the amount of **time** taken by an algorithm to run as a function of the length of the input.

Time Complexity Analysis

$$f(n) = 5n^2 + 6n + 12$$

If $n = 10$

% of time due to $5n^2$: $(500/(500+60+12))*100 = 87.41\%$

% of time due to $6n$: $(60/(500+60+12))*100 = 10.49\%$

% of time due to : $(12/(500+60+12))*100 = 2.09\%$

n	$5n^2$	$6n$	12
1	21.74%	26%	52%
10	87.41%	10.49%	2.09%
100	98.79%	1.19%	0.02%
1000	99.8%	0.12%	0.0002%

Order of growth

It is how the time of execution depends on the length of the input. In the above example, we can clearly see that the time of execution depends on the length of the input. Order of growth will help us to compute the running time with ease. We will ignore the lower order terms, since the lower order terms are relatively insignificant for large input. We use different notation to describe limiting behavior of a function.

Notations to measure growth

$O(n)$ Big-oh Notation – WORST CASE TIME COMPLEXITY

$\Omega(n)$ – Big-omega Notation – BEST CASE TIME COMPLEXITY

$\Theta(n)$ – Big-theta Notation



O(n) Notation

```
int count = 0;
for (int i = 0; i < N; i++)
    for (int j = 0; j < i; j++)
        count++;
```

Question-1

```
int count = 0;
for (int i = N; i > 0; i /= 2)
    for (int j = 0; j < i; j++)
        count++;
```


Question-2

Recursion vs Iteration

Recursion vs Iteration

The **Recursion** and **Iteration** both repeatedly execute the set of instructions. **Recursion** is when a statement in a function **calls itself repeatedly**. The **iteration** is when a loop **repeatedly executes until the controlling condition becomes false**.

The primary difference between recursion and iteration is that **recursion** is a process, always applied to a function and **iteration** is applied to the **set of instructions** which we want to get **repeatedly executed**.



Examples of Recursion and Iteration

```
function factorial(number) {  
  if (number === 1) {  
    return 1;  
  }  
  return number * factorial(number - 1);  
}
```

Recursion

```
var step;  
for (step = 0; step < 5; step++) {  
  // Runs 5 times, with values of step 0 through 4.  
  console.log('Walking east one step');  
}
```

Iteration

PROPERTY	RECURSION	ITERATION
Definition	Function calls itself.	A set of instructions repeatedly executed.
Application	For functions.	For loops.
Termination	Through base case, where there will be no function call.	When the termination condition for the iterator ceases to be satisfied.
Usage	Used when code size needs to be small, and time complexity is not an issue.	Used when time complexity needs to be balanced against an expanded code size.
Code Size	Smaller code size	Larger Code Size.
Time Complexity	Very high(generally exponential) time complexity.	Relatively lower time complexity(generally polynomial-logarithmic).

ARRAYS

Arrays

An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array).

Array is defined as a set of homogeneous data items.

An Array is a group of elements that share a common name that are differentiated from one another by their positions within the array

DECLARATION OF AN ARRAY



```
datatype arr_name[size];
```

POINTS TO BE NOTED :

- 1) Arrayname should be a valid “C” variable**
- 2) Arrayname should be unique**
- 3) The elements in the array should be of same type**
- 4) Subscript (array size) cannot be negative**
- 5) Subscript must always be an integer**

How to declare an array?

1. **Array declaration by specifying size**
2. **Array declaration by initializing elements**
3. **Array declaration by specifying size and initializing elements**

TYPES OF ARRAY

One Dimensional Array

Two Dimensional Array

Multi Dimensional Array

Single or One Dimensional Arrays

➤ Arrays whose elements are specified by one subscript are called One dimensional array or linear array.

➤ Syntax :

datatype arrayname[size];

➤ *For Example :*

int a [3]

➤ Note :

By default array index should starts with zero (0)

Two Dimensional Arrays

- A Arrays whose elements are specified by two subscript such as row and column are called One dimensional array or linear array.
- Row → means horizontally
- Column → means vertically
- A two - dimensional array looks like a school time-table consisting of rows and columns.
- A two – dimensional array is declared as -

int a [3] [3]

Two Dimensional Array Initialization

int ary [3] [4] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };

➤ The result of the above assignment will be as follows :

ary [0] [0] = 1	ary [0] [1] = 2	ary [0] [2] = 3	ary [0] [3] = 4
ary [1] [0] = 5	ary [1] [1] = 6	ary [1] [2] = 7	ary [1] [3] = 8
ary [2] [0] = 9	ary [2] [1] = 10	ary [2] [2] = 11	ary [2] [3] = 12

(OR)

**int ary [3] [4] =
{
 { 1, 2, 3 },
 { 4, 5, 6 },
 { 7, 8, 9 },
 { 10, 11, 12 }
};**

Write a “C” program to find the largest and smallest numbers given in the array

Write a “C” program to sort the given number is in ascending order using one dimensional array

Write a “C” program to perform the addition of two matrices


Write a “C” program to perform the subtraction of two matrices

Write a “C” program to perform matrix multiplication using two dimensional array

Session Summary

- ✍ **Arrayname should be a unique and valid “C” Variable name**
- ✍ **The number of elements in a multi dimensional array is the product of its subscripts**
- ✍ **Arrays can be initialized to the same type in which they are declared**
- ✍ **The character array receives the terminating ‘\0’ in the string constant**
- ✍ **The individual values in the array are called as elements**
- ✍ **It is not necessary to specify the length of an array, explicitly in case if initializers are provided for the array during declaration itself**

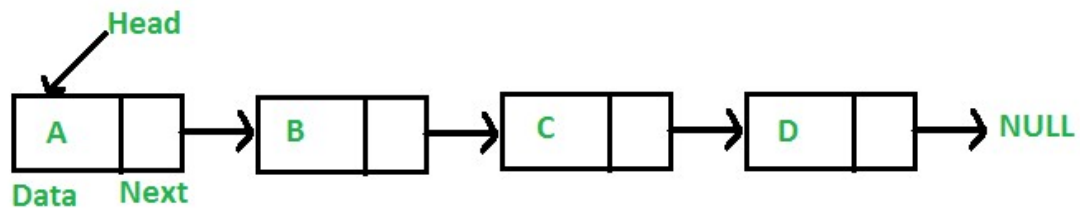
EXERCISES

1. Write a program to search an element and to find how many times it is present in the array?
 2. Write a program to find the sum of diagonal elements in a 4x4 matrix.
 3. Write a program to find the second largest number in an array?
 4. Write a program to remove the duplicate elements of the array?
 5. Write a program to merge two arrays and print the merged array in ascending order?
 6. Write a program to insert an element into an sorted array of integers?
 7. Write a program to display only the negative elements of the array?
- 

Linked Lists

Linked List

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:



Why Linked List / Advantages?

Arrays can be used to store linear data of similar types, but arrays have the 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 the room has to be created for the new elements and to create room existing elements have to be shifted.

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 efficiently with its default implementation.
- 2) Extra memory space for a pointer is required with each element of the list.

CREATION OF A LINKED LIST

```
struct Node{
    int data;
    struct Node* next;
};

int main()
{
    Node* head = new Node();
    head->data=1;
    head->next=NULL;

    head->next = new Node();
    head->next->data=2;
    head->next->next=NULL;

    head->next->next=new Node();
    head->next->next->data=3;
    head->next->next->next=NULL;

    head->next->next->next = new Node();
    head->next->next->next->data=4;
    head->next->next->next->next=NULL;

    cout<<head->next->next->next->data;
}
```

TRAVERSAL IN A LINKED LIST

```
|  
Node* ptr=head;  
  
while(ptr!=NULL)  
{  
    cout<<ptr->data<<"-->";  
    ptr=ptr->next;  
}
```

INSERT AT END OF A LINKED LIST

```
//insert at end of a Linked List
void insert_end(Node *head, int item)
{
    Node *temp=head;

    while(temp->next!=NULL)
    {
        temp=temp->next;
    }

    temp->next=new Node();
    temp->next->data = item;
    temp->next->next = NULL;
}
```

INSERT AT BEGINNING OF LINKED LIST

```
//insert at beggining of a linked list
Node* insert_beg(Node* head,int item)
{
    Node* temp= new Node();

    temp->data = item;
    temp->next = head;
    head=temp;

    return head;
}
```

INSERT AT Nth POSITION IN A LINKED LIST

```
// insert at a position n
// for position greater than or equal to 2
// Home work to change it to work for positions 0,1 as well.
Node* insert_at_pos(Node* head,int item,int pos)
{
    pos--;

    Node* ptr=head,*temp;

    while(pos!=1&&ptr!=NULL)
    {
        ptr=ptr->next;
        pos--;
    }

    temp= new Node();
    temp->data=item;
    temp->next = ptr->next;
    ptr->next =temp;

    return head;
}
```


DELETE LAST ELEMENT OF A LINKED LIST

```
//delete last elemnt
// for list having size greater than 2
// Home work to change it to work for all sizes.
Node * delete_last(Node* head)
{
    Node* ptr=head;

    while(ptr->next->next!=NULL)
    {
        ptr=ptr->next;
    }

    delete(ptr->next);
    ptr->next=NULL;
}
|
```

SEARCH IN A LINKED LIST

```
//search in a linked list  
//Home work change it to also print the position of element if found  
void search(Node* head,int item)  
{  
    Node* ptr=head;  
  
    while(ptr!=NULL)  
    {  
        if(ptr->data==item)  
        {  
            cout<<item<<" found"<<endl;  
            return;  
        }  
        ptr=ptr->next;  
    }  
  
    cout<<item<<" not found"<<endl;  
}
```

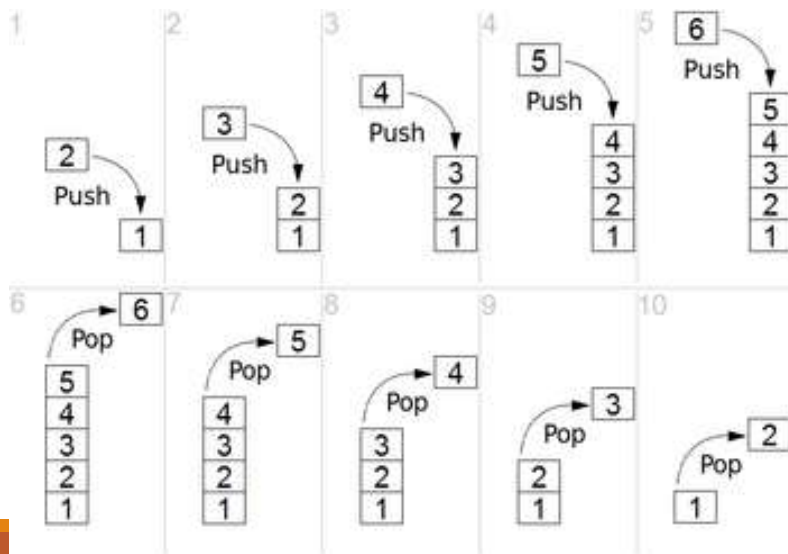
Self-Read

1. Detect loop in a linked list.
2. Find length of loop in a linked list.
3. Pair-wise swap elements of linked list.
4. Reverse a linked list.
5. Find intersection of two linked lists.
6. Segregate even and odd nodes(not elements) in a linked list(make two new linked lists)

Stacks

Stack

Stack is a linear data structure which follows a particular order in which the operations are performed. The order is LIFO (Last In First Out)



Any item will be removed from the top



New item will be added on the top

In-built methods in Stack

The functions associated with stack are:

[empty\(\)](#) – Returns whether the stack is empty – Time Complexity : $O(1)$

[size\(\)](#) – Returns the size of the stack – Time Complexity : $O(1)$

[top\(\)](#) – Returns a reference to the top most element of the stack – Time Complexity : $O(1)$

[push\(g\)](#) – Adds the element 'g' at the top of the stack – Time Complexity : $O(1)$

[pop\(\)](#) – Deletes the top most element of the stack – Time Complexity : $O(1)$



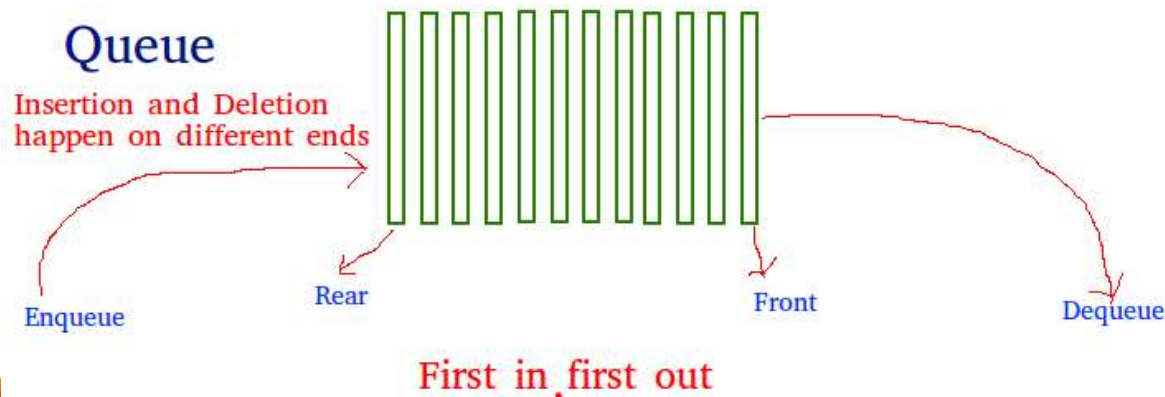
Self-Read

1. What is an infix expression, postfix expression, prefix expression.
2. Conversion of Infix expression to postfix expression using stacks
3. Prefix to Infix conversion.
4. Prefix to Postfix conversion.
5. Evaluation of postfix expression.
6. Tower of Hanoi problem(stacks/recursion both approaches).

Queues

Queue

A 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 a 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.



Questions

1. Implement stacks in arrays, (a)with last element as top (b)with a[0] as top element.
2. Implement a queue in an array, such that all elements are shifted to left, if rear reaches end.
3. WAP to implement two stacks in a single array that supports function (pushA(), pushB(), popA(), popB()).
4. WAP to implement two queues in a single array, queue should not overflow if there is even a single slot available in array.
5. Implement a queue using two stacks
6. Implement a stack using a queue.
7. WAP to convert a decimal number to binary using stacks

Trees

Tree

Tree is a hierarchical data structure which stores the information naturally in the form of hierarchy style.

Tree is one of the most powerful and advanced data structures.

It is a non-linear data structure compared to arrays, linked lists, stack and queue.

It represents the nodes connected by edges.

Advantages of using Tree DS

Tree reflects structural relationships in the data.

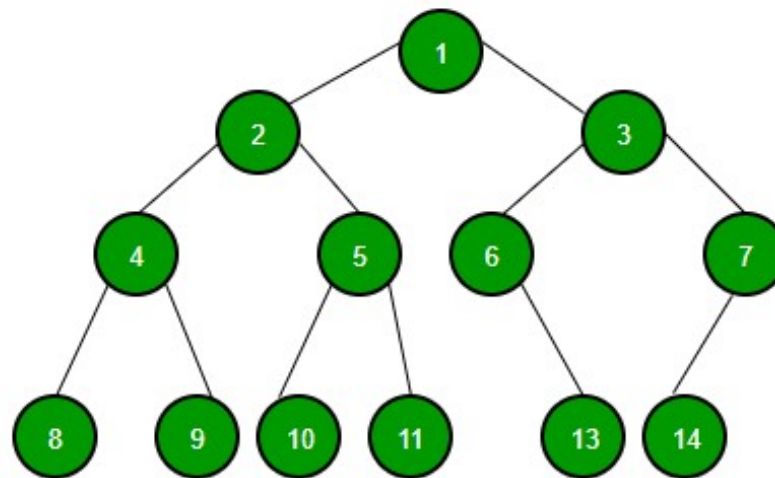
It is used to represent hierarchies.

It provides an efficient insertion and searching operations.

Trees are flexible. It allows to move subtrees around with minimum effort.

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.



Tree Traversals

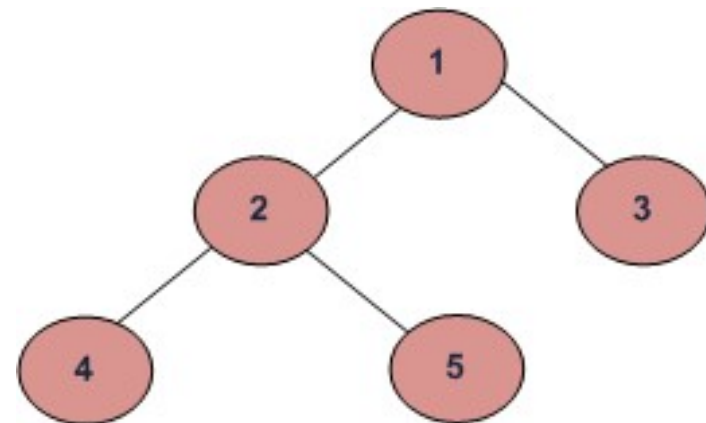
Depth First Traversals:

(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



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)

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)

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.

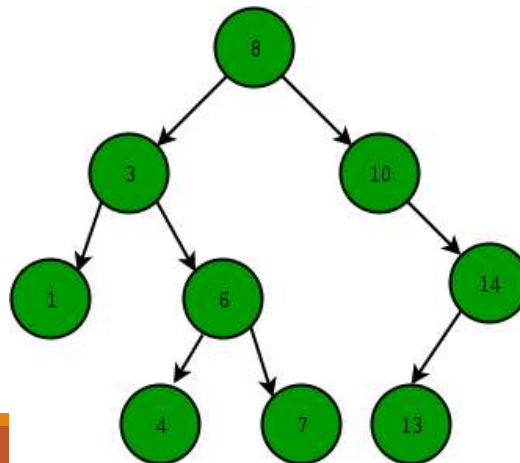
Binary Search Tree

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 lesser 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.



Insertion in a BST

```
Node* insert(struct Node* Node, int item)
{
    /* If the tree is empty, return a new Node */
    if (Node == NULL)
        return createnode(item);

    /* Otherwise, recur down the tree */
    if (item < Node->data)
        Node->left = insert(Node->left, item);
    else if (item > Node->data)
        Node->right = insert(Node->right, item);

    /* return the (unchanged) Node pointer */
    return Node;
}
```

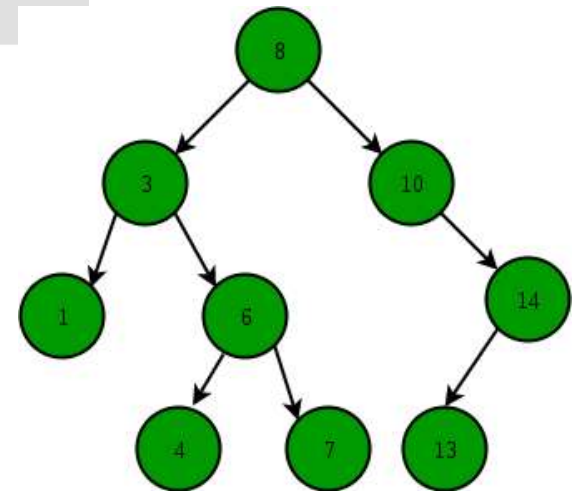
Inorder Traversal in a BST

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)

Prints all the elements in Ascending order

Inorder Traversal - 1, 3, 4, 6, 7, 8, 10, 13, 14



Self Read Topics

Deletion in a BST.

Evaluation of a expression Tree.

Construct Tree from given Inorder and Preorder traversals.

Construct a tree from Inorder and Level order traversals.

Diagonal traversal of a Tree.

Spiral traversal in a Tree.

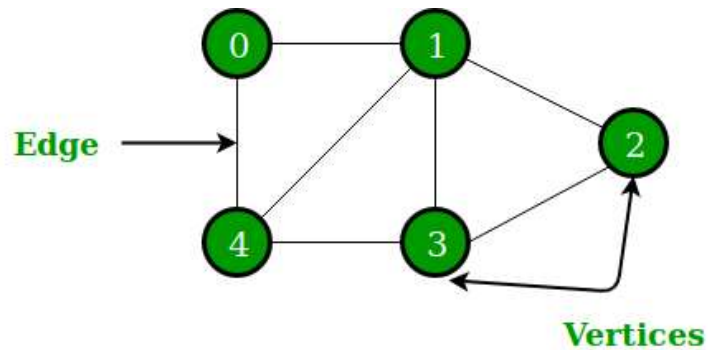
AVL Trees(Advance).

Rotation in AVL Trees(Advance).

Graphs

Graphs

A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph.



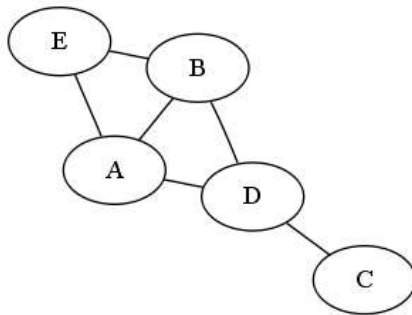
Real-Life uses

Graphs are used to solve many real-life problems. 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, locale etc.

Types of Graphs on basis of edge(dir)

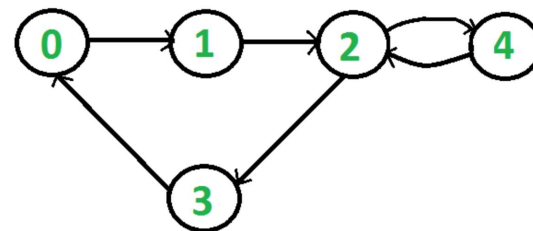
UNDIRECTED

Undirected graphs have edges that do not have a direction. The edges indicate a *two-way* relationship, in that each edge can be traversed in both directions.



DIRECTED

Directed graphs have edges with direction. The edges indicate a *one-way* relationship, in that each edge can only be traversed in a single direction.

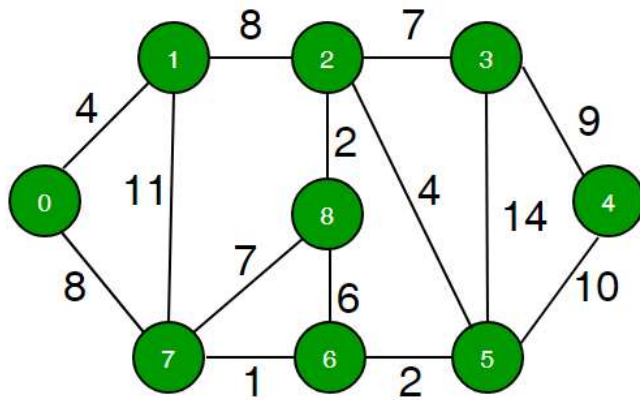


Types of Graphs on basis of weight

WEIGHTED

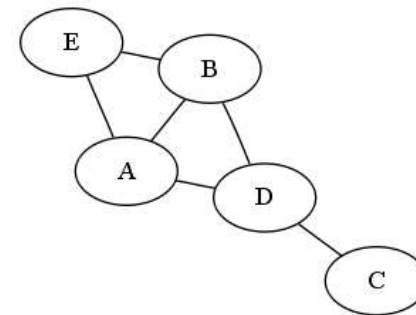
A **weighted graph** is a **graph** in which each branch is given a numerical **weight**.

A **weighted graph** is therefore a special type of labeled **graph** in which the labels are numbers



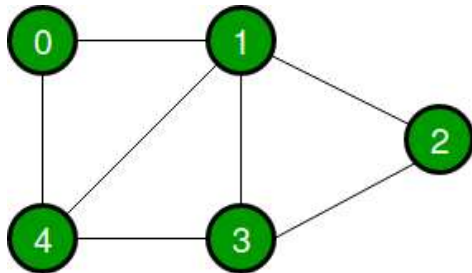
UNWEIGHTED

An **unweighted graph** is one in which an edge does not have any cost or weight associated with it



Adjacency Matrix Representation

Adjacency Matrix is a 2D array of size $V \times V$ where V is the number of vertices in a graph. Let the 2D array be $adj[][]$, 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 .



	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

BFS traversal in a Graph

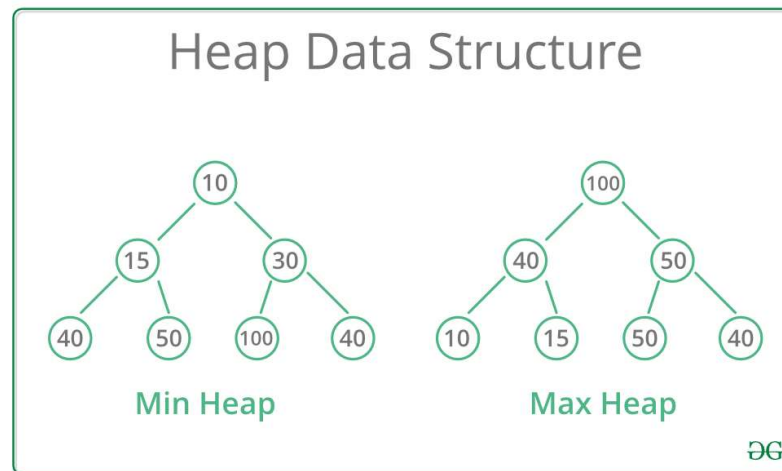
Breadth First Traversal (or Search) for a graph is similar to Breadth First Traversal of a tree (See method 2). The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array.

Heaps

Heaps

A Heap is a special Tree-based data structure in which the tree is a complete binary tree. Generally, Heaps can be of two types:

Min-Heap: In a Min-Heap the key present at the root node must be minimum among the keys present at all of its children. The same property must be recursively true for all sub-trees in that Binary Tree.



Max-Heap: In a Max-Heap the key present at the root node must be greatest among the keys present at all of its children. The same property must be recursively true for all sub-trees in that Binary Tree.

Applications of Heap Data Structure

- 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.

Insertion in a Heap

```
void heapify1(int arr[], int n, int i)
{
    int l = 2*i+1;
    int r = 2*i+2;

    int largest=i;

    if (l < n && arr[l] > arr[largest]) // If left child is larger than root
        largest = l;

    if (r < n && arr[r] > arr[largest]) // If right child is larger than largest so far
        largest = r;

    if (largest != i) { // If largest is not root
        swap(arr[i], arr[largest]);

        heapify1(arr, n, (i-1)/2); // Recursively heapify the affected sub-tree
        //(i-1)/2 for insertion;
        //largest for deletion;
    }
}
```

Deletion in a Heap

```
void heapify2(int arr[], int n, int i)
{
    int l = 2*i+1;
    int r = 2*i+2;

    int largest=i;

    if (l < n && arr[l] > arr[largest]) // If left child is larger than root
        largest = l;

    if (r < n && arr[r] > arr[largest]) // If right child is larger than largest so far
        largest = r;

    if (largest != i) { // If largest is not root
        swap(arr[i], arr[largest]);

        heapify2(arr, n, largest); // Recursively heapify the affected sub-tree
    }
}
```

Searching Algos

-ARRAY



Linear Search

```
int linearsearch(int arr[], int n, int x)
{
    int i;
    for (i = 0; i < n; i++)
        if (arr[i] == x)
            return i;
    return -1;
}
```

Binary Search

```
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l) {
        int mid = l + (r - l) / 2;

        if (arr[mid] == x)
            return mid;

        if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);

        return binarySearch(arr, mid + 1, r, x);
    }

    return -1;
}
```

Sorting Algos

Selection Sort

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning

```
void selectionSort(int arr[], int n)
{
    int i, j, min;

    for (i = 0; i < n-1; i++)
    {
        min = i;
        for (j = i+1; j < n; j++)
            if (arr[j] < arr[min])
                min = j;

        swap(arr[min], arr[i]);
    }
}
```

Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

```
void bubbleSort(int arr[], int n)
{
    int i, j;
    for (i = 0; i < n-1; i++)

        for (j = 0; j < n-i-1; j++)
            if (arr[j] > arr[j+1])
            {
                swap(arr[j], arr[j+1]);
                printArray(arr,15);
            }
}
```


Merge Sort

Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. **The merge() function** is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

```
MergeSort(arr[], l, r)
If r > l
    1. Find the middle point to divide the array into two halves:
        middle m = (l+r)/2
    2. Call mergeSort for first half:
        Call mergeSort(arr, l, m)
    3. Call mergeSort for second half:
        Call mergeSort(arr, m+1, r)
    4. Merge the two halves sorted in step 2 and 3:
        Call merge(arr, l, m, r)
```

QuickSort

QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x

```
/* low --> Starting index, high --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1); // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

```
/* 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 */
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low - 1) // Index of smaller element

    for (j = low; j <= high- 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++; // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

Algorithms

Algorithms Paradigms

1. Greedy Algorithms
2. Backtracking
3. Divide and Conquer
4. Dynamic Programming
5. Bit Algorithms
6. Branch and Bound

Greedy Algorithms

Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. So the problems where choosing locally optimal also leads to global solution are best fit for Greedy

Divide And Conquer Algorithms

Divide and Conquer is an algorithmic paradigm. A typical Divide and Conquer algorithm solves a problem using following three steps.

Divide: Break the given problem into subproblems of same type.

Conquer: Recursively solve these subproblems

Combine: Appropriately combine the answers

Merge Sort

Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. **The merge() function** is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

```
MergeSort(arr[], l, r)
If r > l
    1. Find the middle point to divide the array into two halves:
        middle m = (l+r)/2
    2. Call mergeSort for first half:
        Call mergeSort(arr, l, m)
    3. Call mergeSort for second half:
        Call mergeSort(arr, m+1, r)
    4. Merge the two halves sorted in step 2 and 3:
        Call merge(arr, l, m, r)
```

QuickSort

QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x

```
/* low --> Starting index, high --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1); // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

```
/* 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 */
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low - 1) // Index of smaller element

    for (j = low; j <= high- 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++; // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

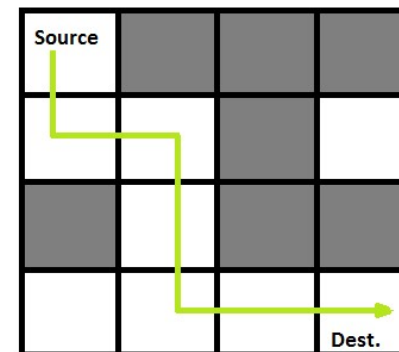
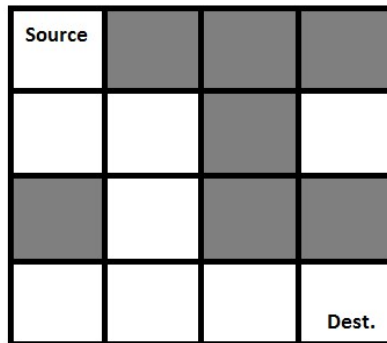

Backtracking Algorithms

Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time (by time, here, is referred to the time elapsed till reaching any level of the search tree).

Rat in a Maze

A Maze is given as $N \times N$ binary matrix of blocks where source block is the upper left most block i.e., `maze[0][0]` and destination block is lower rightmost block i.e., `maze[N-1][N-1]`. A rat starts from source and has to reach the destination. The rat can move only in two directions: forward and down.

In the maze matrix, 0 means the block is a dead end and 1 means the block can be used in the path from source to destination



Combinational Sum Problem

Given an array of positive integers **arr[]** and a sum **x**, find all unique combinations in arr[] where the sum is equal to x. The same repeated number may be chosen from arr[] unlimited number of times.

```
Input : arr[] = 2, 4, 6, 8
```

```
      x = 8
```

```
Output : [2, 2, 2, 2]
```

```
        [2, 2, 4]
```

```
        [2, 6]
```

```
        [4, 4]
```

```
        [8]
```

N-Queens Problem

The N Queen is the problem of placing N chess queens on an $N \times N$ chessboard so that no two queens attack each other.

