Texas College of Management & IT

Bachelor in Information Technology

Mid Term Evaluation

TIME: 1 HOURS 45 MINUTES (1 Hour 15 Minutes writing time 30 minutes Scan and Upload Time)

EXAM DATE: JUNE 27, 2021
SEMESTER: BIT III

SUBJECT: Data Structure & Algorithm

LECTURER: Saroj Ghimire

INSTRUCTIONS:

1. Section A is compulsory.
2. Please start every question on a new page.
3. Answers will not be marked if it is illegible.

Question Number	A	Total
1,2,3,4,5		/35

Section A

Attempt All the Questions **(Marks 5 x 7 = 35)**

1. Define minimum spanning tree. Explain the Kruskal algorithm with suitable example.(2+5)
2. Write recursive algorithm to get Fibonacci term. Why do we need recursion?(4+3)
3. Define stack as an ADT. Explain the condition that is to be checked for Push operation.(Stack)(3+4)
4. How balance factor is calculated in AVL tree. Construct an AVL Tree by inserting numbers from 1 to 8. (2+5)
5. Write algorithm for infix to prefix conversion with suitable example. (7)

Best of Luck

DATA STRUCTURE AND ALGORITHM

LECTURER: ER. SAROJ GHIMIRE
QUALIFICATION : MSC.CSIT, COMPUTER ENGINEERING

Lincoln University College

Overview

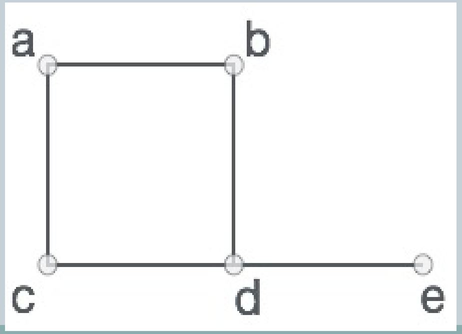
- Concept and representation of graphs, graphs traversal, minimum spanning tree: kruskal algorithm
- Shortest path algorithm: dijksrtra algorithm
- Definition of tree, tree height, level and depth; basic operation in binary tree
- Tree traversals, binary search tree, AVL tree, application of tree

Objectives

- To learn what a graph is and how it is used.
- To implement the graph abstract data type using multiple internal representations.
- To see how graphs can be used to solve a wide variety of problems
- Define trees as data structures
- Define the terms associated with trees
- Discuss tree traversal algorithms Discuss

Concept and representation of graphs

- **Concept of graphs**
  - A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links.
  - Graph is a non linear data structure
  - A graph is a data structure that consists of the following two components:
    - A finite set of vertices also called as nodes.
    - A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not the same as (v, u) in case of a directed graph(di-graph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost
    - In this graph, V = {a, b, c, d, e} E = {ab, ac, bd, cd, de}

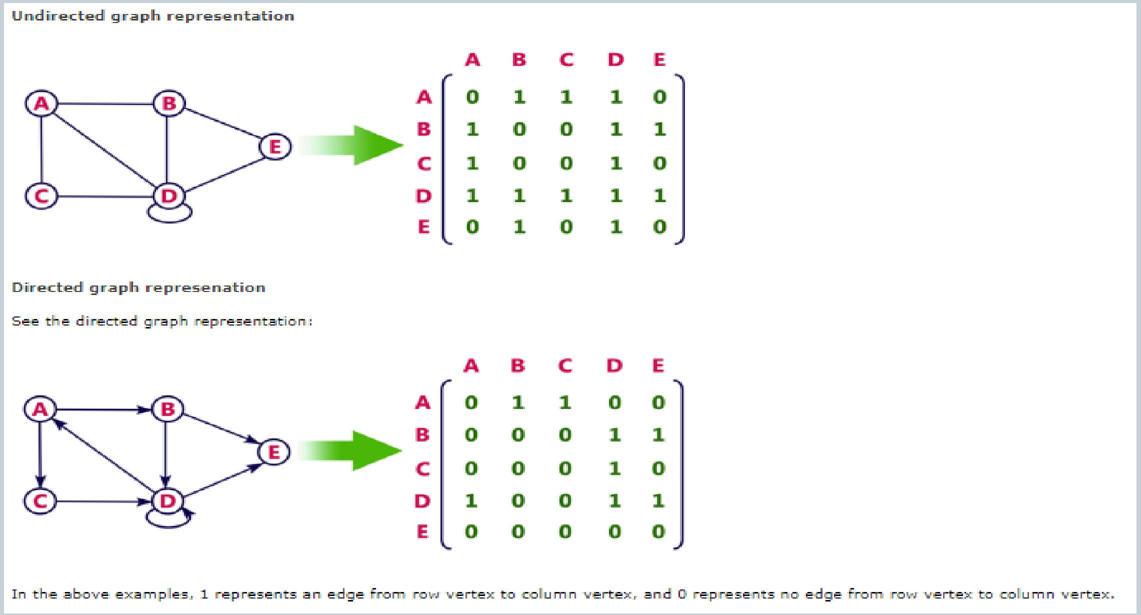


Concept and representation of graphs(contd)

- Representation of graphs
  - In graph theory, a graph representation is a technique to store graph into the memory of computer.
  - To represent a graph, we just need the set of vertices, and for each vertex the neighbors of the vertex (vertices which is directly connected to it by an edge). If it is a weighted graph, then the weight will be associated with each edge.
  - There are different ways to optimally represent a graph, depending on the density of its edges, type of operations to be performed and ease of use.

Concept and representation of graphs(contd)

1. Adjacency Matrix
   - Adjacency matrix is a sequential representation.
   - It is used to represent which nodes are adjacent to each other. i.e. is there any edge connecting nodes to a graph.
   - In this representation, we have to construct a n X n matrix A. If there is any edge from a vertex i to vertex j, then the corresponding element of A, $a^{i,j} = 1$, otherwise $a^{i,j} = 0$
   - If there is any weighted graph then instead of 1s and 0s, we can store the weight of the edge.



Undirected graph representation

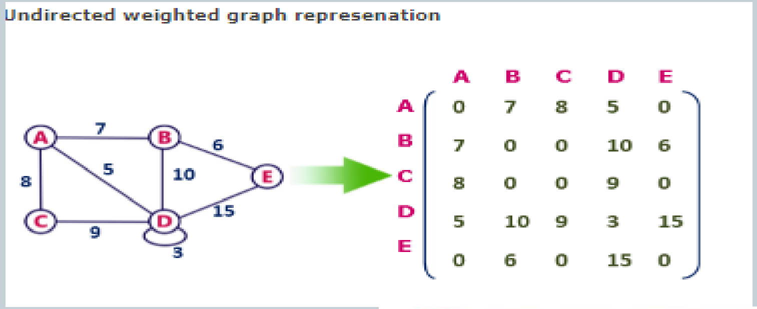
$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 1 & 0 \\ B & 1 & 0 & 0 & 1 & 1 \\ C & 1 & 0 & 0 & 1 & 0 \\ D & 1 & 1 & 1 & 1 & 1 \\ E & 0 & 1 & 0 & 1 & 0 \end{array}$$

Directed graph represenation

See the directed graph representation:

$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 0 & 0 \\ B & 0 & 0 & 0 & 1 & 1 \\ C & 0 & 0 & 0 & 1 & 0 \\ D & 1 & 0 & 0 & 1 & 1 \\ E & 0 & 0 & 0 & 0 & 0 \end{array}$$

In the above examples, 1 represents an edge from row vertex to column vertex, and 0 represents no edge from row vertex to column vertex.



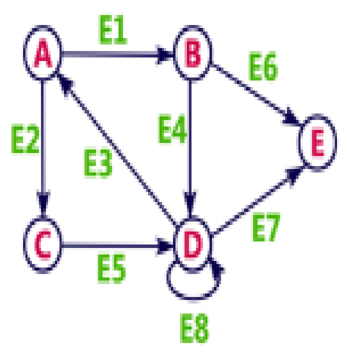
Undirected weighted graph represenation

$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 7 & 8 & 5 & 0 \\ B & 7 & 0 & 0 & 10 & 6 \\ C & 8 & 0 & 0 & 9 & 0 \\ D & 5 & 10 & 9 & 3 & 15 \\ E & 0 & 6 & 0 & 15 & 0 \end{array}$$

Concept and representation of graphs(contd)

2. Incidence matrix
   - In Incidence matrix representation, graph can be represented using a matrix of size:
   - Total number of vertices by total number of edges.
   - It means if a graph has 4 vertices and 6 edges, then it can be represented using a matrix of 4X6 class. In this matrix, columns represent edges and rows represent vertices.
   - This matrix is filled with either 0 or 1 or -1. Where,
     - 0 is used to represent row edge which is not connected to column vertex.
     - 1 is used to represent row edge which is connected as outgoing edge to column vertex.
     - -1 is used to represent row edge which is connected as incoming edge to column vertex

Consider the following directed graph representation.

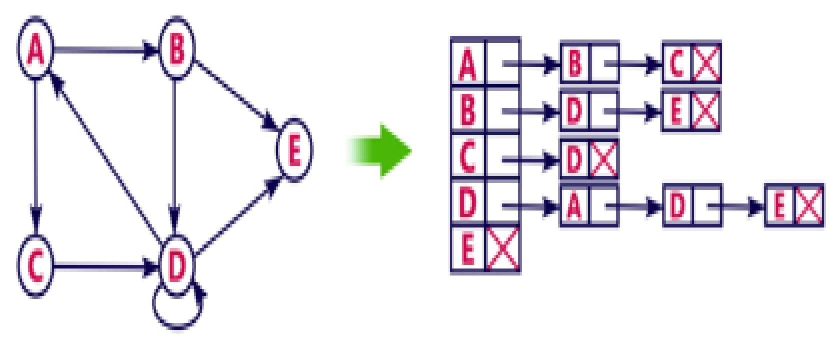


$$\begin{array}{c|cccccccc} & E1 & E2 & E3 & E4 & E5 & E6 & E7 & E8 \\ \hline A & 1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ B & -1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ C & 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ D & 0 & 0 & 1 & -1 & -1 & 0 & 1 & 1 \\ E & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 \end{array}$$

Concept and representation of graphs(contd)

3. Adjacency List
   - Adjacency list is a linked representation.
   - In this representation, for each vertex in the graph, we maintain the list of its neighbors. It means, every vertex of the graph contains list of its adjacent vertices.
   - We have an array of vertices which is indexed by the vertex number and for each vertex v, the corresponding array element points to a singly linked list of neighbors of v.

Let's see the following directed graph representation implemented using linked list:



Graphs Traversal

- **Graph traversal** is a technique used to search for a vertex in a graph. It is also used to decide the order of vertices to be visited in the search process.
- A graph traversal finds the edges to be used in the search process without creating loops. This means that, with graph traversal, we can visit all the vertices of the graph without getting into a looping path. There are two graph traversal techniques:

1. **DFS (Depth First Search)**

   DFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS traversal.

   We use the following steps to implement DFS traversal...

   1. Define a Stack of size total number of vertices in the graph.
   2. Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.
   3. Visit any one of the non-visited **adjacent** vertices of a vertex which is at the top of stack and push it on to the stack.
   4. Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.
   5. When there is no new vertex to visit then use **back tracking** and pop one vertex from the stack.
   6. Repeat steps 3, 4 and 5 until stack becomes Empty.
   7. When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

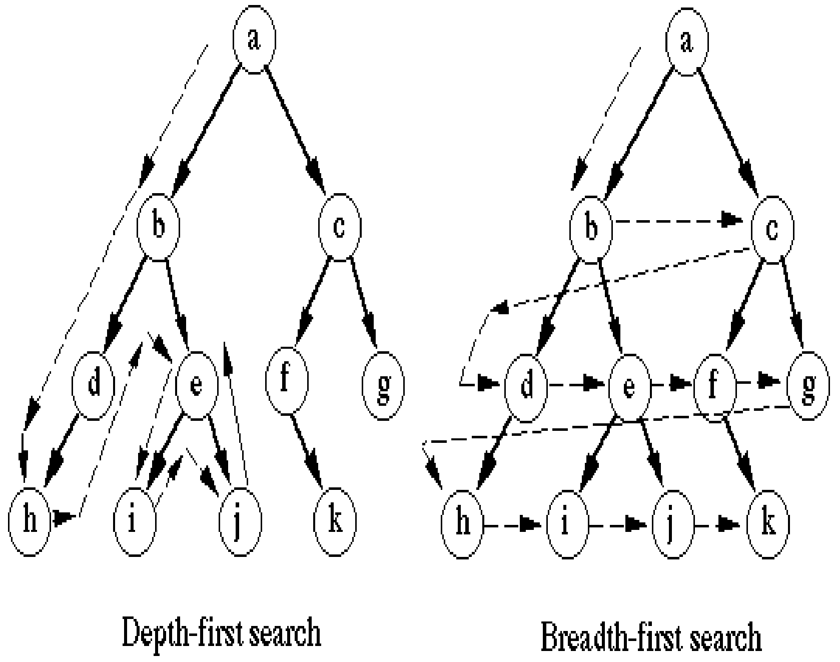
Graphs Traversal(contd)

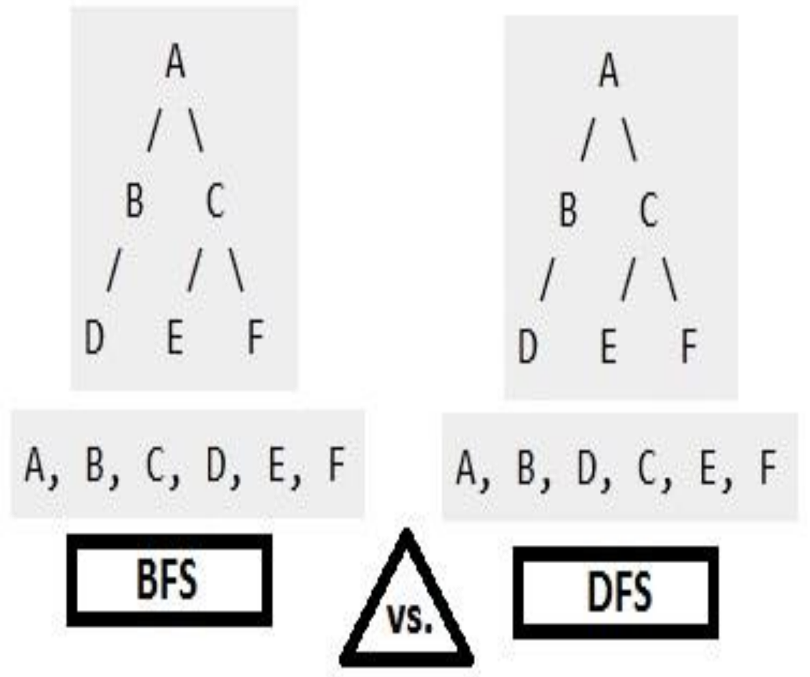
2. **BFS (Breath First Search)**

   BFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal

   1. Define a Queue of size total number of vertices in the graph.
   2. Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.
   3. Visit all the non-visited **adjacent** vertices of the vertex which is at front of the Queue and insert them into the Queue.
   4. When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.
   5. Repeat steps 3 and 4 until queue becomes empty.
   6. When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

Example of **DFS VS BFS** on next page


a
b
c
d
e
f
g
h
i
j
k
Depth-first search
Breadth-first search


A
B C
D E F
A, B, C, D, E, F
BFS
vs.
A, B, D, C, E, F
DFS

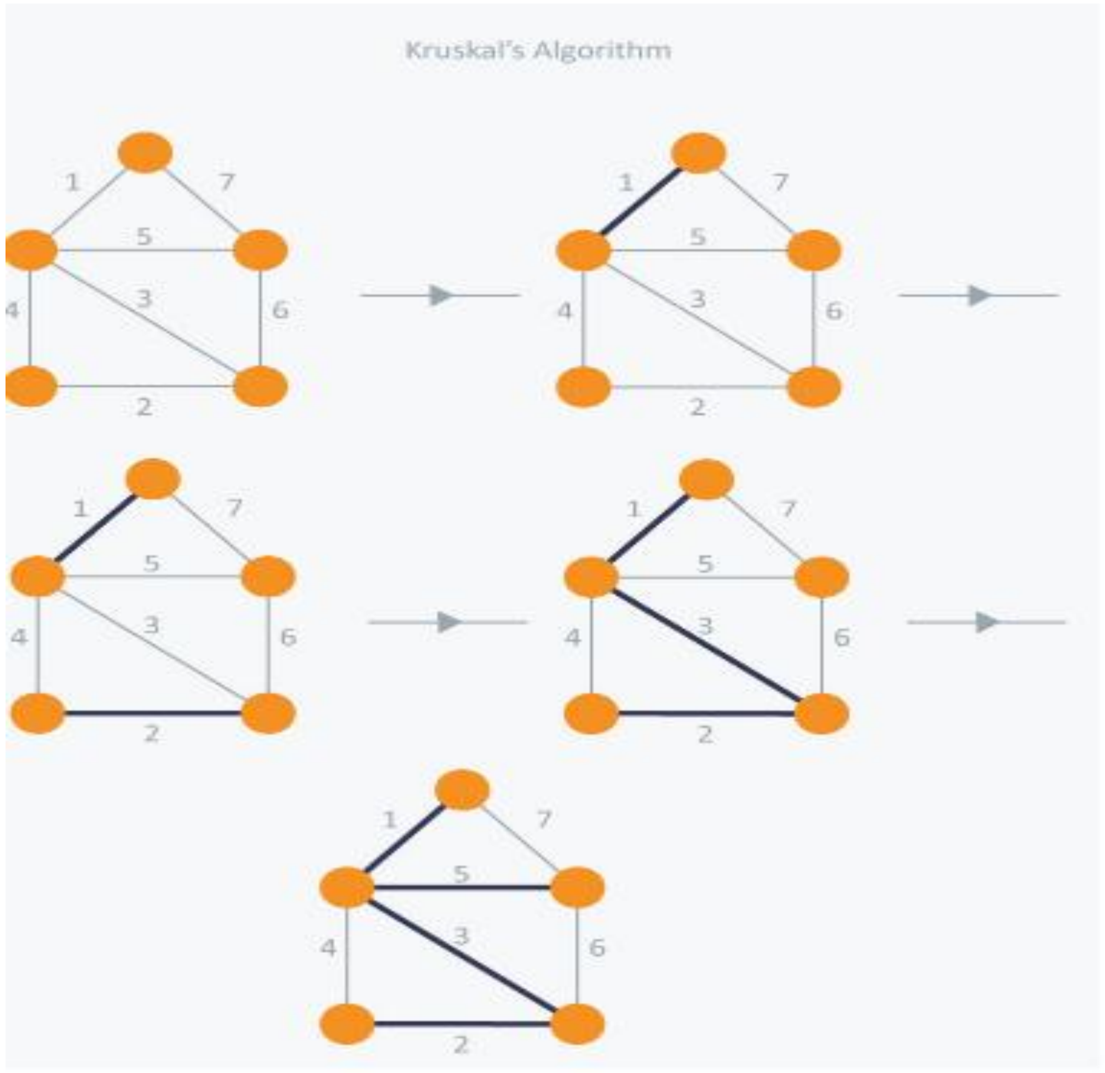
Minimum Spanning Tree: Kruskal Algorithm

- A minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. Application in the design of networks, Handwriting recognition, Image segmentation
  - **Kruskal's Algorithm**

    Kruskal's Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree. Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has least weight and add it to the growing spanning tree.

    Algorithm Steps:

    i. Sort the graph edges with respect to their weights.
    ii. Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
    iii. Only add edges which doesn't form a cycle , edges which connect only disconnected components.


Kruskal's Algorithm
1
7
5
3
4
6
2

Shortest Path Algorithm: Dijksrtra Algorithm

- The shortest path problem is about finding a path between 2 vertices in a graph such that the total sum of the edges weights is minimum.

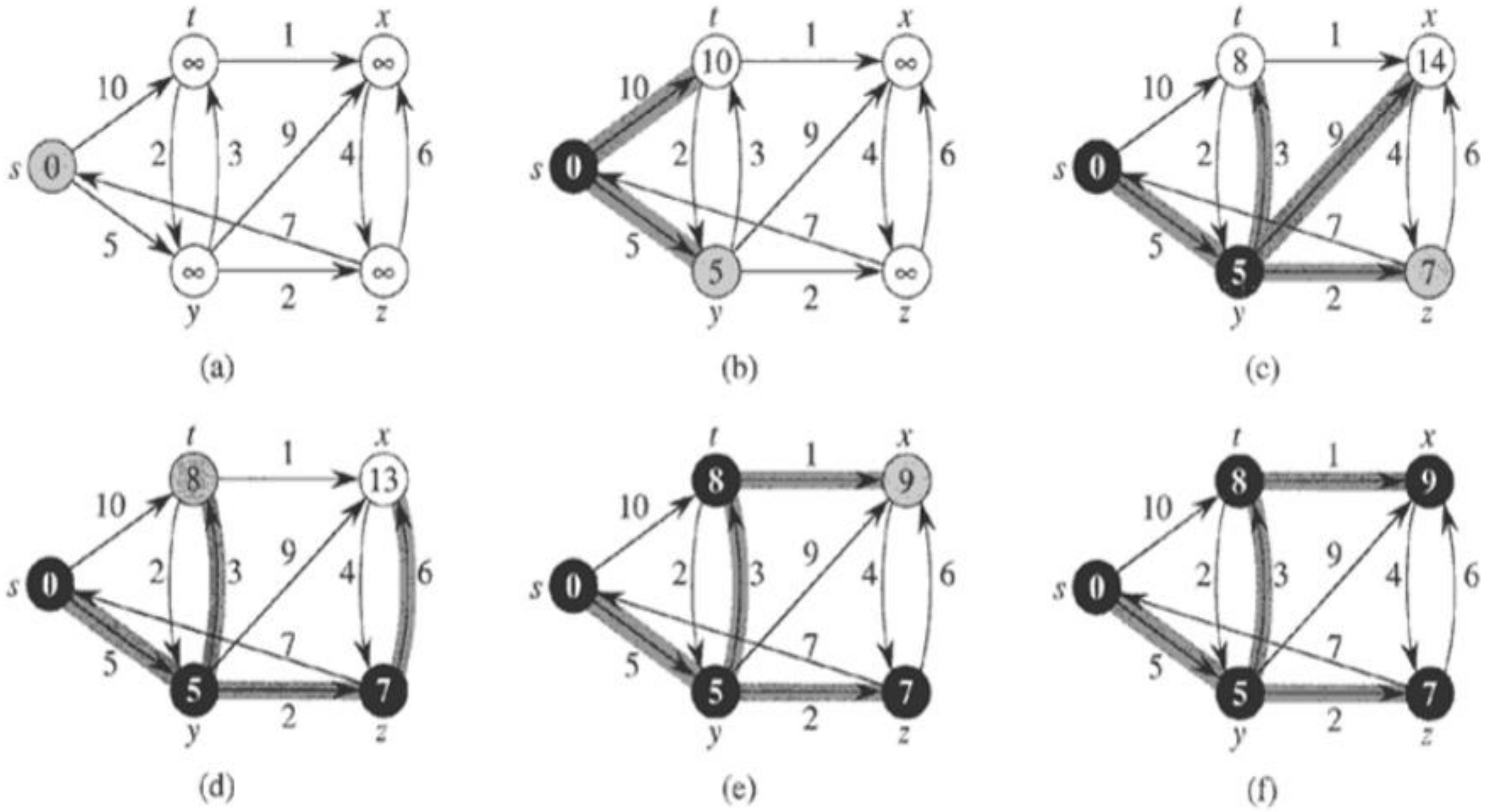
Dijksrtra Algorithm:

It is a greedy algorithm that solves the single-source shortest path problem for a directed graph G = (V, E) with nonnegative edge weights, i.e., w (u, v) ≥ 0 for each edge (u, v) ∈ E.

Dijkstra's Algorithm maintains a set S of vertices whose final shortest - path weights from the source s have already been determined. That's for all vertices v ∈ S; we have d [v] = δ (s, v). T

1. set it to zero for our initial node and to infinity for all other nodes. Set the initial node as current
2. find minimum weight/distance from source to adjacent node that is unvisited repeat process until all node are visited .
3. Finally display the shortest path from source to destination.

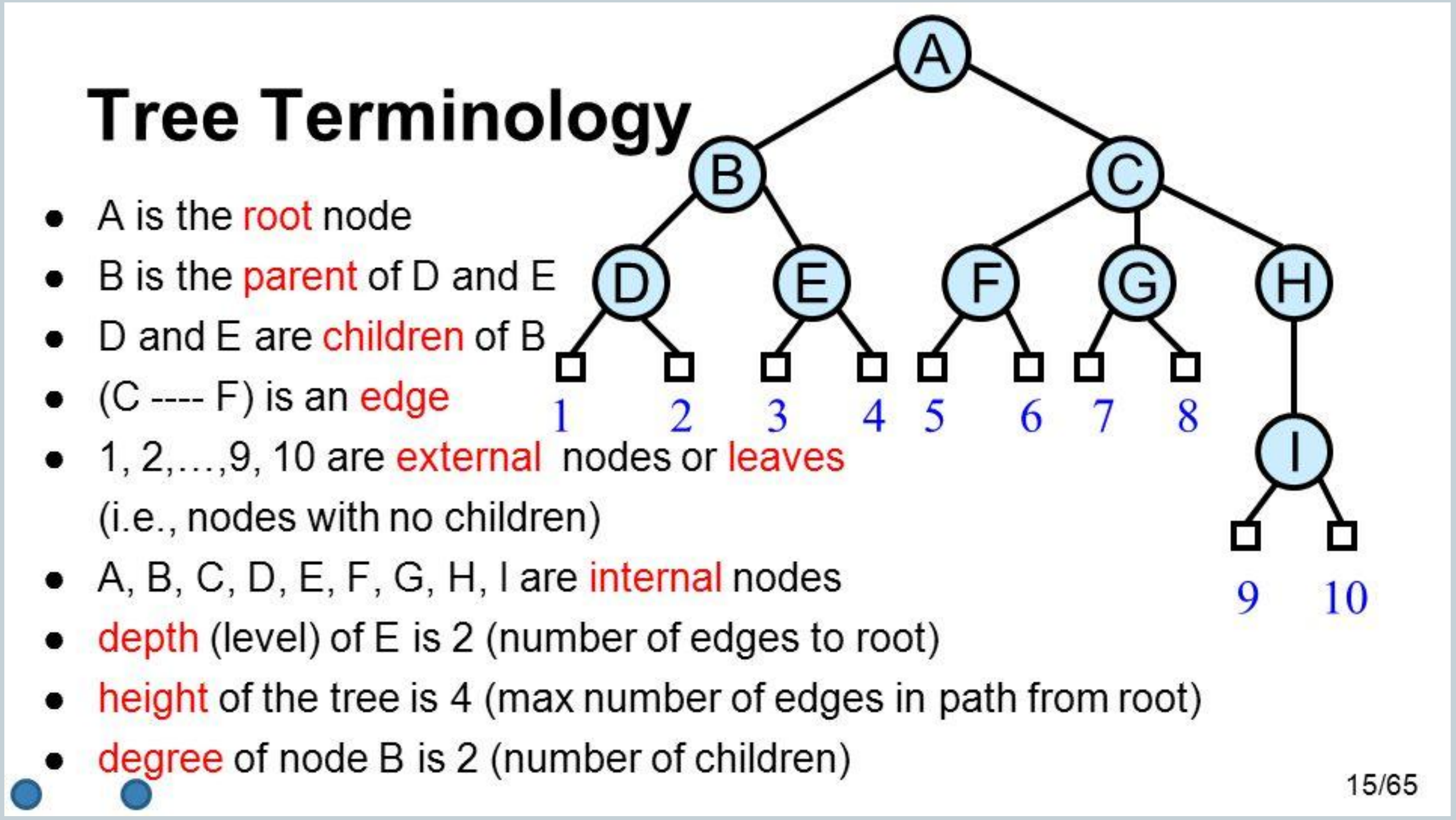
The Dijkstra's Algorithm - An Example



Definition Of Tree, Tree Height, Level And Depth

- A tree is a connected graph without any circuits **OR** If in a graph, there is one and only one path between every pair of vertices, then graph is called as a tree.
- Terminology of tree
  - ✓ **Root:** In a tree data structure, the first node is called as **Root Node**
  - ✓ **Edge: In** a tree data structure, the connecting link between any two nodes is called as **EDGE**. In a tree with '**N**' number of nodes there will be a maximum of '**N-1**' number of edges.
  - ✓ **Parent:** In a tree data structure, the node which is a predecessor of any node is called as **PARENT NODE**.
  - ✓ **Child:** In a tree data structure, the node which is descendant of any node is called as **CHILD Node**
  - ✓ **Siblings:** In a tree data structure, nodes which belong to same Parent are called as **SIBLINGS**
  - ✓ **Leaf:** In a tree data structure, the node which does not have a child is called as **LEAF Node.** In a tree, leaf node is also called as '**Terminal**' node.
  - ✓ **Internal Nodes :**In a tree data structure, nodes other than leaf nodes are called as **Internal Nodes**. **The root node is also said to be Internal Node** if the tree has more than one node. Internal nodes are also called as '**Non-Terminal**' nodes.
  - ✓ **Degree:** In a tree data structure, the total number of children of a node is called as **DEGREE** of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as '**Degree of Tree.**
  - ✓ **Level:** In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on..
  - ✓ **Height:** In a tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as **HEIGHT** of that Node. In a tree, height of the root node is said to be **height of the tree**.
  - ✓ **Depth:** In a tree data structure, the total number of edges from root node to a particular node is called as **DEPTH** of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be **Depth of the tree**

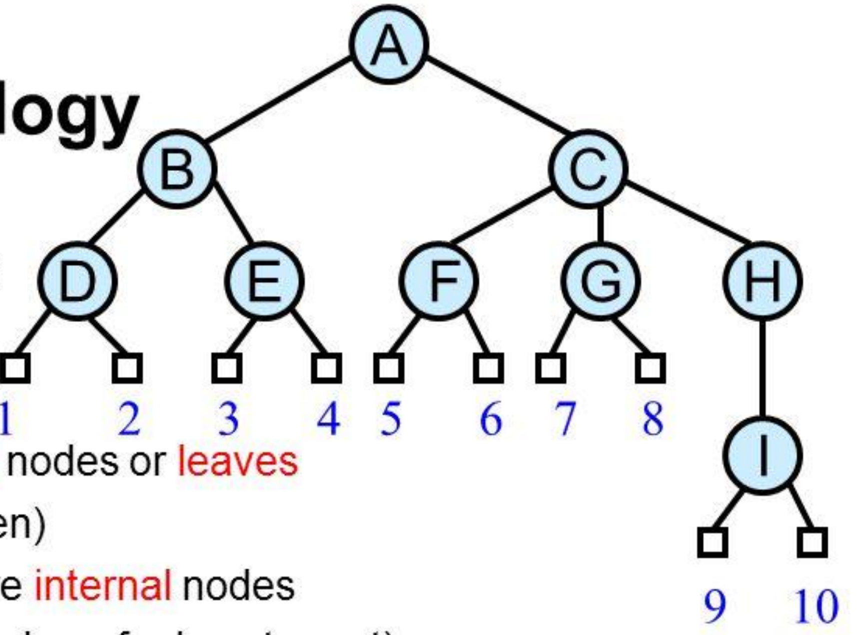
Definition Of Tree, Tree Height, Level And Depth



Tree Terminology

- A is the root node
- B is the parent of D and E
- D and E are children of B
- (C ---- F) is an edge
- 1, 2,…,9, 10 are external nodes or leaves (i.e., nodes with no children)
- A, B, C, D, E, F, G, H, I are internal nodes
- depth (level) of E is 2 (number of edges to root)
- height of the tree is 4 (max number of edges in path from root)
- degree of node B is 2 (number of children)

15/65



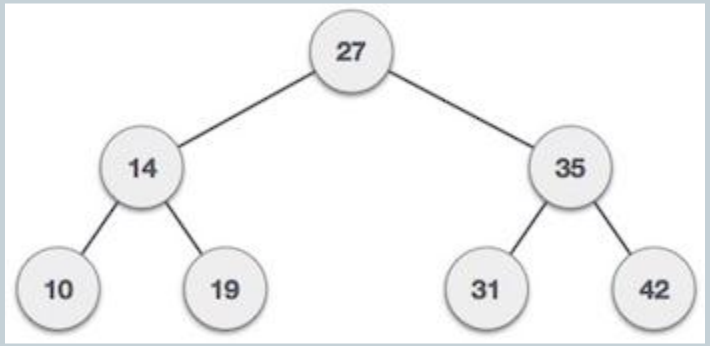
Texas College of Management & IT

LINCOLN UNIVERSITY COLLEGE DKU016 (B)

BASIC OPERATION IN BINARY TREE

- Binary Search tree exhibits a special behavior. A node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent value

```
Tree node :
struct node {
        int data;
        struct node *leftChild;
        struct node *rightChild;
};
```



- A binary tree is a tree with the following properties:
  - Each internal node has at most two children
  - The children of a node are an ordered pair (left and right)
- The basic operations that can be performed on a binary search tree data structure, are **Insert, Search Preorder Traversal, In order Traversal and Post order Traversal** –

Basic Operation In Binary Tree

- **Insert:** The very first insertion creates the tree. Afterwards, whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left sub tree and insert the data. Otherwise, search for the empty location in the right sub tree and insert the data

```
Algorithm
If root is NULL
  then create root node
return
If root exists then
  compare the data with node.data
  while until insertion position is located
    If data is greater than node.data
      goto right subtree
    else
      goto left subtree
  endwhile
  insert data
end If
```

- **Search:** Whenever an element is to be searched, start searching from the root node, then if the data is less than the key value, search for the element in the left sub tree. Otherwise, search for the element in the right sub tree. Follow the same algorithm for each node.

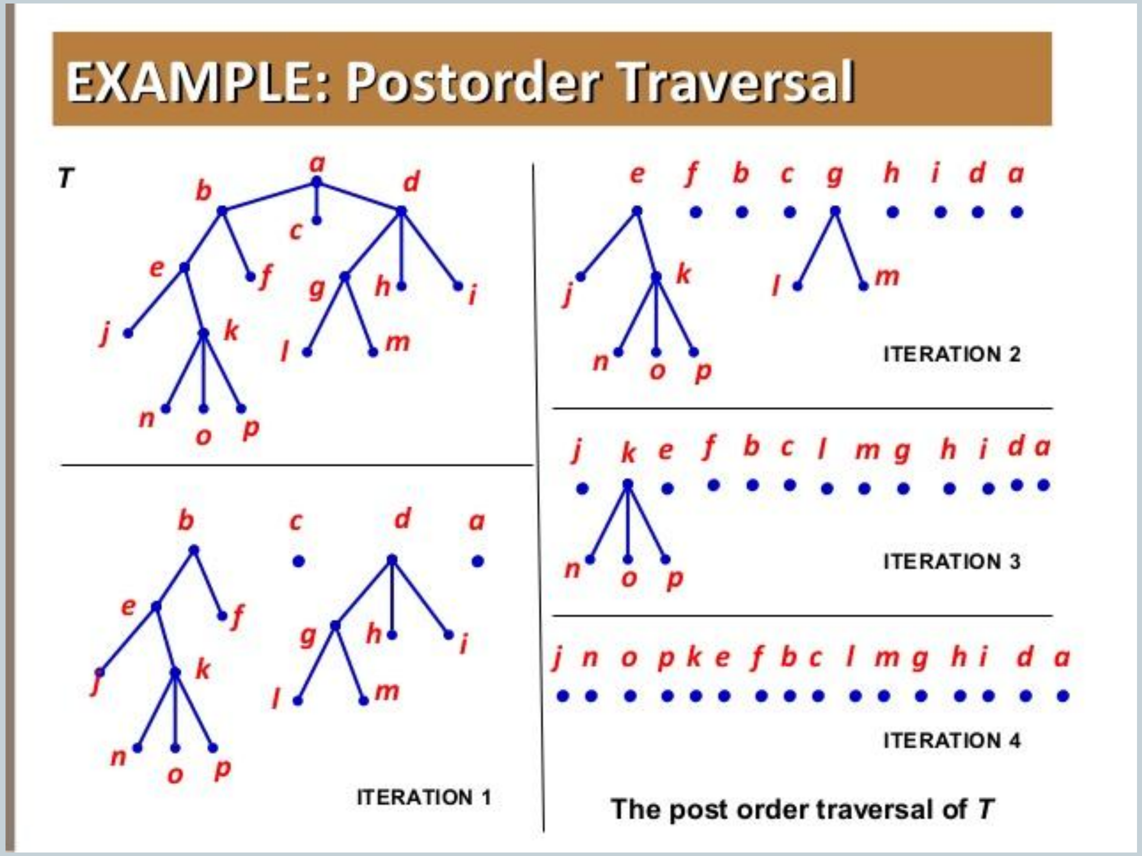
```
Algorithm
If root.data is equal to search.data
  return root
else
  while data not found
    If data is greater than node.data
      goto right subtree
    else
      goto left subtree
    If data found
      return node
  endwhile
  return data not found
end if
```

Tree Traversal

- A tree traversal is a method of visiting every node in the tree. By visit, we mean that some type of operation is performed. For example, you may wish to print the contents of the nodes.

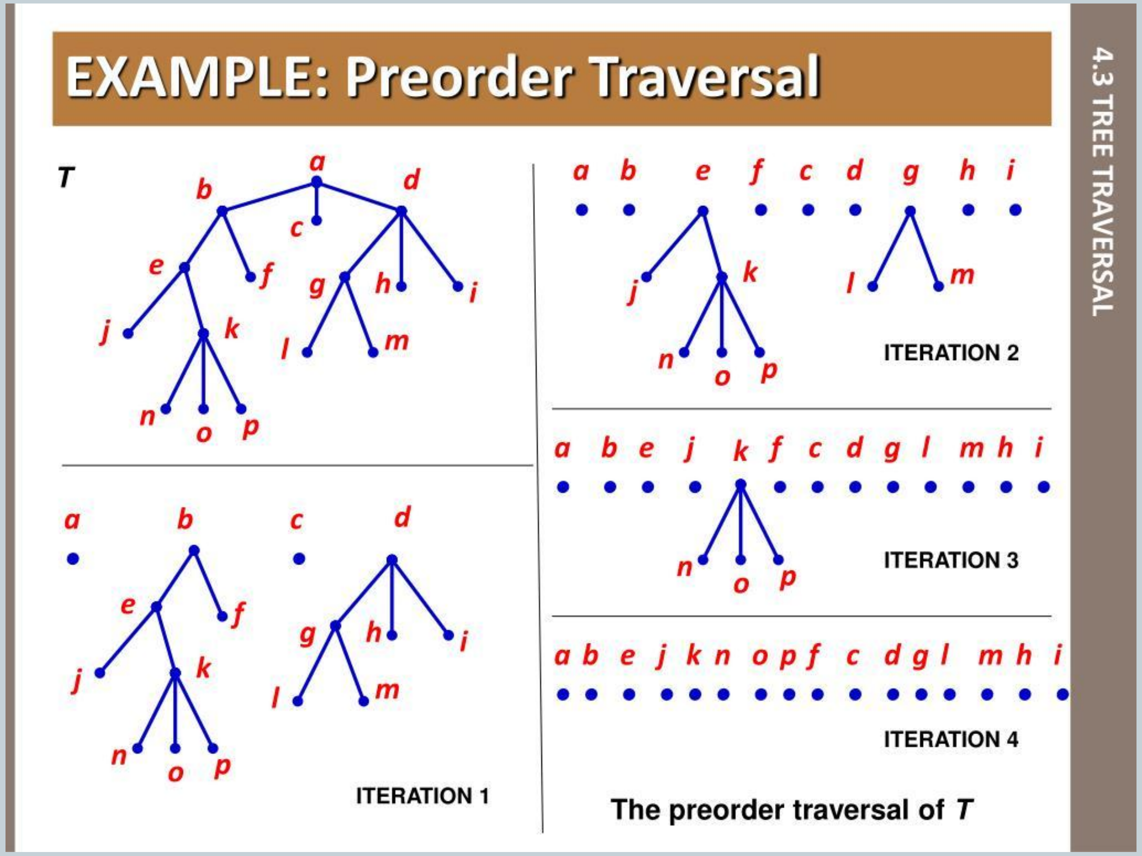
1. **In order Traversal: In the case of in order traversal, the root of each sub tree is visited after its left sub tree has been traversed but before the traversal of its right sub tree begins. The steps for traversing a binary tree in in order traversal are:**
   - **Visit the left sub tree, using in order.**
   - **Visit the root.**
   - **Visit the right sub tree, using in order**
2. **Preorder Traversal: In a preorder traversal, each root node is visited before its left and right sub trees are traversed. Preorder search is also called backtracking. The steps for traversing a binary tree in preorder traversal are:**
   - **Visit the root.**
   - **Visit the left sub tree, using preorder.**
   - **Visit the right sub tree, using preorder**
3. **Post order Traversal: In a post order traversal, each root is visited after its left and right sub trees have been traversed. The steps for traversing a binary tree in post order traversal are:**
   - **Visit the left sub tree, using post order.**
   - **Visit the right sub tree, using post order**
   - **Visit the root**

Tree Traversal



EXAMPLE: Postorder Traversal

The post order traversal of *T*



EXAMPLE: Preorder Traversal

The preorder traversal of *T*

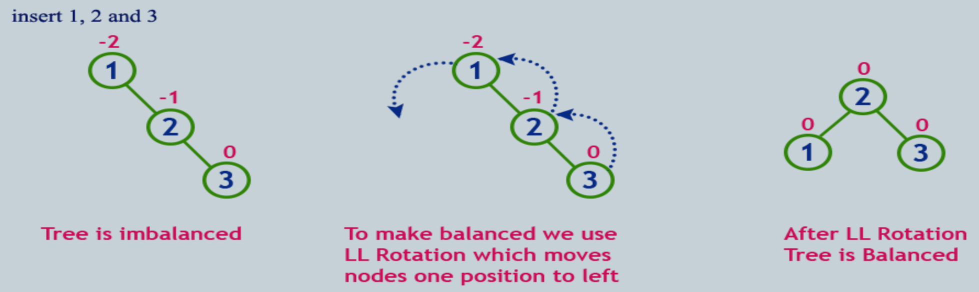
AVL Tree

- **AVL** tree is a height-balanced binary search tree. That means, an **AVL** tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced if, the difference between the heights of left and right sub trees of every node in the tree is either -1, 0 or +1.
- In other words, a binary tree is said to be balanced if the height of left and right children of every node differ by either -1, 0 or +1. In an AVL tree, every node maintains extra information known as balance factor.
- In **AVL** tree, after performing operations like insertion and deletion we need to check the **balance factor** of every node in the tree.
- If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced. Whenever the tree becomes imbalanced due to any operation we use **rotation** operations to make the tree balanced.
- Rotation operations are used to make the tree balanced

AVL Tree

- **Single Left Rotation (LL Rotation)**

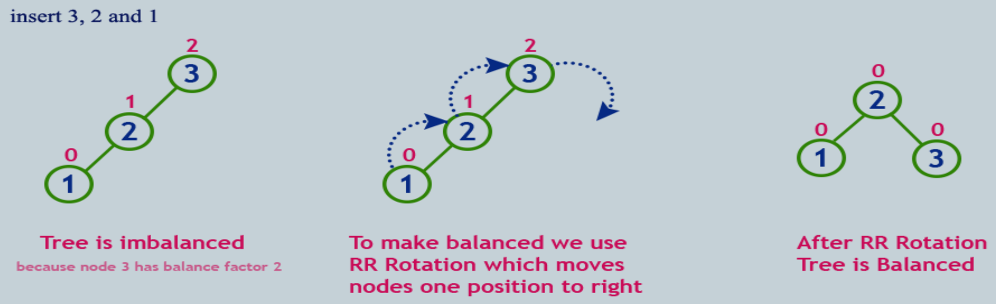
  In LL Rotation, every node moves one position to left from the current position. To understand LL Rotation, let us consider the following insertion operation in AVL Tree.



insert 1, 2 and 3
-2
1
-1
2
0
3
Tree is imbalanced
To make balanced we use LL Rotation which moves nodes one position to left
After LL Rotation Tree is Balanced

- **Single Right Rotation (RR Rotation)**

  In RR Rotation, every node moves one position to right from the current position. To understand RR Rotation, let us consider the following insertion operation in AVL Tree..

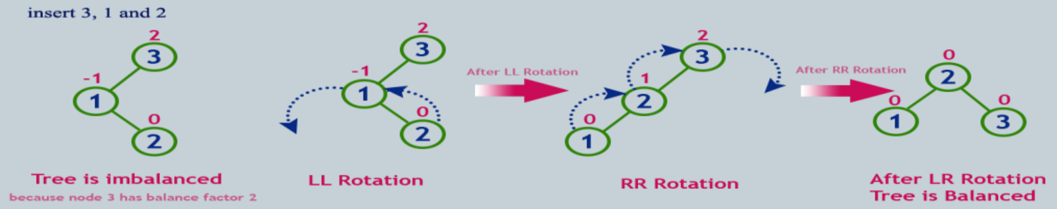


insert 3, 2 and 1
2
3
1
2
0
1
Tree is imbalanced
because node 3 has balance factor 2
To make balanced we use RR Rotation which moves nodes one position to right
After RR Rotation Tree is Balanced

AVL Tree

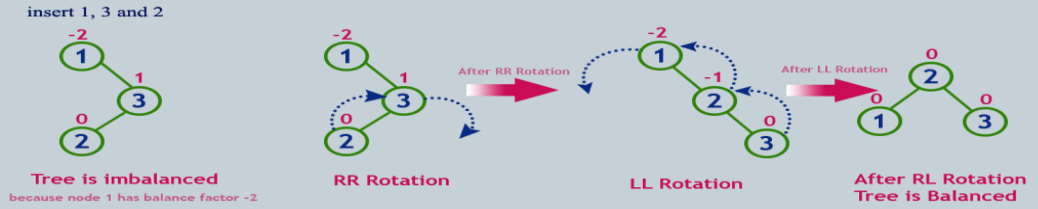
- **Left Right Rotation (LR Rotation)**

  The LR Rotation is a sequence of single left rotation followed by a single right rotation. In LR Rotation, at first, every node moves one position to the left and one position to right from the current position. To understand LR Rotation, let us consider the following insertion operation in AVL Tree..



- **Right Left Rotation (RL Rotation)**

  The RL Rotation is sequence of single right rotation followed by single left rotation. In RL Rotation, at first every node moves one position to right and one position to left from the current position. To understand RL Rotation, let us consider the following insertion operation in AVL Tree..



AVL Tree

- **Insertion Operation in AVL Tree**
  - ✓ In an AVL tree, the insertion operation is performed with **O(log n)** time complexity. In AVL Tree, a new node is always inserted as a leaf node. The insertion operation is performed as follows...

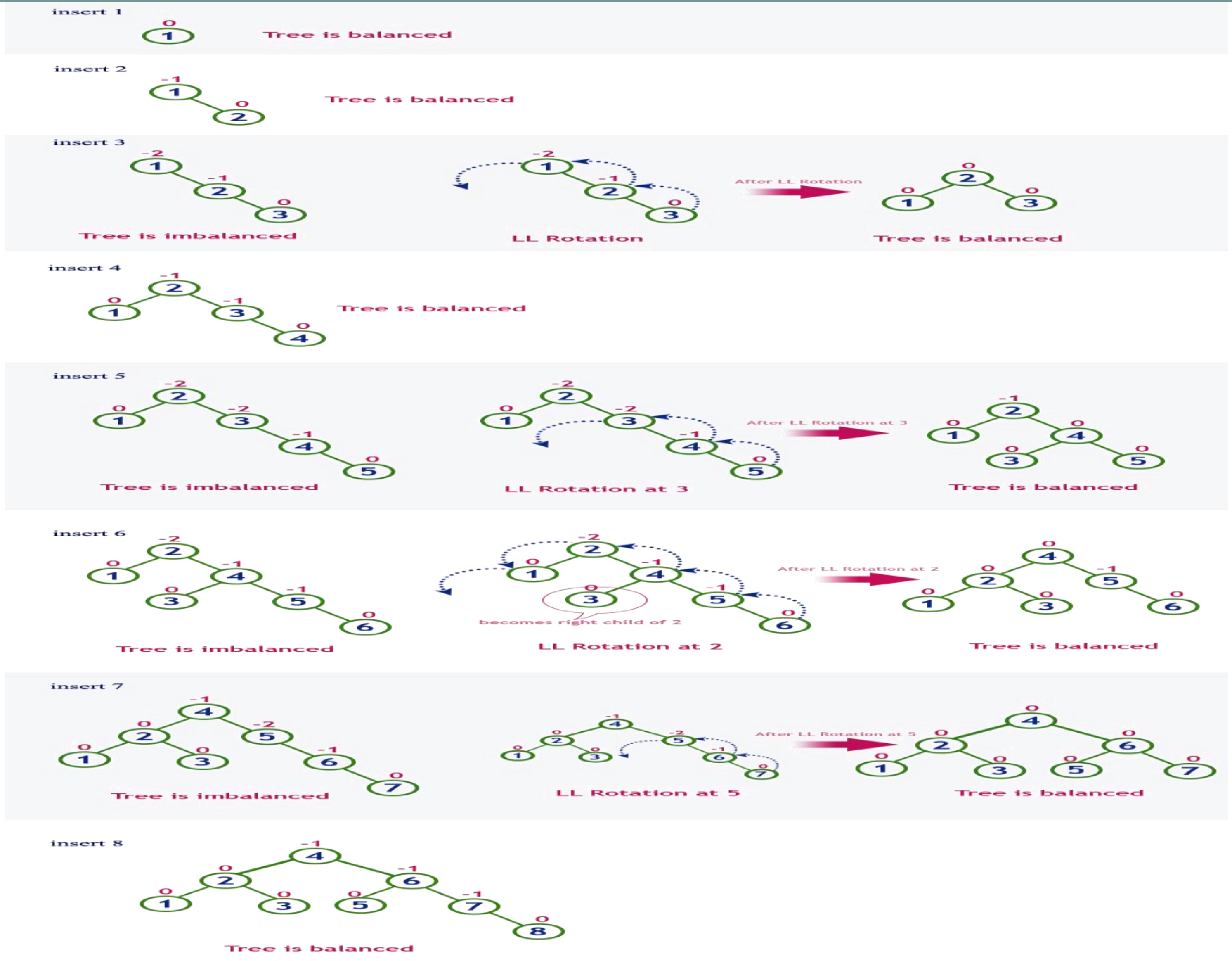
    i. Insert the new element into the tree using Binary Search Tree insertion logic.

    ii. After insertion, check the Balance Factor of every node.

    iii. If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.

    iv. If the Balance Factor of any node is other than 0 or 1 or -1 then that tree is said to be imbalanced. In this case, perform suitable Rotation to make it balanced and go for next operation.

Example: Construct an AVL Tree by inserting numbers from 1 to 8. On next page


insert 1
Tree is balanced
insert 2
Tree is balanced
insert 3
Tree is imbalanced
LL Rotation
After LL Rotation
Tree is balanced
insert 4
Tree is balanced
insert 5
Tree is imbalanced
LL Rotation at 3
After LL Rotation at 3
Tree is balanced
insert 6
Tree is imbalanced
becomes right child of 2
LL Rotation at 2
After LL Rotation at 2
Tree is balanced
insert 7
Tree is imbalanced
LL Rotation at 5
After LL Rotation at 5
Tree is balanced
insert 8
Tree is balanced

Application Of Tree

- Represent organization
- Represent computer file systems
- Networks to find best path in the Internet
- Chemical formulas representation
- Manipulate hierarchical data.
- Make information easy to search (see tree traversal).
- Manipulate sorted lists of data.
- As a workflow for compositing digital images for visual effects.
- Router algorithms

Further Readings

- “Introduction to Algorithms” by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
- “Algorithms Unlocked” by Thomas H. Cormen

DATA STRUCTURE AND ALGORITHM

LECTURER: ER. SAROJ GHIMIRE
QUALIFICATION: MSC.CSIT, BE(COMPUTER)

Lincoln University College

Objectives

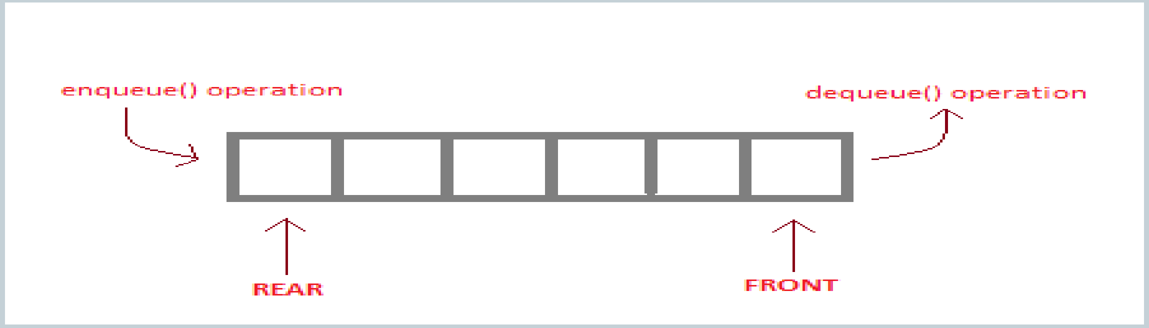
- Examine queue processing
- Define a queue abstract data type
- Demonstrate how a queue can be used to solve problems
- Examine various queue implementations

Overview

- Basic Concept of Queue, Queue as ADT, Basic Operation
- Linear Queue, Circular Queue, Priority Queue
- Application of Queue

Concept Of Queue

- Queue is also an abstract data type or a linear data structure in which the first element is inserted from one end called the REAR(also called tail), and the removal of existing element takes place from the other end called as FRONT(also called head
- This makes queue as **FIFO**(First in First Out) data structure, which means that element inserted first will be removed first
- Which is exactly how queue system works in real world. If you go to a ticket counter to buy movie tickets, and are first in the queue, then you will be the first one to get the tickets. Right? Same is the case with Queue data structure. Data inserted first, will leave the queue first
- The process to add an element into queue is called **Enqueue** and the process of removal of an element from queue is called **Dequeue**



Queue as ADT

- Queue() creates a new queue that is empty. It needs no parameters and returns an empty queue.
- Enqueue(item) adds a new item to the rear of the queue. It needs the item and returns nothing.
- Dequeue() removes the front item from the queue. It needs no parameters and returns the item. The queue is modified.
- Is Empty() tests to see whether the queue is empty. It needs no parameters and returns a Boolean value.
- size() returns the number of items in the queue. It needs no parameters and returns an integer

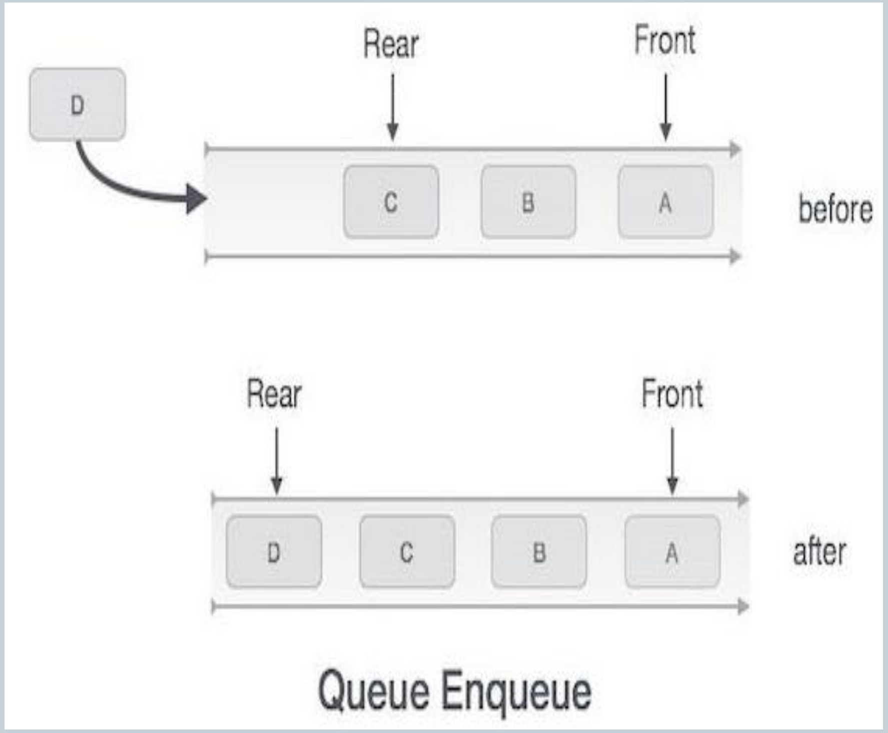
Queue operation

Enqueue Operation

- Queues maintain two data pointers, front and rear. Therefore, its operations are comparatively difficult to implement than that of stacks.

  Algorithm:

  1. Check if the queue is full.
  2. If the queue is full, produce overflow error and exit.
  3. If the queue is not full, increment rear pointer to point the next empty space.
  4. Add data element to the queue location, where the rear is pointing.
  5. Return success.

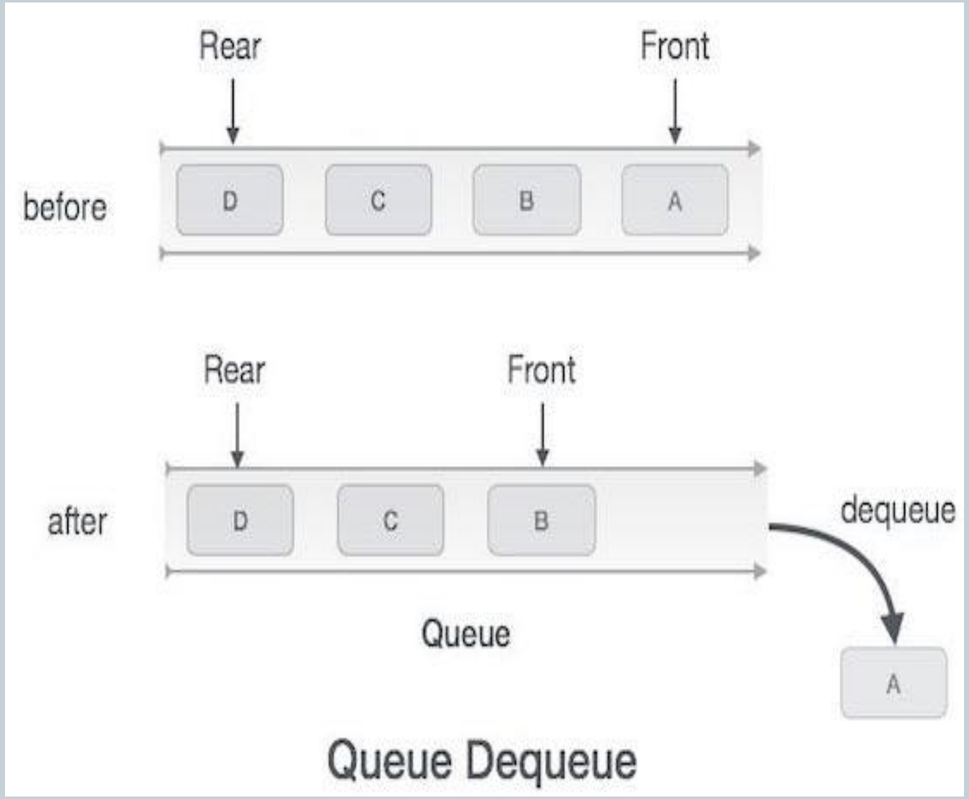


Queue operation

- Dequeue Operation
  - Accessing data from the queue is a process of two tasks – access the data where front is pointing and remove the data after access.

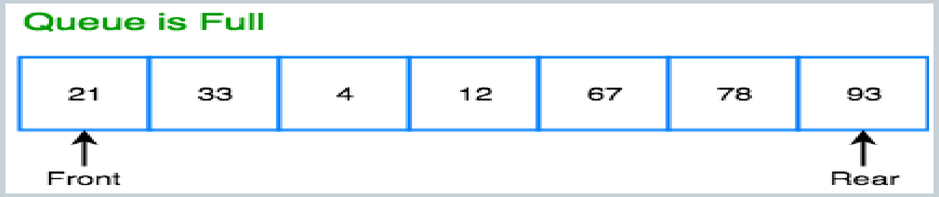
    Algorithm :

    1. Check if the queue is empty.
    2. If the queue is empty, produce underflow error and exit.
    3. If the queue is not empty, access the data where front is pointing.
    4. Increment front pointer to point to the next available data element.
    5. Return success

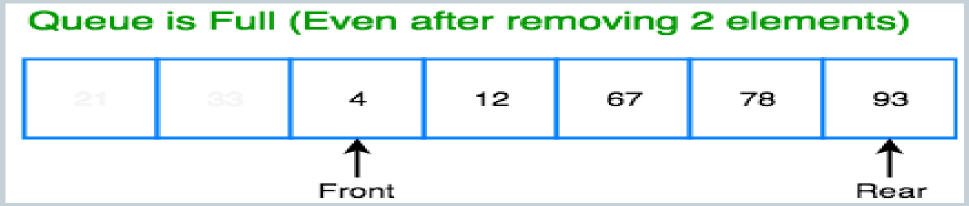


Circular Queue

- In a Linear queue, once the queue is completely full, it's not possible to insert more elements. Even if we dequeue the queue to remove some of the elements, until the queue is reset, no new elements can be inserted. You must be wondering why?



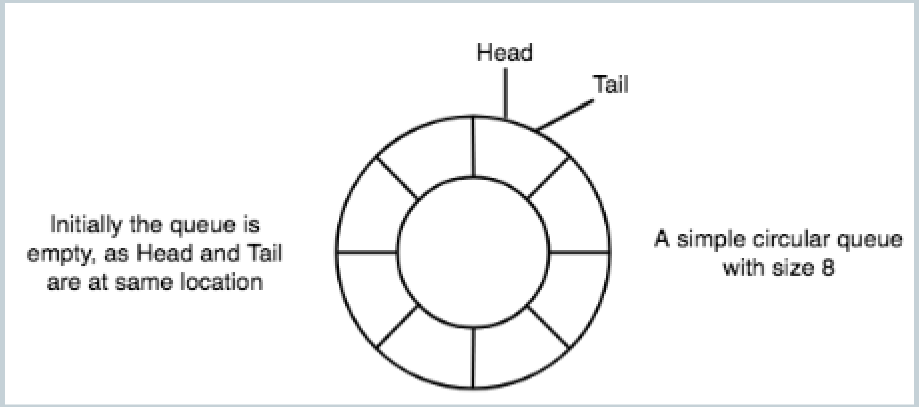
- When we **dequeue** any element to remove it from the queue, we are actually moving the **front** of the queue forward, thereby reducing the overall size of the queue. And we cannot insert new elements, because the **rear** pointer is still at the end of the queue



- The only way is to reset the linear queue, for a fresh start

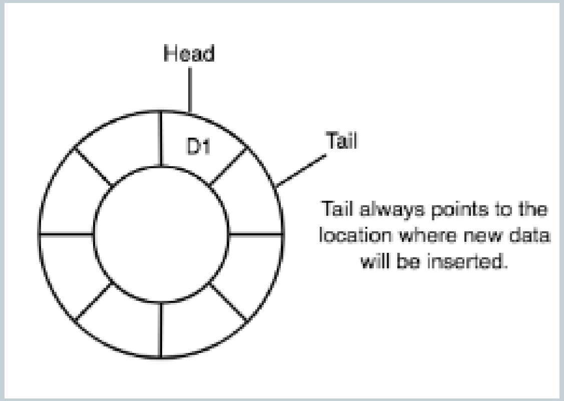
Circular Queue …

- **Circular Queue** is also a linear data structure, which follows the principle of **FIFO**(First In First Out), but instead of ending the queue at the last position, it again starts from the first position after the last, hence making the queue behave like a circular data structure. It is also called **'Ring Buffer'**.
- Basic features of Circular Queue
  1. In case of a circular queue, head pointer will always point to the front of the queue, and tail pointer will always point to the end of the queue.
  2. Initially, the head and the tail pointers will be pointing to the same location, this would mean that the queue is empty

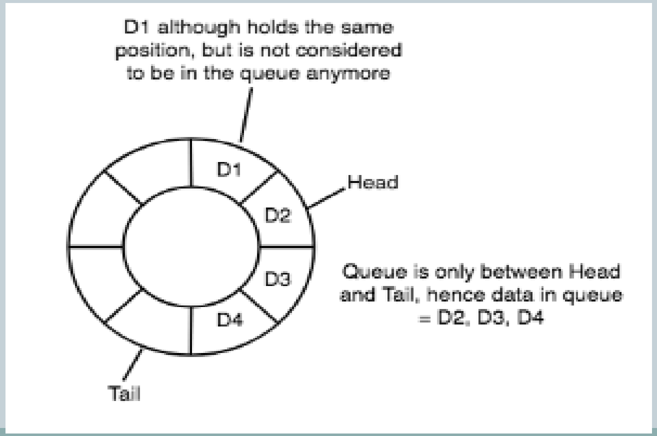

Head
Tail
Initially the queue is empty, as Head and Tail are at same location
A simple circular queue with size 8

Circular Queue …

3. New data is always added to the location pointed by the tail pointer, and once the data is added, tail pointer is incremented to point to the next available location

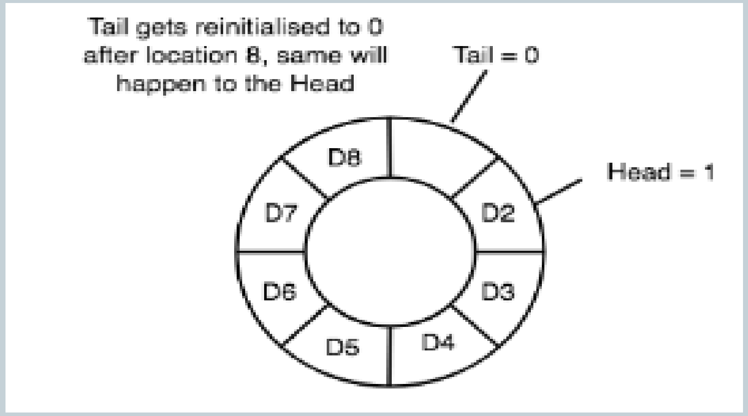

Head
D1
Tail
Tail always points to the location where new data will be inserted.

4. In a circular queue, data is not actually removed from the queue. Only the head pointer is incremented by one position when **dequeue** is executed. As the queue data is only the data between head and tail, hence the data left outside is not a part of the queue anymore, hence removed.

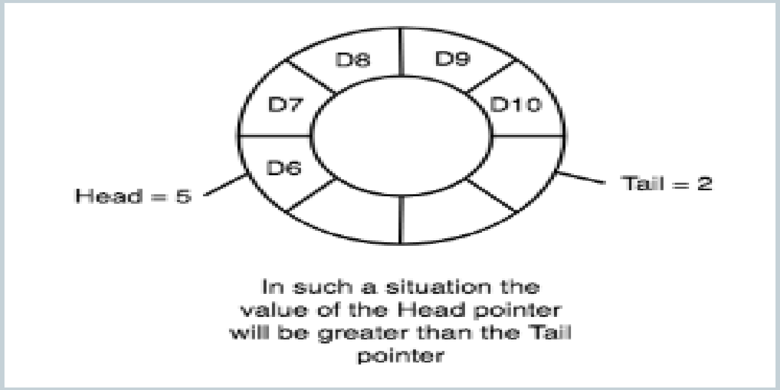

D1 although holds the same position, but is not considered to be in the queue anymore
D1
Head
D2
D3
D4
Queue is only between Head and Tail, hence data in queue = D2, D3, D4
Tail

Circular Queue

5. The head and the tail pointer will get reinitialized to **0** every time they reach the end of the queue


Tail gets reinitialised to 0 after location 8, same will happen to the Head
Tail = 0
Head = 1
D8
D7
D2
D6
D3
D5
D4

6. Also, the head and the tail pointers can cross each other. In other words, head pointer can be greater than the tail. Sounds odd? This will happen when we dequeue the queue a couple of times and the tail pointer gets re initialized upon reaching the end of the queue.

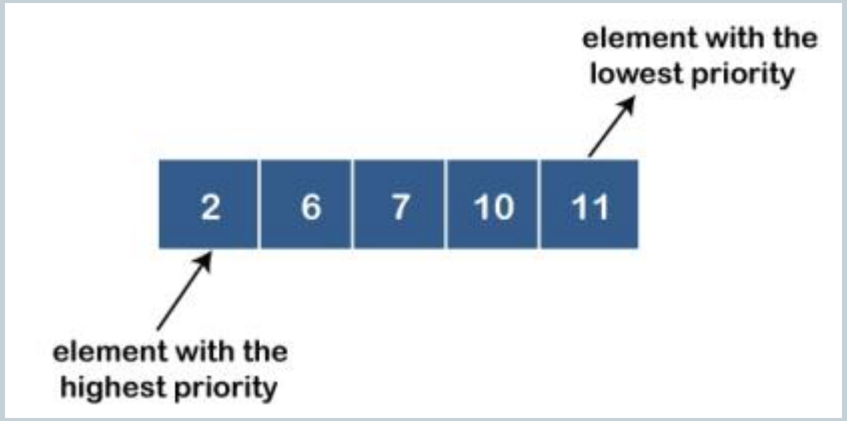

D8
D9
D7
D10
D6
Head = 5
Tail = 2
In such a situation the value of the Head pointer will be greater than the Tail pointer

Priority Queue

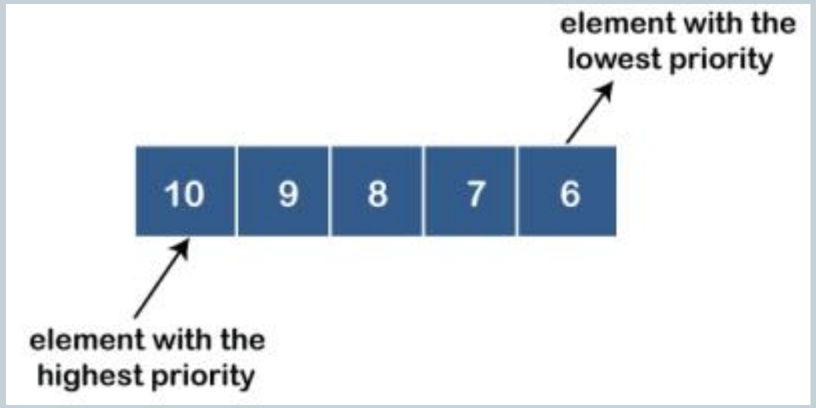
- ✓ A priority queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority, i.e., the element with the highest priority would come first in a priority queue. The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.
- ✓ The priority queue supports only comparable elements, which means that the elements are either arranged in an ascending or descending order.
- ✓ For example, suppose we have some values like 1, 3, 4, 8, 14, 22 inserted in a priority queue with an ordering imposed on the values is from least to the greatest. Therefore, the 1 number would be having the highest priority while 22 will be having the lowest priority

Priority Queue

- Ascending order priority queue
  - In ascending order priority queue, a lower priority number is given as a higher priority in a priority.
  - For example, we take the numbers from 1 to 5 arranged in an ascending order like 1,2,3,4,5; therefore, the smallest number, i.e., 1 is given as the highest priority in a priority queue.


element with the lowest priority
2
6
7
10
11
element with the highest priority

- Descending order priority queue
  - In descending order priority queue, a higher priority number is given as a higher priority in a priority.
  - For example, we take the numbers from 1 to 5 arranged in descending order like 5, 4, 3, 2, 1; therefore, the largest number, i.e., 5 is given as the highest priority in a priority


element with the lowest priority
10
9
8
7
6
element with the highest priority

Applications of Queues

- It is used to schedule the jobs to be processed by the CPU.
- When multiple users send print jobs to a printer, each printing job is kept in the printing queue. Then the printer prints those jobs according to first in first out (FIFO) basis.
- Breadth first search uses a queue data structure to find an element from a graph.
- Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
- In real life scenario, Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free.
- Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e First come first served
- real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.
- Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served

Further Readings

- ✓ Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language, Prentice Hall, 1988.
- ✓ Data Structures and Algorithms; Shi-Kuo Chang; World Scientifi c.
- ✓ Data Structures and Effi cient Algorithms, Burkhard Monien, Thomas Ottmann, Springer.
- ✓ Kruse Data Structure & Program Design, Prentice Hall of India, New Delhi
- ✓ Mark Allen Weles: Data Structure & Algorithm Analysis in C Second Adition. Addison-Wesley publishing
- ✓ RG Dromey, How to Solve it by Computer, Cambridge University Press.
- ✓ Shi-kuo Chang, Data Structures and Algorithms, World Scientifi c
- ✓ Sorenson and Tremblay: An Introduction to Data Structure with Algorithms.
- ✓ Thomas H. Cormen, Charles E, Leiserson & Ronald L. Rivest: Introduction to Algorithms.
- ✓ Prentice-Hall of India Pvt. Limited, New Delhi Timothy A. Budd, Classic Data Structures in C++, Addison Wesley

DATA STRUCTURE AND ALGORITHM

Lecturer: Er. Saroj Ghimire

Qualification: Msc.CSIT, BE(COMPUTER)

Lincoln University College

Overview

- Principles of recursion
- Comparison between recursion and iteration
- Factorial, Fibonacci, GCD & Tower of Hanoi

Objectives

- To learn how to formulate programs recursively.
- To understand and apply the three laws of recursion.

Principles of recursion

- The recursion is a process by which a function calls itself.
- We use recursion to solve bigger problem into smaller sub-problems.
- Recursion is a programming technique using function or algorithm that calls itself one or more times until a specified condition is met at which time the rest of each repetition is processed from the last one called to the first.
- For example, let's look at a recursive definition of a person's ancestors:

  One's parents are one's ancestors. The parents of any ancestor are also ancestors of the person under consideration

  We can write pseudo code to determine whether somebody is someone's ancestor.

```
FUNCTION is Ancestor (Person x, Person y):
    IF x is y's parent, THEN:
        return true
    ELSE
        return is Ancestor (x, y's mom) OR is Ancestor (x, y's dad)
}
```

- Thus, a recursive function usually has a certain structure:
  1. **base case**, which does not call the function itself;
  2. **recursive step**, which calls the function itself and moves closer to the base case.

Comparison between recursion and iteration

Recursion	Iteration
1. Recursion is like piling all of those steps on top of each other and then quashing the mall into the solution.	1. In iteration, a problem is converted into a train of steps that are finished one at a time, one after another.
2. In recursion, each step replicates itself at a smaller scale, so that all of them combined together eventually solve the problem.	2. With iteration, each step clearly leads on to the next, like stepping stones across a river.
3. Not all recursive problem can solved by iteration	3. Any iterative problem is solved recursively
4. It uses Stack	4. It does not use Stack

5. **int fib**(**int** n)
{ **if**(n <= 1)
{ **return** n }
return fib(n-1) + fib(n-2)
}
	5. **int fib**(**int** n)
	{ **if**(n <= 1)
	return n
	a = 0, b = 1 **for**(i = 2 to n)
	{ c = a + b
	a = b
	b = c }
	return c
	}

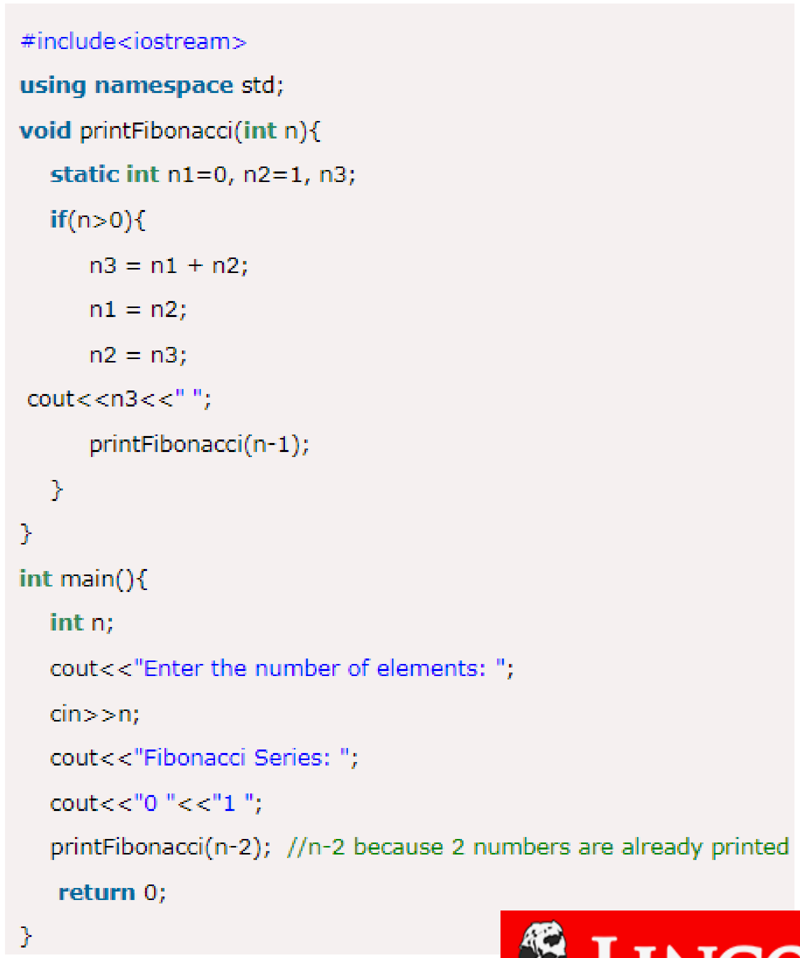
Factorial, Fibonacci, GCD & Tower of Hanoi

- Factorial
  - The factorial of a number is the product of all the integers from 1 to that number.
  - For example, the factorial of 6 is 1*2*3*4*5*6 = 720.
  - Factorial is not defined for negative numbers and the factorial of zero is one, 0! = 1

```cpp
#include<iostream>
using namespace std;

int factorial(int n);

int main()
{
    int n;

    cout << "Enter a positive integer: ";
    cin >> n;

    cout << "Factorial of " << n << " = " << factorial(n);

    return 0;
}

int factorial(int n)
{
    if(n > 1)
        return n * factorial(n - 1);
    else
        return 1;
}
```

Factorial, Fibonacci, GCD & Tower of Hanoi

- Fibonacci
  - The Fibonacci numbers are the numbers in the following integer sequence.0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ……..
  - In mathematical terms, the sequence F_n of Fibonacci numbers is defined by the recurrence relation $F_n = F_{n-1} + F_{n-2}$ with seed values $F_0 = 0$ and $F_1 = 1$



```
#include<iostream>
using namespace std;
void printFibonacci(int n){
    static int n1=0, n2=1, n3;
    if(n>0){
         n3 = n1 + n2;
         n1 = n2;
         n2 = n3;
 cout<<n3<<" ";
         printFibonacci(n-1);
    }
}
int main(){
    int n;
    cout<<"Enter the number of elements: ";
    cin>>n;
    cout<<"Fibonacci Series: ";
    cout<<"0 "<<"1 ";
    printFibonacci(n-2);  //n-2 because 2 numbers are already printed
     return 0;
}
```

Factorial, Fibonacci, GCD & Tower of Hanoi

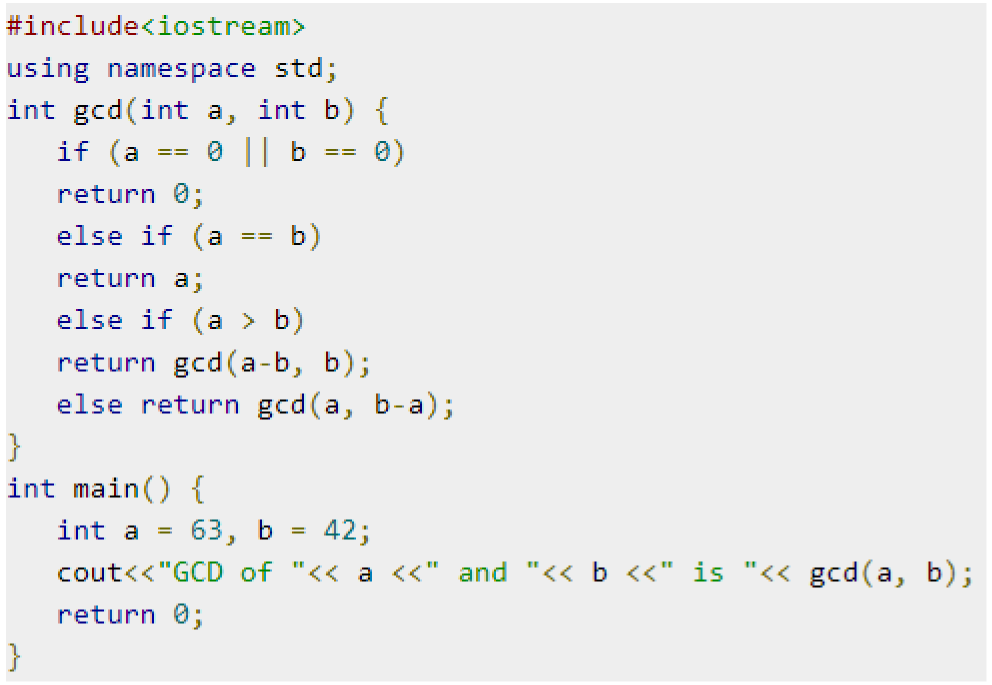
- GCD

- GCD (Greatest Common Divisor) or HCF (Highest Common Factor) of two numbers is the largest number that divides both of them.
- For example: Let’s say we have following two numbers: 63 and 42

  63 = 7 * 3 * 3

  7 * 3 * 2

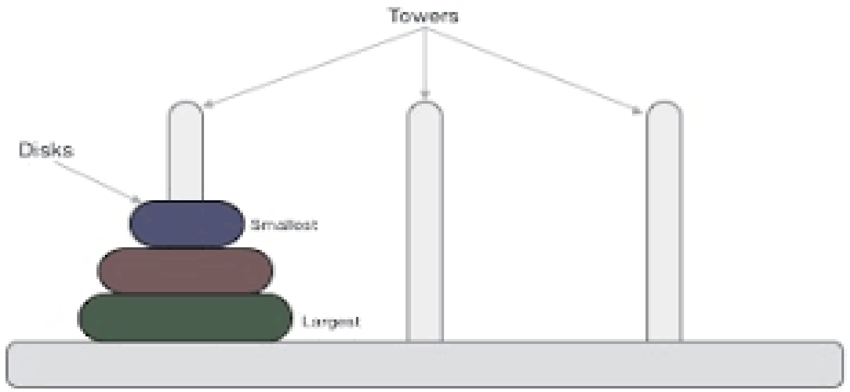
  So, the GCD of 63 and 42 is 21

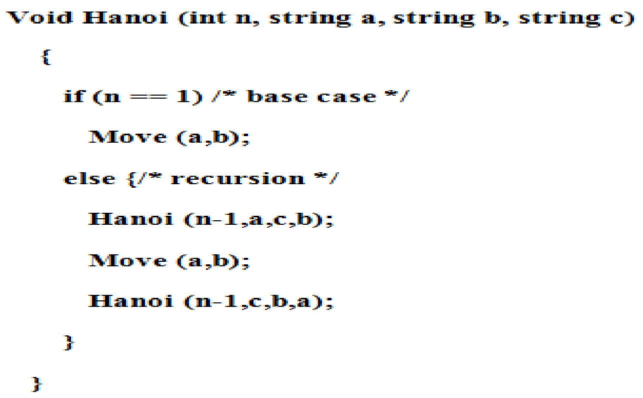


```
#include<iostream>
using namespace std;
int gcd(int a, int b) {
   if (a == 0 || b == 0)
   return 0;
   else if (a == b)
   return a;
   else if (a > b)
   return gcd(a-b, b);
   else return gcd(a, b-a);
}
int main() {
   int a = 63, b = 42;
   cout<<"GCD of "<< a <<" and "<< b <<" is "<< gcd(a, b);
   return 0;
}
```

Tower of Hanoi

- Tower of Hanoi, is a mathematical puzzle which consists of three tower pegs and more than one rings; as depicted below –





```
Void Hanoi (int n, string a, string b, string c)
 {
   if (n == 1) /* base case */
     Move (a,b);
   else {/* recursion */
     Hanoi (n-1,a,c,b);
     Move (a,b);
     Hanoi (n-1,c,b,a);
   }
 }
```

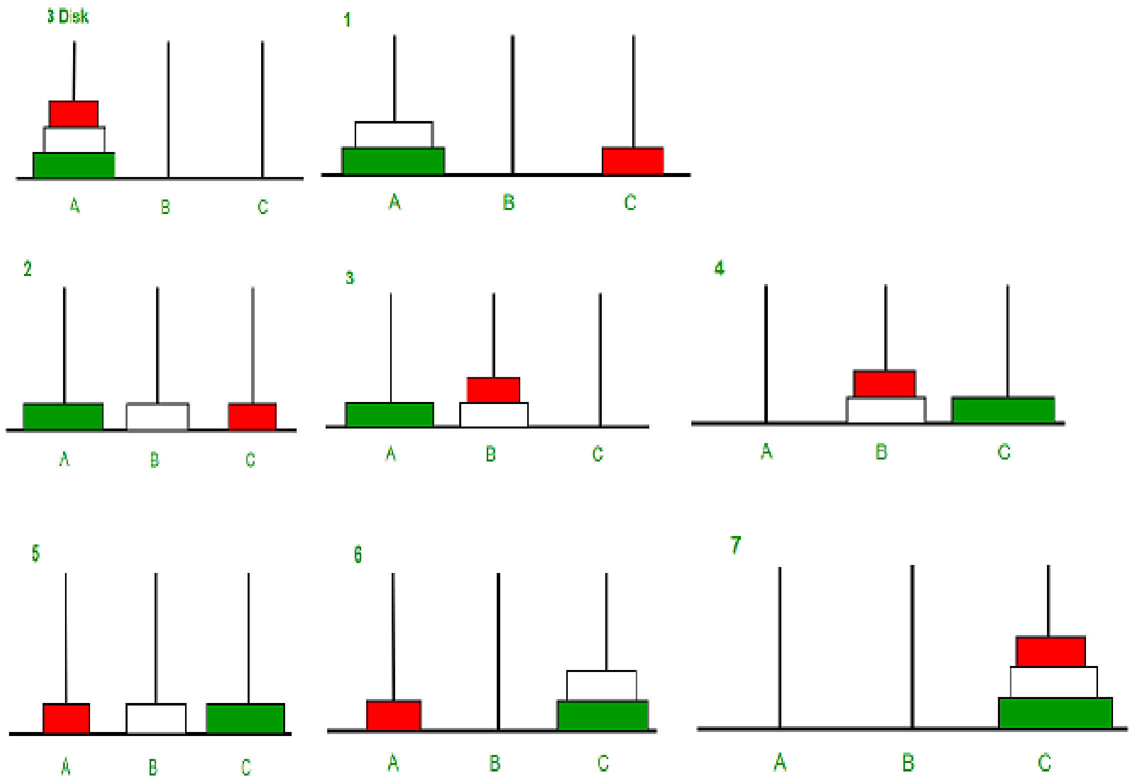
- It consists of three rods and a number of disks of different sizes, which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape. The objective of the puzzle is to move the entire stack to last rod, obeying the following simple rules:
  - Only one disk can be moved at a time.
  - Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
  - No larger disk may be placed on top of a smaller disk.

Tower of Hanoi

- For an even number of disks:
  - make the legal move between pegs A and B (in either direction),
  - make the legal move between pegs A and C (in either direction),
  - make the legal move between pegs B and C (in either direction),
  - repeat until complete.
- For an odd number of disks:
  - make the legal move between pegs A and C (in either direction),
  - make the legal move between pegs A and B (in either direction),
  - make the legal move between pegs B and C (in either direction),
  - repeat until complete.
- In each case, a total of $2^n - 1$ moves are made.

Tower of Hanoi

```
Void Hanoi (int n, string a, string b, string c)
  {
    if (n == 1) /* base case */
      Move ( a, b);
    else {/* recursion */
      Hanoi (n-1,a,c,b);
      Move ( a, b);
      Hanoi (n-1,c,b,a);
    }
  }
```



Further Readings

- Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language, Prentice Hall, 1988.
- Data Structures and Algorithms; Shi-Kuo Chang; World Scientifi c.
- Data Structures and Effi cient Algorithms, Burkhard Monien, Thomas Ottmann, Springer.
- Kruse Data Structure & Program Design, Prentice Hall of India, New Delhi
- Mark Allen Weles: Data Structure & Algorithm Analysis in C Second Adition. Addison-Wesley publishing
- RG Dromey, How to Solve it by Computer, Cambridge University Press.
- Shi-kuo Chang, Data Structures and Algorithms, World Scientifi c
- Sorenson and Tremblay: An Introduction to Data Structure with Algorithms.
- Thomas H. Cormen, Charles E, Leiserson & Ronald L. Rivest: Introduction to Algorithms.
- Prentice-Hall of India Pvt. Limited, New Delhi Timothy A. Budd, Classic Data Structures in C++, Addison Wesley

DATA STRUCTURE AND ALGORITHM

Lecturer: Er. Saroj Ghimire

Qualification: Msc.CSIT, BE(COMPUTER)

Lincoln University College

Overview

- Definition of Searching
- Searching Algorithm: Sequential and Binary search
- Concept of Hash: Hash Tables, Hash Function
- Collision Resolution Techniques

Objectives

- To understand searching algorithm in large and small dataset
- Learning different techniques of hashing and its importance

Definition of Searching

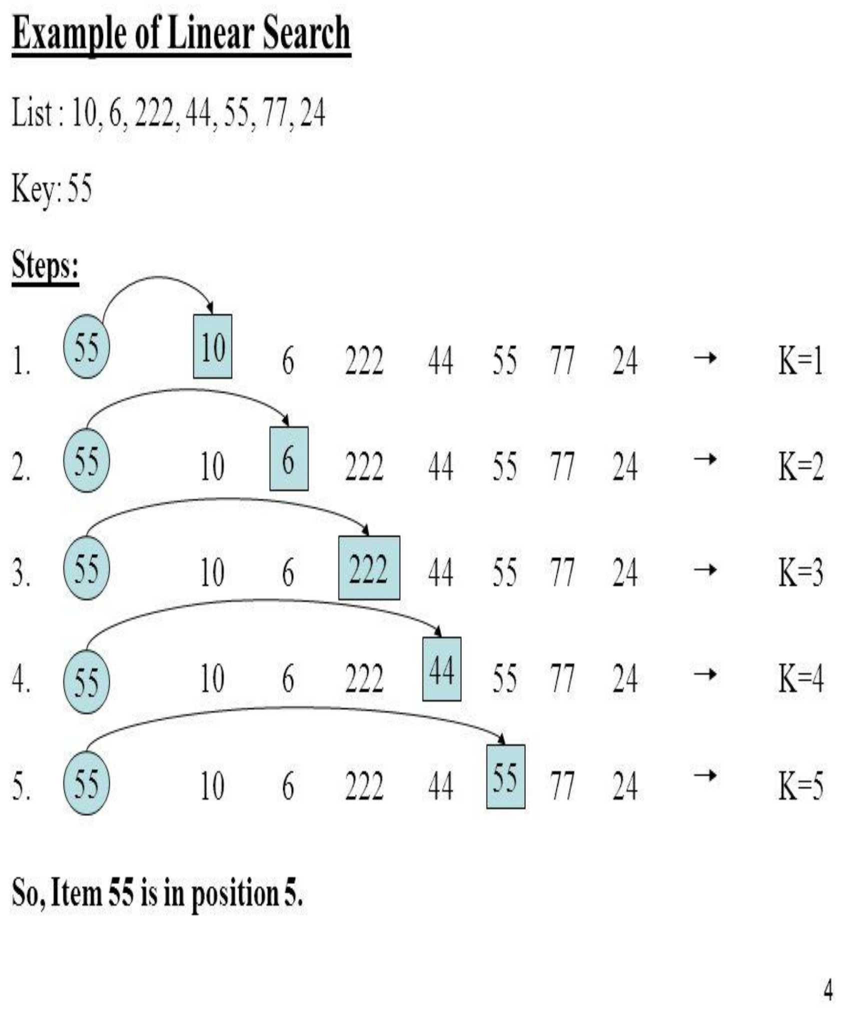
- Searching is a process of checking and finding an element from a list of elements.
- Let A be a collection of data elements, i.e., A is a linear array of say n elements. If we want to find the presence of an element "data" in A, then we have to search for it.
- The search is successful if data does appear in A and unsuccessful if otherwise.
- Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored.
- Based on the type of search operation, these algorithms are generally classified into two categories:
  - Linear search
    - Small arrays
    - Unsorted arrays
  - Binary search
    - Large arrays
    - Sorted arrays

Linear search

- In linear search, each element of an array is read one by one sequentially and it is compared with the desired element.
- A search will be unsuccessful if all the elements are read and the desired element is not found.
- Linear search is mostly used to search an unordered list in which the items are not sorted.

- Let A be an array of n elements, A[1],A[2],A[3], ... A[n]. "data" is the element to be searched. Then this algorithm will find the location "loc" of data in A. Set loc = – 1,if the search is unsuccessful.

```
1. Input an array A of n elements and "data" to be searched and initialize loc = – 1.
2. Initialized i = 0; and repeat through step 3 if (i < n) by incrementing i by one .
3. If (data = A[i])
    1. loc = 1
    2. GOTO step 4
4. If (loc > 0)
    1. Display "data is found and searching is successful"
5. Else
    1. Display "data is not found and searching is unsuccessful"
6. Exit
```



Example of Linear Search

List : 10, 6, 222, 44, 55, 77, 24

Key: 55

Steps:

1. 55 10 6 222 44 55 77 24 → K=1
2. 55 10 6 222 44 55 77 24 → K=2
3. 55 10 6 222 44 55 77 24 → K=3
4. 55 10 6 222 44 55 77 24 → K=4
5. 55 10 6 222 44 55 77 24 → K=5

So, Item 55 is in position 5.

4

Binary Search

- Binary search is an extremely efficient algorithm when it is compared to linear search.
- Binary search technique searches "data" in minimum possible comparisons.
- Suppose the given array is a sorted one, otherwise first we have to sort the array elements.

Binary Search

Algorithm

1. Find the middle element of the array (i.e., n/2 is the middle element if the array or the sub-array contains n elements).
2. Compare the middle element with the data to be searched, then there are following three cases.
   1. If it is a desired element, then search is successful.
   2. If it is less than desired data, then search only the first half of the array, i.e., the elements which come to the left side of the middle element.
   3. If it is greater than the desired data, then search only the second half of the array, i.e., the elements which come to the right side of the middle element.

Repeat the same steps until an element is found or exhaust the search area.

Binary Search (example)

Suppose we have an array of 7 elements

9	10	25	30	40	45	70

1. Following steps are generated if we binary search a data = 45 from the above array.

9	10	25	30	40	45	70
0	1	2	3	4	5	6

LB = 0; UB = 6 (lower bound and upper bound)

mid = (0 + 6)/2 = 3

A[mid] = A[3] = 30

2. Since (A[3] < data) - i.e., 30 < 45 - reinitialize the variable LB, UB and mid

9	10	25	30	40	45	70
0	1	2	3	4	5	6

LB = 3 UB = 6

mid = (3 + 6)/2 = 4

A[mid] = A[4] = 40

Binary Search (example)

3. Since (A[4] < data) - i.e., 40 < 45 - reinitialize the variable LB, UB and mid

9	10	25	30	40	45	70
0	1	2	3	4	5	6

LB = 4 UB = 6

mid = (4 + 6)/2 = 5

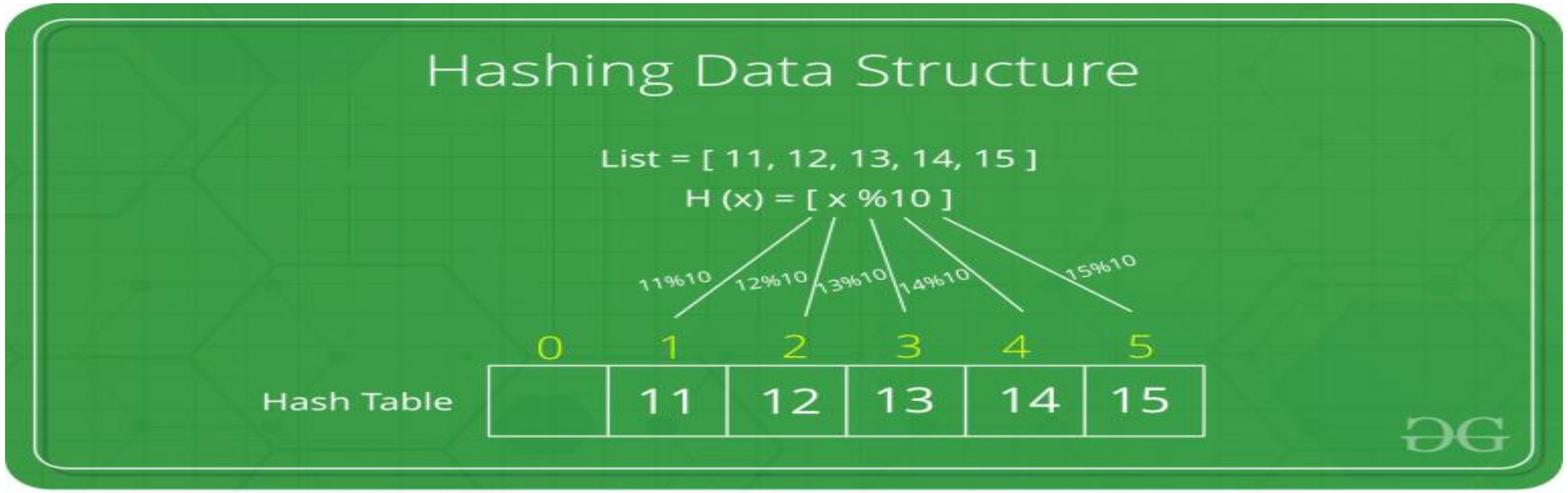
A[mid] = A[5] = 45

Step 4:

Since (A[5] == data) - i.e., 45 == 45 - searching is successful.

Concept of Hash: Hash Tables, Hash Function

- ✓ Hashing is the process of mapping large amount of data item to smaller table with the help of hashing function. Hashing is also known as **Hashing Algorithm** or **Message Digest Function**.
- ✓ Hashing is a technique or process of mapping keys, values into the hash table by using a hash function.
- ✓ It is done for faster access to elements. The efficiency of mapping depends on the efficiency of the hash function used.
- ✓ Let a hash function H(x) maps the value at the index **x%10** in an Array. For example if the list of values is [11,12,13,14,15] it will be stored at positions {1,2,3,4,5} in the array or Hash table respectively.



Concept of Hash: Hash Tables, Hash Function

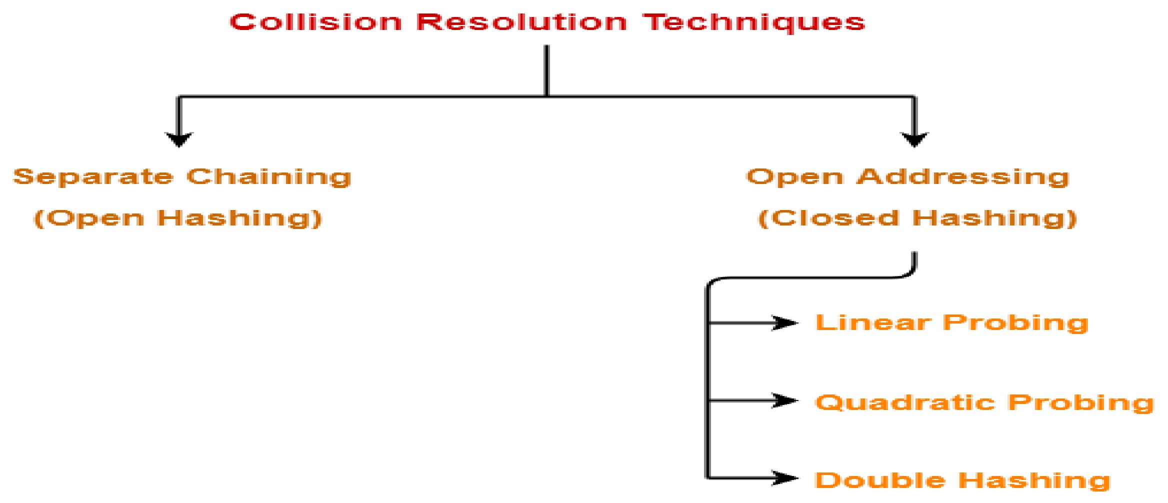
- A fixed process converts a key to a hash key is known as a **Hash Function.**
- This function takes a key and maps it to a value of a certain length which is called a **Hash value** or **Hash.**
- Hash value represents the original string of characters, but it is normally smaller than the original.
- It transfers the digital signature and then both hash value and signature are sent to the receiver. Receiver uses the same hash function to generate the hash value and then compares it to that received with the message.
- If the hash values are same, the message is transmitted without errors.

Concept of Hash: Hash Tables, Hash Function

- Hash table or hash map is a data structure used to store key-value pairs.
- It is a collection of items stored to make it easy to find them later.
- It uses a hash function to compute an index into an array of buckets or slots from which the desired value can be found.
- It is an array of list where each list is known as bucket.It contains value based on the key.
- Hash table is used to implement the map interface and extends Dictionary class. Hash table is synchronized and contains only unique elements

Collision Resolution Techniques

- ✓ When the hash value of a key maps to an already occupied bucket of the hash table, it is called as a **Collision**
- ✓ A collision occurs when two different keys hash to the same value
  - E.g. For Table Size = 17, the keys 18 and 35 hash to the same value
  - 18 mod 17 = 1 and 35 mod 17 = 1
- ✓ Collision Resolution Techniques are the techniques used for resolving or handling the collision.



Separate Chaining

- ✓ To handle the collision, this technique creates a linked list to the slot for which collision occurs.
- ✓ The new key is then inserted in the linked list.
- ✓ These linked lists to the slots appear like chains.

Example :Use separate chaining technique for collision resolution.

Using the hash function 'key mod 7', insert the following sequence of keys in the hash table- 50, 700, 76, 85, 92, 73 101

1. Draw empty hash table and possible range of hash values is (0,6)
2. The first key to be inserted in the hash table = 50.The hash table to which key 50 maps = 50 mod 7 = 1
3. similarly for each item to inserted we calculate hash value

    700mod 7=0

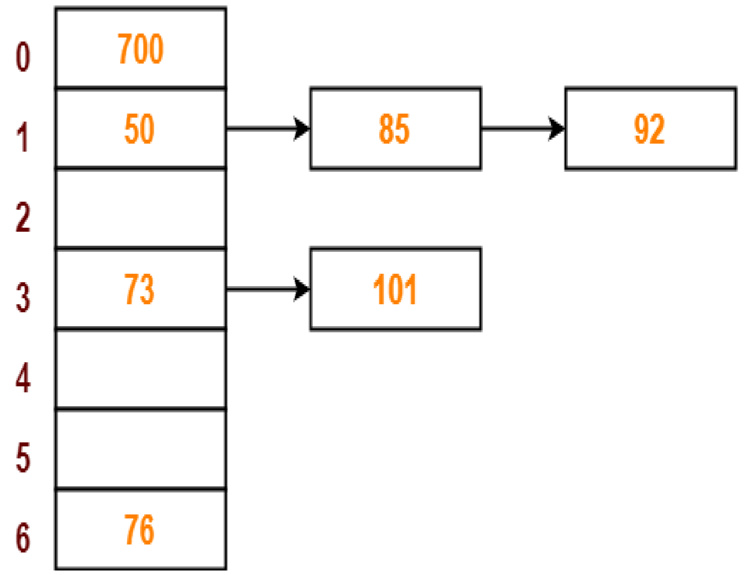
    76mod 7=6

    85mod 7=1

    92mod 7=1

    73mod 7=3

    101mod 7=3



Open Addressing

- ✓ The open addressing is another technique for collision resolution. Unlike chaining, it does not insert elements to some other data-structures.
- ✓ It inserts the data into the hash table itself. The size of the hash table should be larger than the number of keys.
- ✓ There are three different popular methods for open addressing techniques. These methods are −
  - Linear Probing
  - Quadratic Probing
  - Double Hashing

linear probing

- ✓ In linear probing, When collision occurs, we linearly probe for the next bucket.
- ✓ We keep probing until an empty bucket is found.

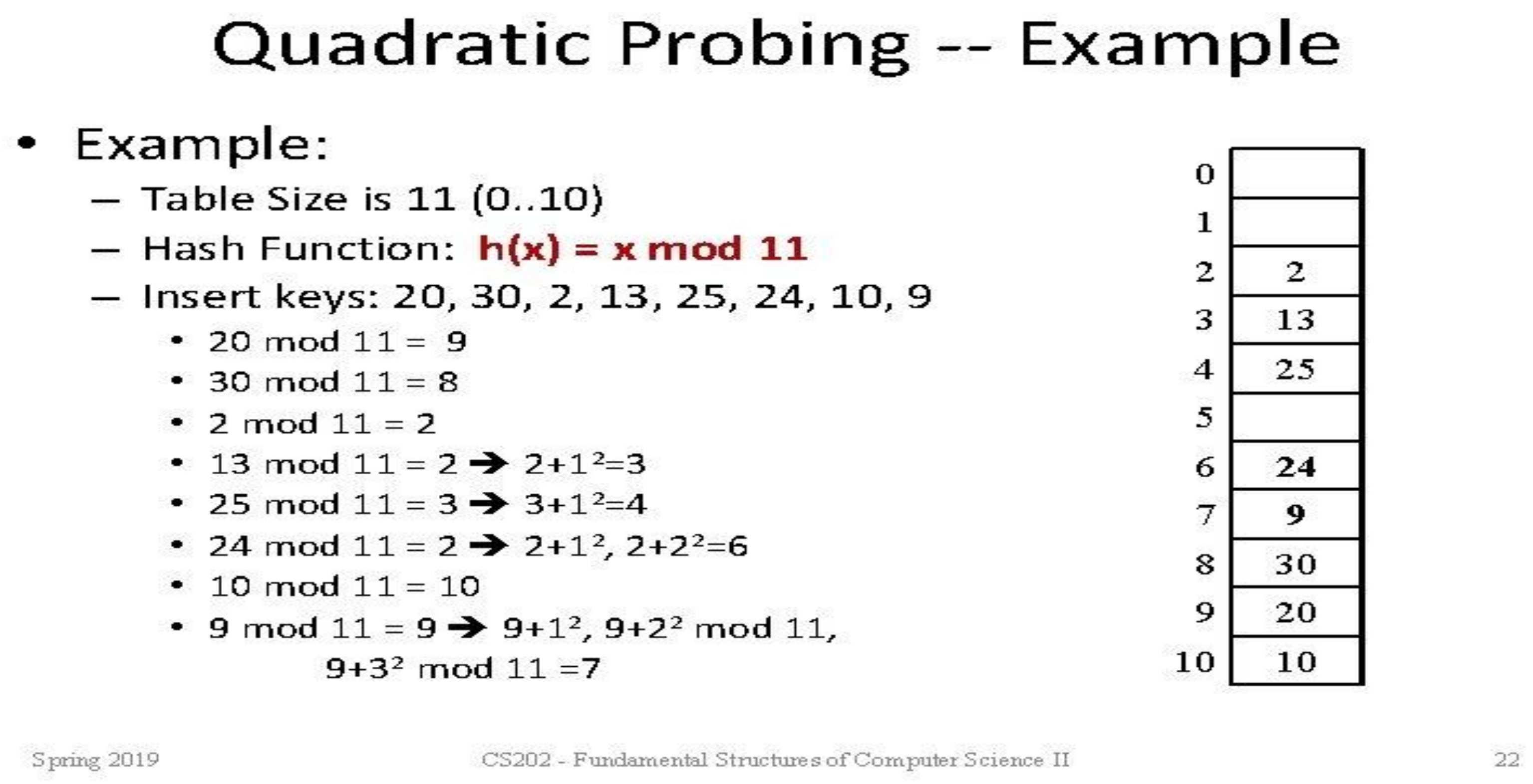
Using the hash function 'key mod 7', insert the following sequence of keys in the hash table- 50, 700, 76, 85, 92, 73,101

1. Draw empty hash table and possible range of hash values is (0,6)
2. The first key to be inserted in the hash table = 50.The hash table to which key 50 maps = 50 mod 7 = 1
3. Bucket of the hash table to which key 700 maps = 700 mod 7 = 0
4. Bucket of the hash table to which key 76 maps = 76 mod 7 = 6.
5. Bucket of the hash table to which key 85 maps = 85 mod 7 = 1. Since bucket-1 is already occupied, so collision occurs. To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found. The empty bucket is 2 so we insert 85 in bucket 2.
6. Bucket of the hash table to which key 92 maps = 92 mod 7 = 1.Since bucket-1 is already occupied, so collision occurs. The first empty bucket is 3 so 92 inserted in it.
7. Bucket of the hash table to which key 73 maps = 73 mod 7 = 3. Since bucket-3 is already occupied, so collision occurs. The first empty bucket is 4 so 73 inserted in it.
8. 101 maps = 101 mod 7 = 3. The first empty bucket is 5 so 101 inserted in it.
9. 76 maps = 76 mod 7 = 6.

0	700
1	50
2	85
3	92
4	73
5	101
6	76

Quadratic Probing

- ✓ In quadratic probing, When collision occurs, we probe for i^2'th bucket in ith iteration.
- ✓ We keep probing until an empty bucket is found.



Quadratic Probing -- Example

- Example:
  - Table Size is 11 (0..10)
  - Hash Function: **h(x) = x mod 11**
  - Insert keys: 20, 30, 2, 13, 25, 24, 10, 9
    - 20 mod 11 = 9
    - 30 mod 11 = 8
    - 2 mod 11 = 2
    - 13 mod 11 = 2 → $2+1^2=3$
    - 25 mod 11 = 3 → $3+1^2=4$
    - 24 mod 11 = 2 → $2+1^2$, $2+2^2=6$
    - 10 mod 11 = 10
    - 9 mod 11 = 9 → $9+1^2$, $9+2^2$ mod 11, $9+3^2$ mod 11 =7

0	
1	
2	2
3	13
4	25
5	
6	24
7	9
8	30
9	20
10	10

Spring 2019 CS202 - Fundamental Structures of Computer Science II 22

Double Hashing

- ✓ In double hashing, We use another hash function. First hash function is typically hash1(key) = key % TABLE_SIZE
- ✓ A popular second hash function is : hash2(key) = PRIME – (key % PRIME) where PRIME is a prime smaller than the TABLE_SIZE.
- ✓ It requires more computation time as two hash functions need to be computed.

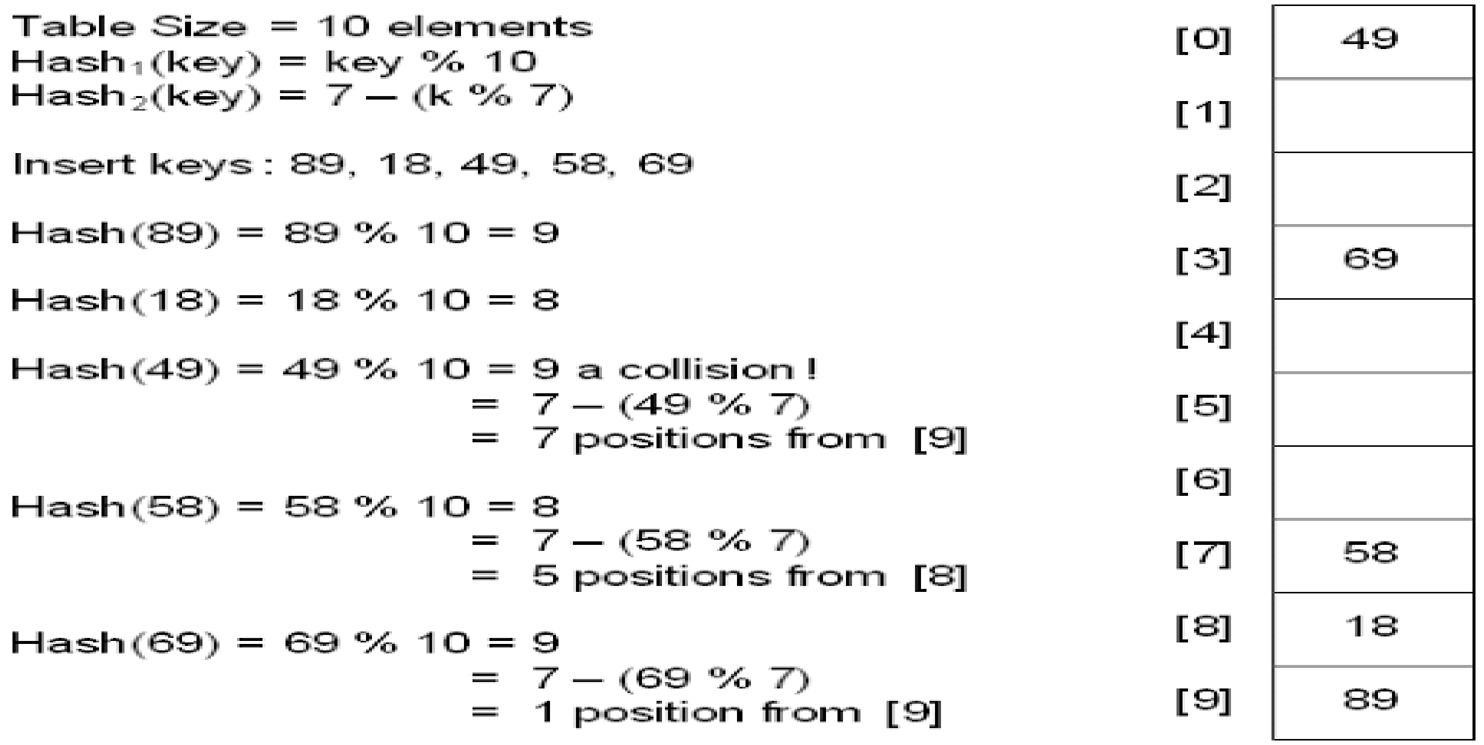


Table Size = 10 elements
$Hash_1(key)$ = key % 10
$Hash_2(key)$ = 7 – (k % 7)

Insert keys : 89, 18, 49, 58, 69

Hash(89) = 89 % 10 = 9

Hash(18) = 18 % 10 = 8

Hash(49) = 49 % 10 = 9 a collision !
= 7 – (49 % 7)
= 7 positions from [9]

Hash(58) = 58 % 10 = 8
= 7 – (58 % 7)
= 5 positions from [8]

Hash(69) = 69 % 10 = 9
= 7 – (69 % 7)
= 1 position from [9]

[0]	49
[1]	
[2]	
[3]	69
[4]	
[5]	
[6]	
[7]	58
[8]	18
[9]	89

Further Readings

- Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language, Prentice Hall, 1988.
- Data Structures and Algorithms; Shi-Kuo Chang; World Scientifi c.
- Data Structures and Effi cient Algorithms, Burkhard Monien, Thomas Ottmann, Springer.
- Kruse Data Structure & Program Design, Prentice Hall of India, New Delhi
- Mark Allen Weles: Data Structure & Algorithm Analysis in C Second Adition. Addison-Wesley publishing
- RG Dromey, How to Solve it by Computer, Cambridge University Press.
- Shi-kuo Chang, Data Structures and Algorithms, World Scientifi c
- Sorenson and Tremblay: An Introduction to Data Structure with Algorithms.
- Thomas H. Cormen, Charles E, Leiserson & Ronald L. Rivest: Introduction to Algorithms.
- Prentice-Hall of India Pvt. Limited, New Delhi Timothy A. Budd, Classic Data Structures in C++, Addison Wesley

DATA STRUCTURE AND ALGORITHM

Lecturer: Er. Saroj ghimire

Qualification: Msc.CSIT, BE(COMPUTER)

Lincoln University College

Overview

- Concept of sorting
- Types of sorting: internal and external
- Sorting algorithm: selection, insertion and bubble
- Divide and conquer algorithm : merge sorting

Objectives

- Understanding sorting algorithm
- To learn how to use sorting techniques

sorting

- Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.
- A sorting algorithm is just a series of orders or instructions. In this, an array is an input, on which the sorting algorithm performs operations to give out a sorted array.
- The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Following are some of the examples of sorting in real-life scenarios –
  - **Telephone Directory** – The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.
  - **Dictionary** – The dictionary stores words in an alphabetical order so that searching of any word becomes easy.

Type of sorting

- There are two different categories in sorting:
  - **Internal sorting**: If the input data is such that it can be adjusted in the main memory at once, it is called internal sorting.
  - **External sorting**: If the input data is such that it cannot be adjusted in the memory entirely at once, it needs to be stored in a hard disk, floppy disk, or any other storage device. This is called external sorting.

Selection Sort

- The selection is a straightforward process of sorting values.
- In this method, to sort the data in ascending order, the 0th element is compared with all other elements.
- If the 0th element is found to be greater than the compared element, the two values get interchanged.
- In this way after the first iteration, the smallest element is placed at 0th position.
- The technique is repeated until the full array gets sorted.

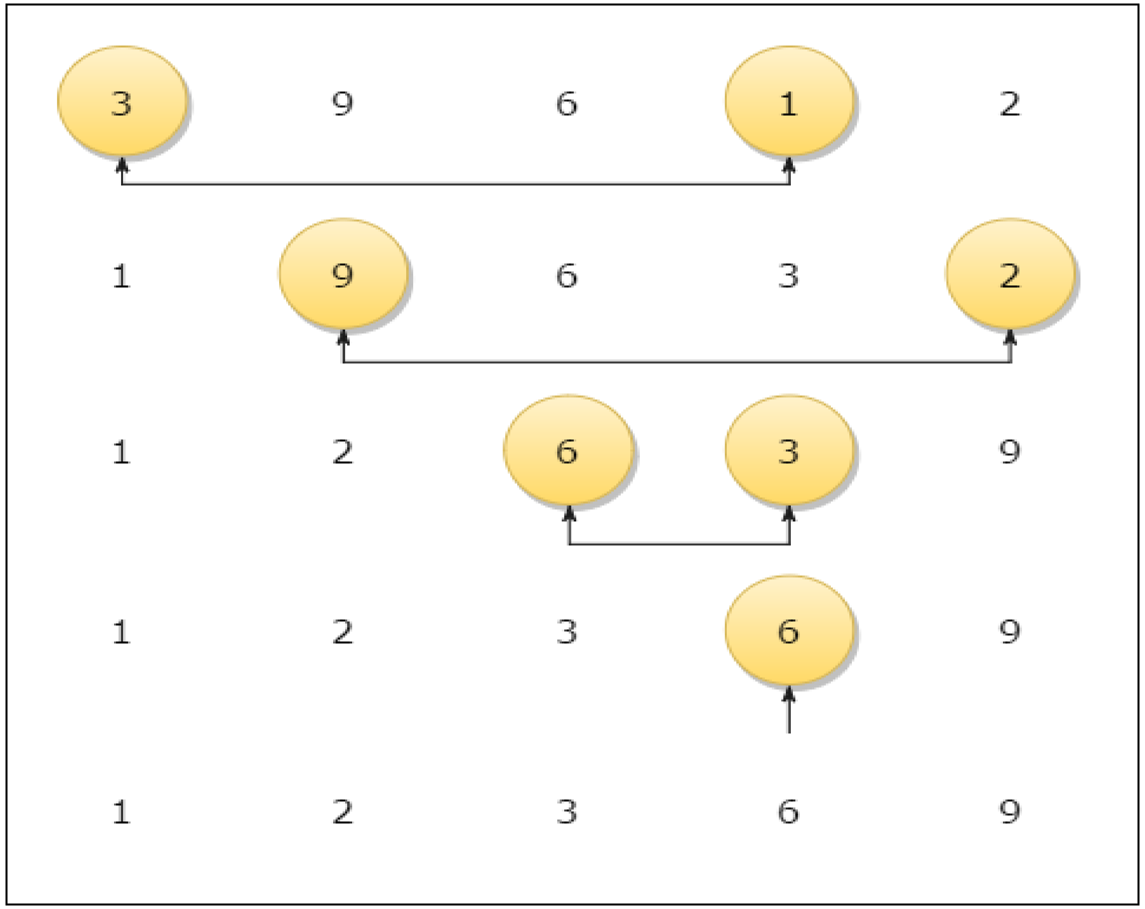


Fig. Selection Sort Technique

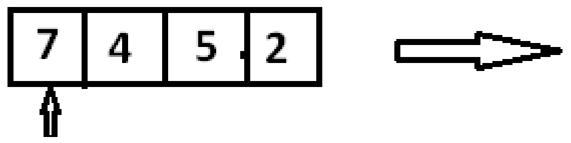
Insertion Sort

- Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands.
- The array is virtually split into a sorted and an unsorted part.
- Values from the unsorted part are picked and placed at the correct position in the sorted part.

Algorithm:

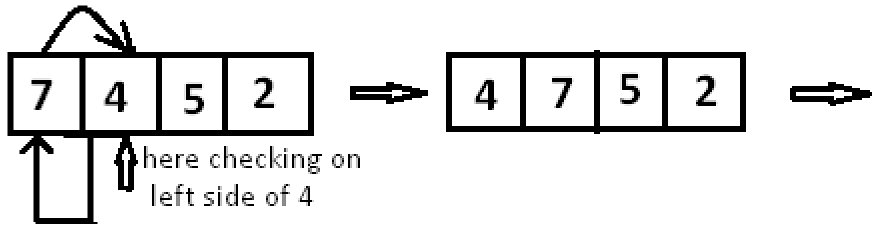
1. If it is the first element, it is already sorted. return 1;
2. Pick next element
3. Compare with all elements in the sorted sub-list
4. Shift all the elements in the sorted sub-list that is greater than the value to be sorted
5. Insert the value
6. Repeat until list is sorted

STEP 1.



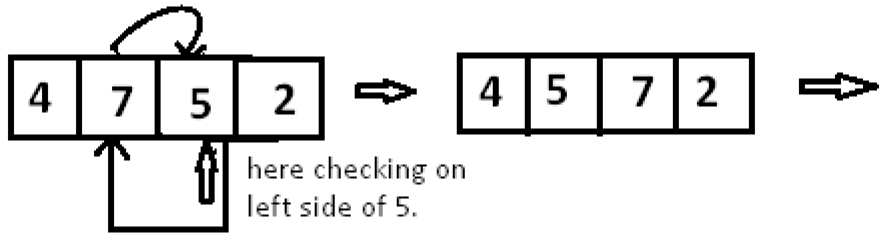
No element on left side of 7,so no change in its position .

STEP 2.



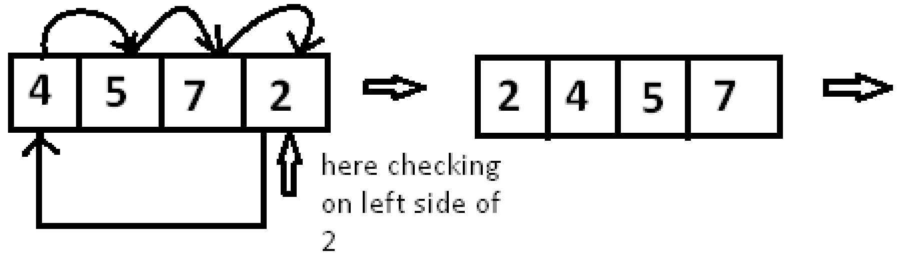
As 7>4 ,therfore 7 will be moved forward and 4 will be moved to 7's position.

STEP 3.



As 7>5,7 will be moved forward,but 4 <5,so no change in position of 4. And 5 will be moved to position of 7.

STEP 4.



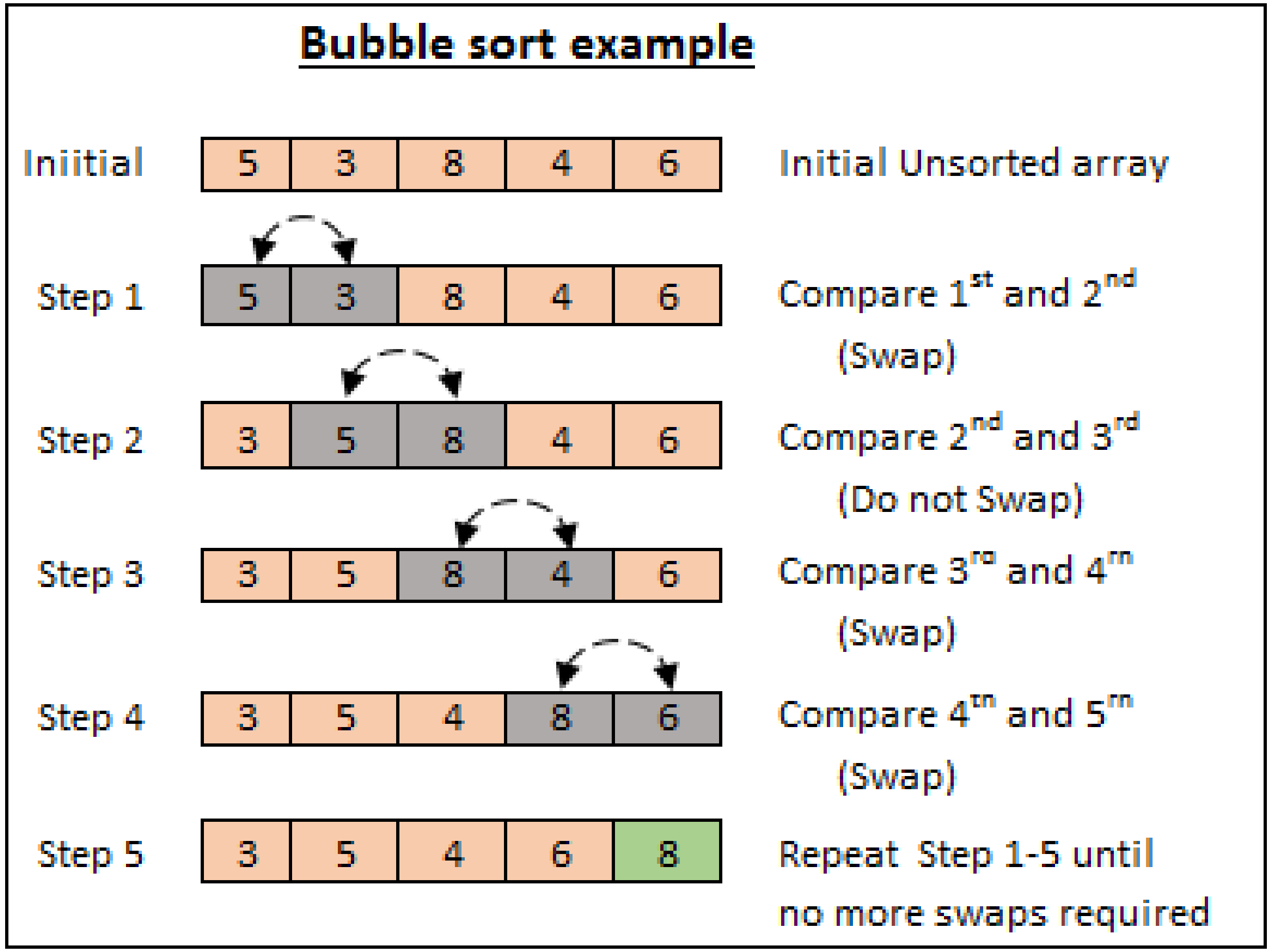
As all the element on left side of 2 are greater than 2,so all the elements will be moved forward and 2 will be shifted to position of 4

Bubble Sort

- Bubble sort is a simple sorting algorithm.
- This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.
- This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where **n** is the number of items.

Algorithm :

1. First iteration(compare and swap)
   - Starting from the first index, compare the first and the second elements.
   - If the first element is greater than the second element, they are swapped.
   - Now, compare the second and the third elements. Swap them if they are not in order.
   - The above process goes on until the last element.
2. Remaining iteration(same process)



Bubble sort example

Iniitial	5	3	8	4	6	Initial Unsorted array
Step 1	5	3	8	4	6	Compare 1st and 2nd (Swap)
Step 2	3	5	8	4	6	Compare 2nd and 3rd (Do not Swap)
Step 3	3	5	8	4	6	Compare 3rd and 4th (Swap)
Step 4	3	5	4	8	6	Compare 4th and 5th (Swap)
Step 5	3	5	4	6	8	Repeat Step 1-5 until no more swaps required

Divide And Conquer Algorithm : Merge Sorting

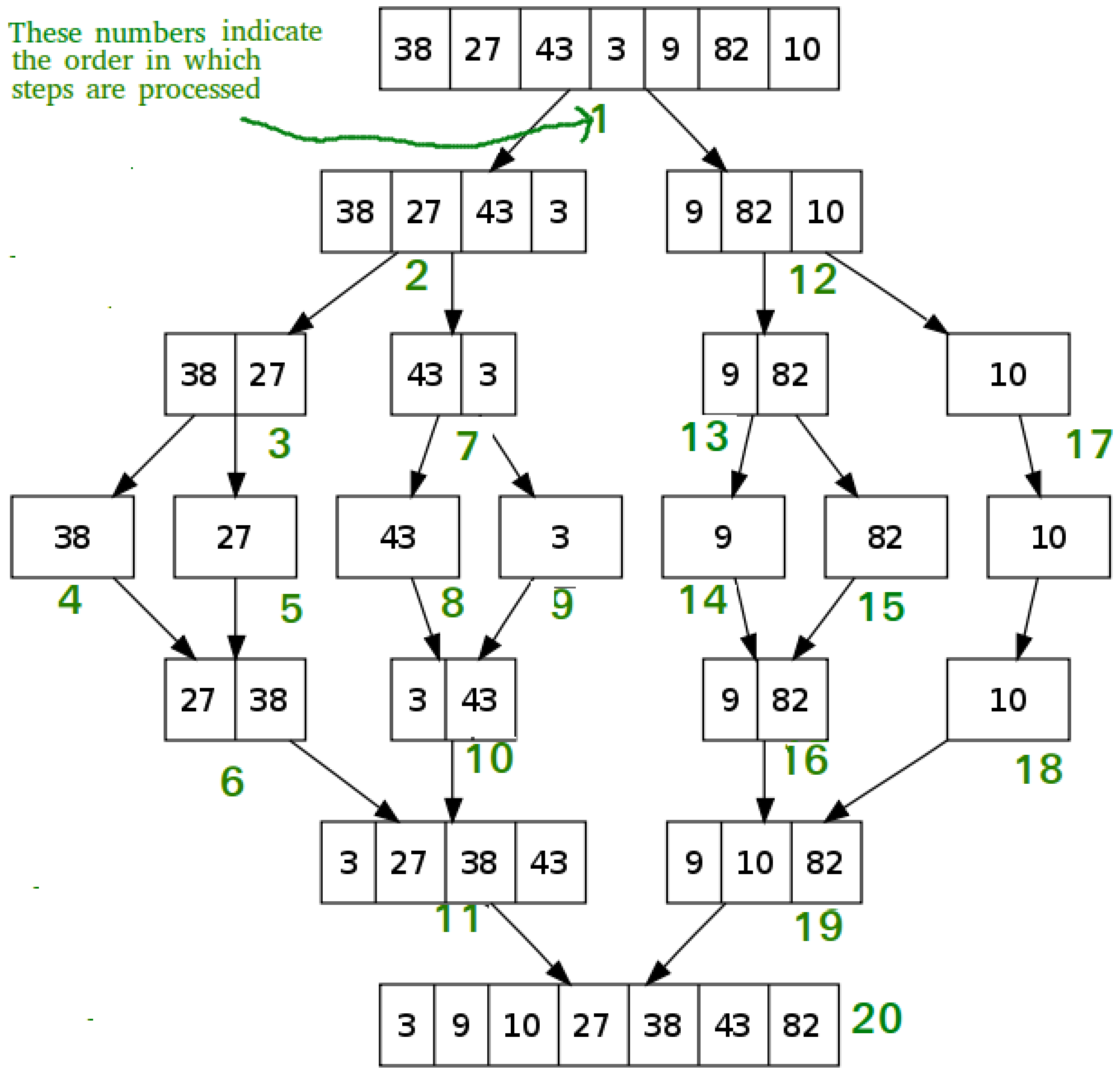
- A **divide and conquer algorithm** is a strategy of solving a large problem by:
  - breaking the problem into smaller sub-problems
  - solving the sub-problems, and
  - combining them to get the desired output.
- The divide-and-conquer technique is the basis of efficient algorithms for many problems, such as sorting (e.g., quicksort, merge sort)

Merge Sorting

- Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n), it is one of the most respected algorithms.
- Merge sort first divides the array into equal halves and then combines them in a sorted manner

Algorithm

- Divide the unsorted list into sub-lists, each containing element.
- Take adjacent pairs of two singleton lists and **merge** them to form a list of 2 elements. N. will now convert into lists of size 2.
- Repeat the process till a single **sorted** list of obtained


These numbers indicate the order in which steps are processed
38 27 43 3 9 82 10
1
38 27 43 3
9 82 10
2
12
38 27
43 3
9 82
10
3
7
13
17
38
27
43
3
9
82
10
4
5
8
9
14
15
27 38
3 43
9 82
10
6
10
16
18
3 27 38 43
9 10 82
11
19
3 9 10 27 38 43 82
20

Further Readings

- Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language, Prentice Hall, 1988.
- Data Structures and Algorithms; Shi-Kuo Chang; World Scientifi c.
- Data Structures and Effi cient Algorithms, Burkhard Monien, Thomas Ottmann, Springer.
- Kruse Data Structure & Program Design, Prentice Hall of India, New Delhi
- Mark Allen Weles: Data Structure & Algorithm Analysis in C Second Adition. Addison-Wesley publishing
- RG Dromey, How to Solve it by Computer, Cambridge University Press.
- Shi-kuo Chang, Data Structures and Algorithms, World Scientifi c
- Sorenson and Tremblay: An Introduction to Data Structure with Algorithms.
- Thomas H. Cormen, Charles E, Leiserson & Ronald L. Rivest: Introduction to Algorithms.
- Prentice-Hall of India Pvt. Limited, New Delhi Timothy A. Budd, Classic Data Structures in C++, Addison Wesley

DATA STRUCTURE AND ALGORITHM

Lecturer: Er. Saroj ghimire

Qualification: Msc.CSIT, BE(COMPUTER)

Lincoln University College

OVERVIEW

- Basics concept of stack
- Stack ADT
- Stack Operations
- Stack Applications
- Conversion From Infix To Postfix/Prefix
- Evaluations Of Postfix/Prefix

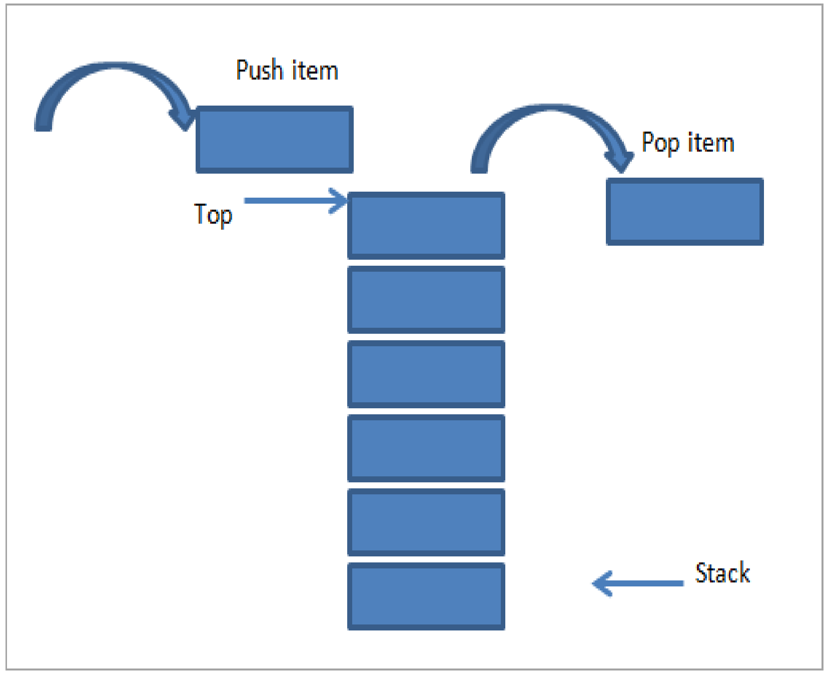
OBJECTIVES

- After studying this unit, you will be able to:
  - Describe the stack model
  - Explain the implementation and applications of stacks

CONCEPT OF STACK

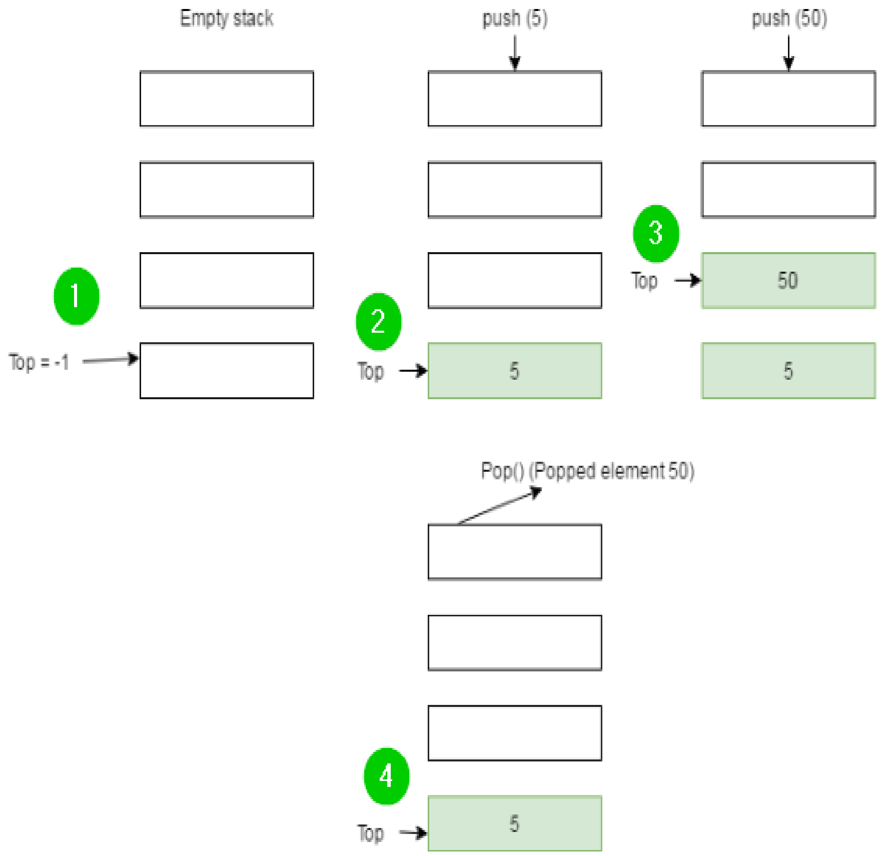
- A stack is similar to real-life stack or a pile of things that we stack one above the other.
- Stack is an **ordered list** of **similar data type**.
- Stack is a **LIFO**(Last in First out) structure or we can say **FILO**(First in Last out).
- **Push()** function is used to insert new elements into the Stack and **Pop()** function is used to remove an element from the stack. Both insertion and removal are allowed at only one end of Stack called **Top**.
- Stack is said to be in **Overflow** state when it is completely full and is said to be in **Underflow** state if it is completely empty

- Pictorial representation of stack



STACK IMPLEMENTATION

1. We initially have an empty stack. The top of an empty stack is set to -1.
2. Next, we have pushed the element 5 into the stack. The top of the stack will points to the element 5.
3. Next, we have pushed the element 50 into the stack. The top of the stack shifts and points to the element 50.
4. We have then performed a pop operation, removing the top element from the stack. The element 50 is popped from the stack. The top of the stack now points to the element 5.



STACK ADT

- **Push(item) -** Adds or pushes an element into the stack.
- **Pop() -** Removes or pops an element out of the stack.
- **peek ()-** Gets the top element of the stack but doesn't remove it.
- **Is Full ()-** Tests if the stack is full return overflow. Top= Size-1
- **Is Empty() -** Tests if the stack is empty return underflow. Top = -1

//Declaration of Stack//

```
class Stack
{
   private:
      int top;
      int ele[MAX];
   public:
      Stack();
      int     isFull();
      int     isEmpty();
      void     push(int item);
      int     pop(int *item);
}
```

STACK OPERATIONS

- Push Operation
  - The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps −
    1. Checks if the stack is full. (**top == SIZE-1**)
    2. If the stack is full, produces an error and exit.
    3. If the stack is not full, increments **top** to point next empty space. (**top++**)
    4. Adds data element to the stack location, where top is pointing. (**stack[top] = value**
    5. Returns success

STACK OPERATIONS(CONTD)

- Pop operation:

The process of removing a new data element from stack is known as a Pop Operation. Push operation involves a series of steps −

1. Checks if the stack is empty. (**top == -1**)
2. If the stack is empty, produces an error and exit.
3. If the stack is not empty, accesses the data element at which **top** is pointing. **stack[top]**
4. Decreases the value of top by 1. (**top--**)
5. Returns success

STACK APPLICATION

- Stacks can be used for expression evaluation.
- Stacks can be used to check parenthesis matching in an expression.
- Stacks can be used for Conversion from one form of expression to another.
- Stacks can be used for Memory Management.
- Stack data structures are used in backtracking problems.
- To evaluate the expressions (postfix, prefix)
- To keep the page-visited history in a Web browser
- To perform the undo sequence in a text editor
- Used in recursion
- To pass the parameters between the functions in a C program
- Can be used as an auxiliary data structure for implementing algorithms
- Can be used as a component of other data structures

CONVERSION FROM INFIX TO POSTFIX/ PREFIX

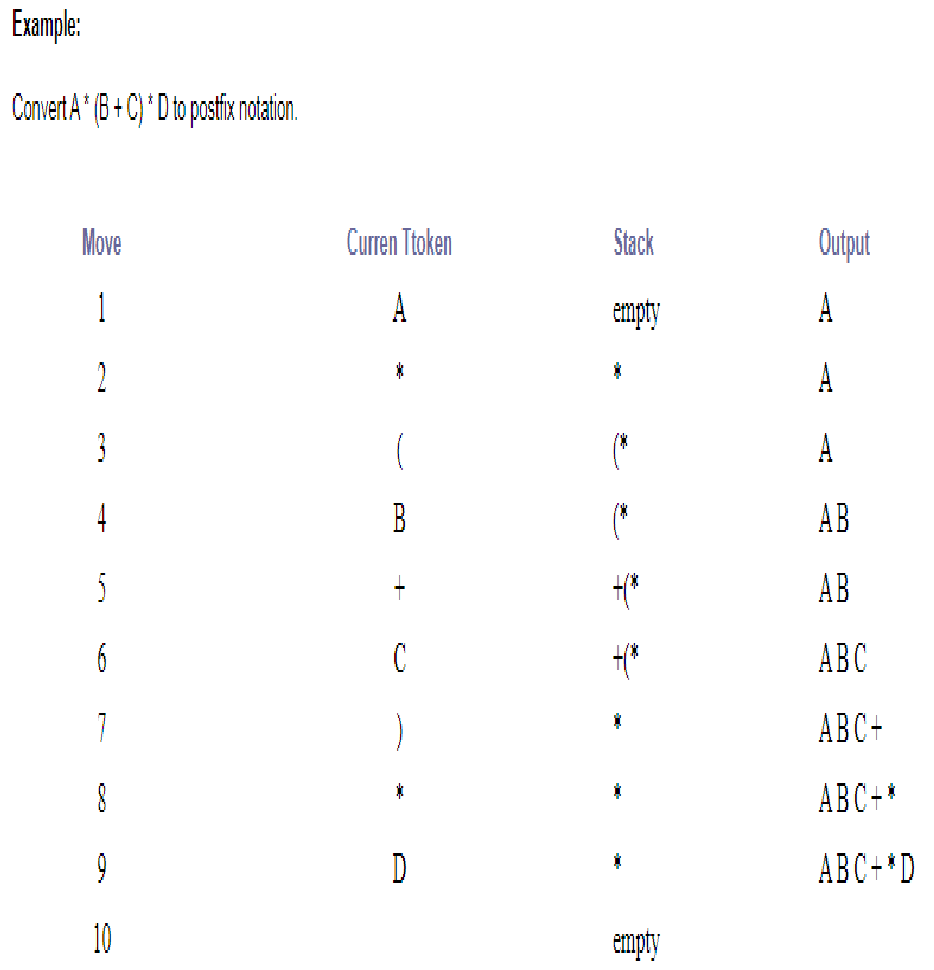
- Infix, Postfix, Prefix

  - Infix expression is the normal expression that consists of operands and operators. For example, **A+B**
  - Postfix expression consists of operands followed by operators. For example, **AB+**
  - Prefix expression consists of operators followed by operands. For example, **+AB**

INFIX TO POSTFIX EXAMPLE

In this case, We use the stacks to convert infix to postfix. We have operator's stack, output's stack and one input string. Operator's stack works as FILO(First In Last Out). Output's stack works as FIFO (First In First Out).

- The following algorithm converts infix to postfix.
  - Scan input string from left to right character by character.
  - If the character is an operand, put it into output stack.
  - If the character is an operator and operator's stack is empty, push operator into operators' stack.
  - If the operator's stack is not empty, there may be following possibilities.
    - If the precedence of scanned operator is greater than the top most operator of operator's stack, push this operator into operand's stack.
    - If the precedence of scanned operator is less than or equal to the top most operator of operator's stack, pop the operators from operand's stack until we find a low precedence operator than the scanned character. Never pop out (**'('**) or (**')'**) whatever may be the precedence level of scanned character.
    - If the character is opening round bracket (**'('**), push it into operator's stack.
    - If the character is closing round bracket (**')'**), pop out operators from operator's stack until we find an opening bracket (**'('**).
    - Now pop out all the remaining operators from the operator's stack and push into output stack



Example:

Convert A * (B + C) * D to postfix notation.

Move	Curren Ttoken	Stack	Output
1	A	empty	A
2	*	*	A
3	(	(*	A
4	B	(*	AB
5	+	+(*	AB
6	C	+(*	ABC
7	)	*	ABC+
8	*	*	ABC+*
9	D	*	ABC+*D
10		empty	

INFIX TO PREFIX

In this case, we have operator's stack, output's stack and one input string. Operator's stack works as FILO(First In Last Out). Output's stack works as FIFO (First In First Out).

- The following algorithm must be followed for infix to prefix conversion.
  - Reverse the input string.
  - Convert the reversed string into infix expression.
  - Now reverse the resulting infix expression obtained from the previous step. The resulting expression is prefix expression

EVALUATIONS OF POSTFIX/ PREFIX

- In this case stack contents the operands. In stead of operators. Whenever any operator occurs in scanning we evaluate with last two elements of stack.
  1. Add the unique symbol # at the end of array post fix.
  2. Scan the symbol of array post fix one by one from left to right.
  3. If symbol is operand, two push in to stack.
  4. If symbol is operator, then pop last two element of stack and evaluate it as [top- 1] operator [top] & push it to stack.
  5. Do the same process until '#' comes in scanning.
  6. Pop the element of stack which will be value of evaluation of post fix arithmetic expression

(Note : a=4 and l= / in next page)

EVALUATIONS OF POSTFIX/ PREFIX(CONTD)

Postfix expression ABCD $+* EF$GHI * –

Evaluate postfix expression where A = 5, B = 5, C = 4, D = 2 E = 2, F= 2 , G = 9, H= 3 now, 4, 5, 2, $, +, *, 2, 2, $, 9, 3, /, *, - , #

Step	**Symbol**	**Operand in stack**
1	4	4
2	5	4, 5
3	4	4,5,4
4	2	4,5,4,2
5	$	4,5,16
6	+	4,21
7	*	84
8	2	84,2
9	2	84, 2,2
10	$	84,4
11	9	84,4,9
12	3	84,4,9,3
13	/	84, 4, 3
14	*	84,12
15	-	72

The required value of postfix expression is 72

FURTHER READINGS

- Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language, Prentice Hall, 1988.
- Data Structures and Algorithms; Shi-Kuo Chang; World Scientifi c.
- Data Structures and Effi cient Algorithms, Burkhard Monien, Thomas Ottmann, Springer.
- Kruse Data Structure & Program Design, Prentice Hall of India, New Delhi
- Mark Allen Weles: Data Structure & Algorithm Analysis in C Second Adition. Addison-Wesley publishing
- RG Dromey, How to Solve it by Computer, Cambridge University Press.
- Shi-kuo Chang, Data Structures and Algorithms, World Scientifi c
- Sorenson and Tremblay: An Introduction to Data Structure with Algorithms.
- Thomas H. Cormen, Charles E, Leiserson & Ronald L. Rivest: Introduction to Algorithms.
- Prentice-Hall of India Pvt. Limited, New Delhi Timothy A. Budd, Classic Data Structures in C++, Addison Wesley