What is Data Structure?

- Data structure is an arrangement of data in computer's memory. It makes the data quickly available to the processor for required operations.
- It is a software artifact which allows data to be stored, organized and accessed.
- It is a structure program used to store ordered data, so that various operations can be performed on it easily.
  For example, if we have an employee's data like name 'ABC' and salary 10000. Here, 'ABC' is of String data type and 10000 is of Float data type.
  We can organize this data as a record like Employee record and collect & store employee's records in a file or database as a data structure like 'ABC' 10000, 'PQR' 15000, 'STU' 5000.
- Data structure is about providing data elements in terms of some relationship for better organization and storage.
- It is a specialized format for organizing and storing data that can be accessed within appropriate ways.

Why is Data Structure important?

- Data structure is important because it is used in almost every program or software system.
- It helps to write efficient code, structures the code and solve problems.
- Data can be maintained more easily by encouraging a better design or implementation.
- Data structure is just a container for the data that is used to store, manipulate and arrange. It can be processed by algorithms.
  For example, while using a shopping website like Flipkart or Amazon, the users know their last orders and can track them. The orders are stored in a database as records.
  However, when the program needs them so that it can pass the data somewhere else (such as to a warehouse) or display it to the user, it loads the data in some form of data structure.

Types of Data Structure

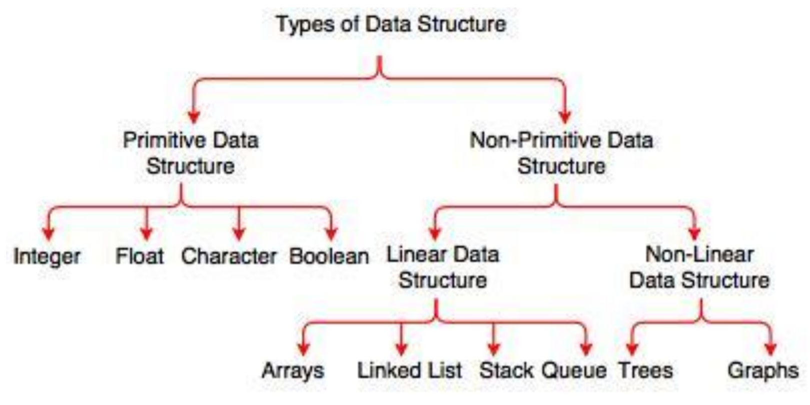


Fig. Types of Data Structure

A. Primitive Data Type

- Primitive data types are the data types available in most of the programming languages.
- These data types are used to represent single value.
- It is a basic data type available in most of the programming language.

Data type	Description
Integer	Used to represent a number without decimal point.
Float	Used to represent a number with decimal point.
Character	Used to represent single character.
Boolean	Used to represent logical values either true or false.

B. Non-Primitive Data Type

- Data type derived from primary data types are known as Non-Primitive data types.
- Non-Primitive data types are used to store group of values.

It can be divided into two types:

1. Linear Data Structure
2. Non-Linear Data Structure

1. Linear Data Structure

- Linear data structure traverses the data elements sequentially.
- In linear data structure, only one data element can directly be reached.
- It includes array, linked list, stack and queues.

Types	Description
Arrays	Array is a collection of elements. It is used in mathematical problems like matrix, algebra etc. each element of an array is referenced by a subscripted variable or value, called subscript or index enclosed in parenthesis.
Linked list	Linked list is a collection of data elements. It consists of two parts: Info and Link. Info gives information and Link is an address of next node. Linked list can be implemented by using pointers.
Stack	Stack is a list of elements. In stack, an element may be inserted or deleted at one end which is known as Top of the stack. It performs two operations: Push and Pop. Push means adding an element in stack and Pop means removing an element in stack. It is also called Last-in-First-out (LIFO).
Queue	Queue is a linear list of element. In queue, elements are added at one end called rear and the existing elements are deleted from other end called front. It is also called as First-in-First-out (FIFO).

2. Non-Linear Data Structure

- Non-Linear data structure is opposite to linear data structure.
- In non-linear data structure, the data values are not arranged in order and a data item is connected to several other data items.
- It uses memory efficiently. Free contiguous memory is not required for allocating data items.
- It includes trees and graphs.

Type	Description
Tree	Tree is a flexible, versatile and powerful non-linear data structure. It is used to represent data items processing hierarchical relationship between the grandfather and his children & grandchildren. It is an ideal data structure for representing hierarchical data.
Graph	Graph is a non-linear data structure which consists of a finite set of ordered pairs called edges. Graph is a set of elements connected by edges. Each elements are called a vertex and node.

Abstract Data type (ADT)

What is ADT?

- ADT stands for **Abstract Data Type.**
- It is an abstraction of a data structure.
- Abstract data type is a mathematical model of a data structure.

- It describes a container which holds a finite number of objects where the objects may be associated through a given binary relationship.
- It is a logical description of how we view the data and the operations allowed without regard to how they will be implemented.
- ADT concerns only with what the data is representing and not with how it will eventually be constructed.
- It is a set of objects and operations. For example, List, Insert, Delete, Search, Sort.

It consists of following three parts:

1. Data
2. Operation
3. Error

1. Data describes the structure of the data used in the ADT.

2. Operation describes valid operations for the ADT. It describes its interface.

3. Error describes how to deal with the errors that can occur.

Advantages of ADT

- ADT is reusable and ensures robust data structure.
- It reduces coding efforts.
- Encapsulation ensures that data cannot be corrupted.
- ADT is based on principles of Object Oriented Programming (OOP) and Software Engineering (SE).
- It specifies error conditions associated with operations.

What is Stack?

- Stack is an ordered list of the same type of elements.
- It is a linear list where all insertions and deletions are permitted only at one end of the list.
- Stack is a LIFO (Last In First Out) structure.

- In a stack, when an element is added, it goes to the top of the stack.

Definition
"Stack is a collection of similar data items in which both insertion and deletion operations are performed based on LIFO principle".

There are two basic operations performed in a Stack:

1. Push()
2. Pop()

1. Push() function is used to add or insert new elements into the stack.

2. Pop() function is used to delete or remove an element from the stack.

- When a stack is completely full, it is said to be **Overflow state** and if stack is completely empty, it is said to be **Underflow state**.
- Stack allows operations at **one end only**. Stack behaves like a real life stack, for example, in a real life, we can remove a plate or dish from the top of the stack only or while playing a deck of cards, we can place or remove a card from top of the stack only.
  Similarly, here also, we can only access the top element of a stack.
- According to its LIFO structure, the element which is inserted last, is accessed first.

Implementation of Stack

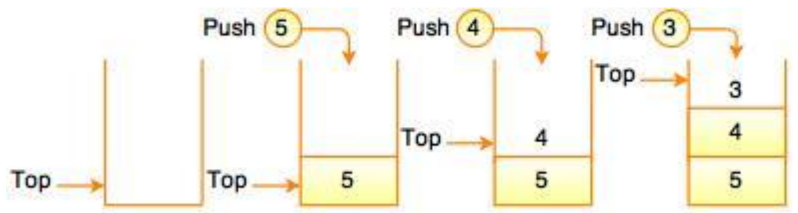


Fig. Insertion of Elements in a Stack

The above diagram represents a stack insertion operation. In a stack, inserting and deleting of elements are performed at a single position which is known as, **Top**.

Insertion operation can be performed using Push() function. New element is added at top of the stack and removed from top of the stack, as shown in the diagram below:

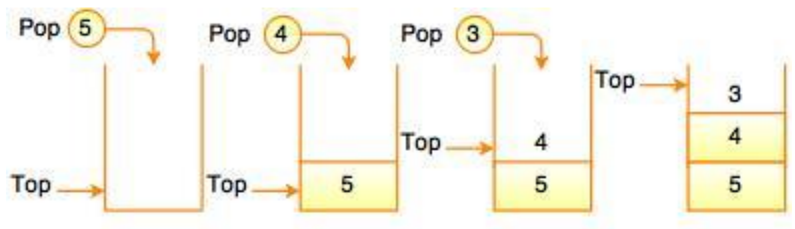


Fig. Deletion of Elements in a Stack

An element is removed from top of the stack. Delete operation is based on LIFO principle. This operation is performed using a Pop() function. It means that the insertion and deletion operations are performed at one end i.e at Top.

Following table shows the Position of Top which indicates the status of stack:

Position of Top	Status of Stack
-1	Stack is empty.
0	Only one element in a stack.
N - 1	Stack is full.
N	Stack is overflow. (Overflow state)

Stack using Array

Stack can be implemented using one-dimensional array. One-dimensional array is used to hold elements of a stack. Implementing a stack using array can store fixed number of data values. In a stack, initially top is set to -1. Top is used to keep track of the index of the top most element.

Stack can be defined using array as follows:

```
typedef struct stack
    {
        int element [MAX];
        int top;
    }stack;
```

In the above code snippet, MAX is a constant and can store defined number of elements. When the value of top becomes MAX – 1 after a series of insertion, it means that stack is full.

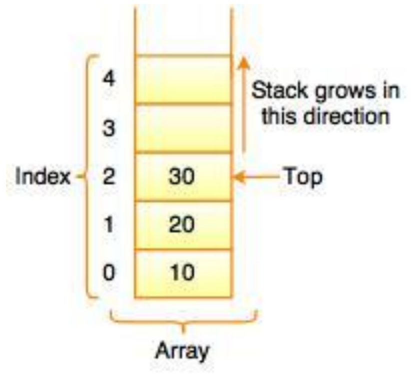


Fig. Implementation Stack using Array

Stack Operations

As we studied, there are two important operations used for adding the elements and removing the elements. They are Push() and Pop().

Following are the other operations used in Stack:

Operations	Description
Peek()	The peek() function gets the top element of the stack, without deleting it.
isEmpty()	The isEmpty() function checks whether the stack is empty or not.
isFull()	The isFull() function is used to check whether the stack is full or not.

Applications of Stack

In a stack, only limited operations are performed because it is restricted data structure. The elements are deleted from the stack in the reverse order.

Following are the applications of stack:

1. Expression Evaluation
2. Expression Conversion
    i. Infix to Postfix
    ii. Infix to Prefix

   iii. Postfix to Infix
   iv. Prefix to Infix
3. Backtracking
4. Memory Management

Expression Representation

There are three popular methods used for representation of an expression:

Infix	A + B	Operator between operands.
Prefix	+ AB	Operator before operands.
Postfix	AB +	Operator after operands.

1. Conversion of Infix to Postfix

Algorithm for Infix to Postfix

Step 1: Consider the next element in the input.

Step 2: If it is operand, display it.

Step 3: If it is opening parenthesis, insert it on stack.

Step 4: If it is an operator, then

- If stack is empty, insert operator on stack.
- If the top of stack is opening parenthesis, insert the operator on stack
- If it has higher priority than the top of stack, insert the operator on stack.
- Else, delete the operator from the stack and display it, repeat Step 4.

Step 5: If it is a closing parenthesis, delete the operator from stack and display them until an opening parenthesis is encountered. Delete and discard the opening parenthesis.

Step 6: If there is more input, go to Step 1.

Step 7: If there is no more input, delete the remaining operators to output.

Example: Suppose we are converting 3*3/(4-1)+6*2 expression into postfix form.

Following table shows the evaluation of Infix to Postfix:

Expression	Stack	Output
3	Empty	3
*	*	3
3	*	33
/	/	33*
(	/(	33*
4	/(	33*4
-	/(-	33*4
1	/(-	33*41
)	-	33*41-
+	+	33*41-/
6	+	33*41-/6
*	+*	33*41-/62
2	+*	33*41-/62
	Empty	**33*41-/62*+**

So, the Postfix Expression is **33*41-/62*+**

2. Infix to Prefix

Algorithm for Infix to Prefix Conversion:

Step 1: Insert ")" onto stack, and add "(" to end of the A .

Step 2: Scan A from right to left and repeat Step 3 to 6 for each element of A until the stack is empty .

Step 3: If an operand is encountered, add it to B .

Step 4: If a right parenthesis is encountered, insert it onto stack .

Step 5: If an operator is encountered then,
a. Delete from stack and add to B (each operator on the top of stack) which has same or higher precedence than the operator.
b. Add operator to stack.

Step 6: If left parenthesis is encountered then ,

a. Delete from the stack and add to B (each operator on top of stack until a left parenthesis is encountered).
b. Remove the left parenthesis.

Step 7: Exit

3. Postfix to Infix

Following is an algorithm for Postfix to Infix conversion:

Step 1: While there are input symbol left.

Step 2: Read the next symbol from input.

Step 3: If the symbol is an operand, insert it onto the stack.

Step 4: Otherwise, the symbol is an operator.

Step 5: If there are fewer than 2 values on the stack, show error /* input not sufficient values in the expression */

Step 6: Else,
a. Delete the top 2 values from the stack.
b. Put the operator, with the values as arguments and form a string.
c. Encapsulate the resulted string with parenthesis.
d. Insert the resulted string back to stack.

Step 7: If there is only one value in the stack, that value in the stack is the desired infix string.

Step 8: If there are more values in the stack, show error /* The user input has too many values */

Example: Suppose we are converting efg-+he-sh-o+/* expression into Infix.

Following table shows the evaluation of Postfix to Infix:

efg-+he-sh-o+/*	NULL
fg-+he-sh-o+/*	“e”
g-+he-sh-o+/*	“f” “e”
-+he-sh-o+/*	“g” “f” “e”

+he-sh-o+/*	“f”-“g” “e”
he-sh-o+/*	“e+f-g”
e-sh-o+/*	“h” “e+f-g”
-sh-o+/*	“e” “h” “e+f-g”
sh-o+/*	“h-e” “e+f-g”
h-o+/*	“s” “h-e” “e+f-g”
-o+/*	“h” “s” “h-e” “e+f-g”
o+/*	“h-s” “h-e” “e+f-g”
+/*	“o” “s-h” “h-e” “e+f-g”
/*	“s-h+o” “h-e” “e+f-g”
*	“(h-e)/(s-h+o)” “e+f-g”
NULL	**“(e+f-g)* (h-e)/(s-h+c**
So, the Infix Expression is **(e+f-g)* (h-e)/(s-h+o)**	

What is Queue?

- Queue is a linear data structure where the first element is inserted from one end called **REAR** and deleted from the other end called as **FRONT**.
- **Front** points to the **beginning** of the queue and **Rear** points to the **end** of the queue.
- Queue follows the **FIFO (First - In - First Out)** structure.
- According to its FIFO structure, element inserted first will also be removed first.
- In a queue, one end is always used to insert data (enqueue) and the other is used to delete data (dequeue), because queue is open at both its ends.
- The enqueue() and dequeue() are two important functions used in a queue.

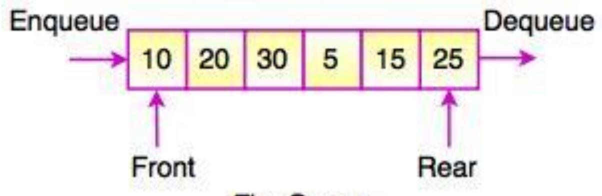


Fig. Queue

Operations on Queue

Following are the basic operations performed on a Queue.

Operations	Description
enqueue()	This function defines the operation for adding an element into queue.
dequeue()	This function defines the operation for removing an element from queue.
init()	This function is used for initializing the queue.
Front	Front is used to get the front data item from a queue.
Rear	Rear is used to get the last item from a queue.

Queue Implementation

- **Array** is the easiest way to implement a queue. Queue can be also implemented using Linked List or Stack.

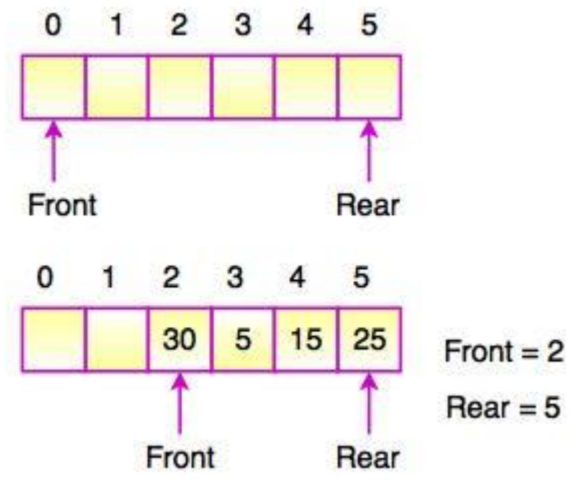


Fig. Implementation of Queue using Array

- In the above diagram, Front and Rear of the queue point at the first index of the array. (Array index starts from 0).
- While adding an element into the queue, the Rear keeps on moving ahead and always points to the position where the next element will be inserted. Front remains at the first index.
    - Queue is an abstract data type which can be implemented as a linear or circular list. It has a front and rear.

      There are four types of Queue:

      1. Simple Queue
      2. Circular Queue
      3. Priority Queue
      4. Dequeue (Double Ended Queue)
    - ## 1. Simple Queue
    - Simple queue defines the simple operation of queue in which insertion occurs at the rear of the list and deletion occurs at the front of the list.

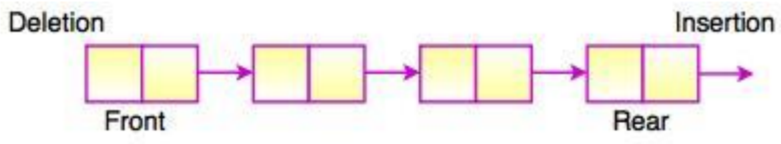


Fig. Simple Queue

Circular Queue

- In a circular queue, all nodes are treated as circular. Last node is connected back to the first node.
- Circular queue is also called as **Ring Buffer.**
- It is an abstract data type.
- Circular queue contains a collection of data which allows insertion of data at the end of the queue and deletion of data at the beginning of the queue.

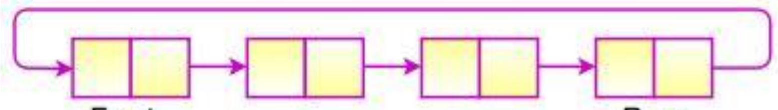


Fig. Circular Queue

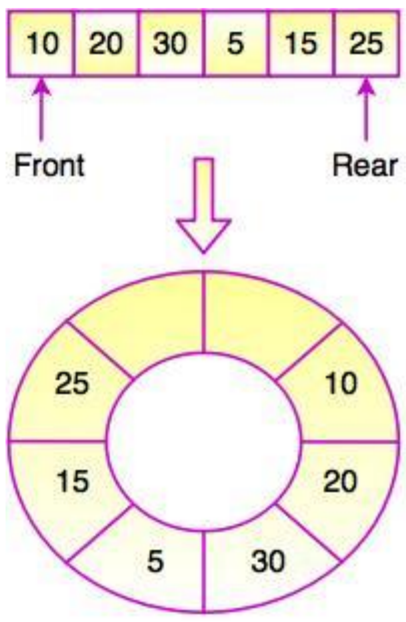
- 

Fig. Circular Queue

The above figure shows the structure of circular queue. It stores an element in a circular way and performs the operations according to its FIFO structure.

Priority Queue

- Priority queue contains data items which have some preset priority. While removing an element from a priority queue, the data item with the highest priority is removed first.
- In a priority queue, insertion is performed in the order of arrival and deletion is performed based on the priority.

Dequeue (Double Ended Queue)

- In Double Ended Queue, insert and delete operation can be occur at both ends that is front and rear of the queue.

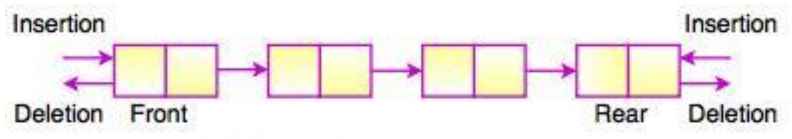


Fig. Double Ended Queue (Dequeue)

What is Linked List?

Linked list is a linear data structure. It is a collection of data elements, called nodes pointing to the next node by means of a pointer.

Linked list is used to create trees and graphs.

In linked list, each node consists of its own data and the address of the next node and forms a chain.

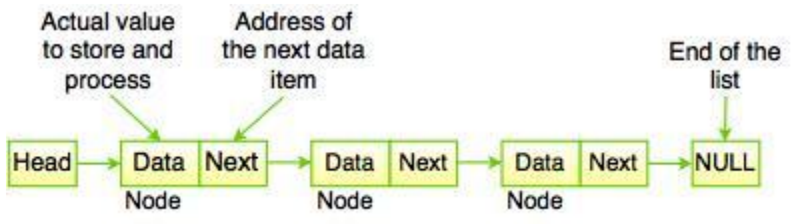

Actual value to store and process
Address of the next data item
End of the list
Head
Data
Next
Data
Next
Data
Next
NULL
Node
Node
Node

Fig. Linked List

The above figure shows the sequence of linked list which contains data items connected together via links. It can be visualized as a chain of nodes, where every node points to the next node.

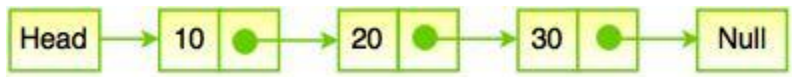
Linked list contains a link element called **first** and each link carries a **data item**. Entry point into the linked list is called the **head of the list.**

Link field is called next and each link is linked with its next link. Last link carries a link to null to mark the end of the list.

Note: Head is not a separate node but it is a reference to the first node. If the list is empty, the head is a null reference.

Linked list is a dynamic data structure. While accessing a particular item, start at the head and follow the references until you get that data item.

Linked list is used while dealing with an unknown number of objects:



In the above diagram, Linked list contains two fields - First field contains value and second field contains a link to the next node. The last node signifies the end of the list that means NULL.

The real life **example of Linked List** is that of Railway Carriage. It starts from engine and then the coaches follow. Coaches can traverse from one coach to other, if they connected to each other.

Advantages of Linked List

- Linked list is dynamic in nature which allocates the memory when required.
- In linked list, stack and queue can be easily executed.
- It reduces the access time.
- Insert and delete operation can be easily implemented in linked list.

Disadvantages of Linked List

- Reverse traversing is difficult in linked list.
- Linked list has to access each node sequentially; no element can be accessed randomly.
- In linked list, the memory is wasted as pointer requires extra memory for storage.

Following are the operations that can be performed on a Linked List:

1. Create
2. Insert
3. Delete
4. Traverse
5. Search
6. Concatenation

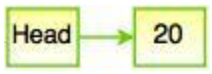
7. Display

1. Create

- Create operation is used to create constituent node when required.
- In create operation, memory must be allocated for one node and assigned to head as follows.

Creating first node

```
head = (node*) malloc (sizeof(node));
head -> data = 20;
head -> next = NULL;
```



2. Insert

- Insert operation is used to insert a new node in the linked list.
- Suppose, we insert a node B(New Node), between A(Left Node) and C(Right Node), it is represented as:

point B.next to B

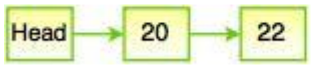
```
NewNode.next -> RightNode;
```

We can insert an element using three cases:

i. At the beginning of the list
ii. At a certain position (Middle)
iii. At the end

Inserting an element

```
node* nextnode = malloc(sizeof(node));
nextnode -> data = 22;
nextnode -> next = NULL;
head -> next = nextnode;
```



The above figure represents the example of create operation, where the next element (i.e 22) is added to the next node by using insert operation.

i. At the beginning of the list

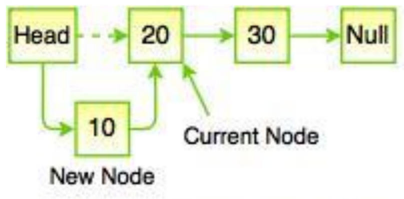


Fig. Inserting a node at the beginning of the list

New node becomes the new head of the linked list because it is always added before the head of the given linked list.

ii. At certain position (Middle)

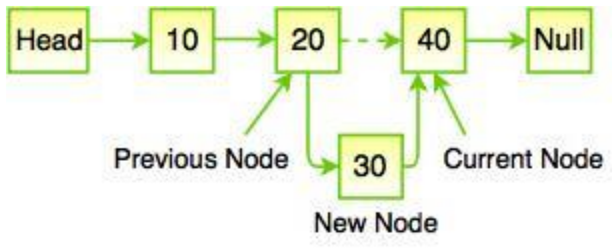


Fig. Inserting a node at middle of the list

While inserting a node in middle of a linked list, it requires to find the current node. The dashed line represents the old node which points to new node.

iii. At the end

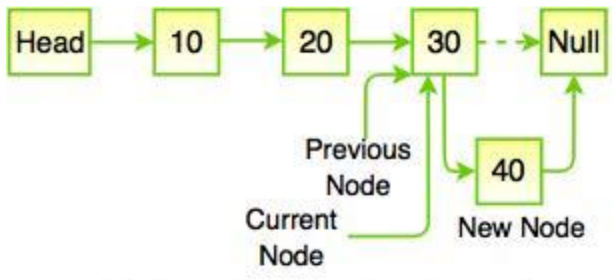


Fig. Inserting a node at the end of the list

While inserting a node at the end of the list, it is achieved by comparing the element values.

3. Delete

- Delete operation is used to delete node from the list.
- This operation is more than one step process.

We can delete an element using three cases:

i. From the beginning of the list
ii. From the middle
iii. From the end

Deleting the element

int delete (node** head, node* n);   // Delete the node n if exists.

i. From the beginning of the list

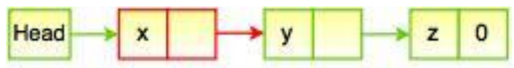
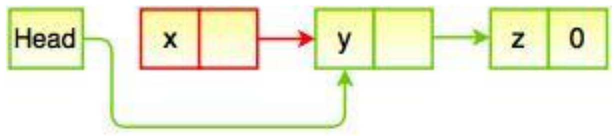


Fig. Deleting a node from the beginning

When deleting the node from the beginning of the list then there is no relinking of nodes to be performed; it means that the first node has no preceding node. The above figure shows the removing node with x. However, it requires to fix the pointer to the beginning of the list which is shown in the figure below:



ii. From the middle

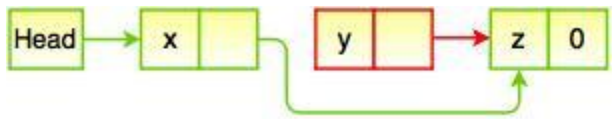


Fig. Deleting a node from the middle

Deleting a node from the middle requires the preceding node to skip over the node being removed.

The above figure shows the removal of node with x. It means that there is a need refer to the node before we can remove it.

iii. From the end

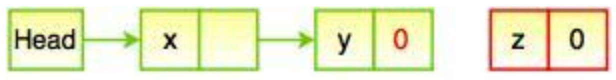


Fig. Deleting a node from the end

Deleting a node from the end requires that the preceding node becomes the new end of the list that points to nothing after it. The above figure shows removing the node with z.

4. Traverse

- Traverse operations is a process of examining all the nodes of linked list from the end to the other end.
- In traverse operation, recursive function is used to traverse a linked list in a reverse order.

The following code snippet represents traversing a node in a linked list:

```
void traverse(node *head)
{
 if(head != NULL)
 {
  traverse (head -> next);
  printf("%d", head -> data);
 }
}
```

5. Search

- Search operation is used for finding a particular element in a linked list.
- Sequential search is the most common search used on linked list structure.
- Search operation ends with a success if the element is found.
- If the element is not found, search ends in a failure.

6. Concatenation

Concatenation is the process of appending a second list to the end of the first list.

7. Display

- Display operation is used to print each and every node's information.
- This operation displays the complete list.

Linked List using Arrays

- Array of linked list is an important data structure used in many applications. It is an interesting structure to form a useful data structure. It combines static and dynamic structure. Static means array and dynamic means linked list used to form a useful data structure. This array of linked list structure is appropriate for applications.

- ## Difference between Array and Linked List

Array	**Linked List**
Array is a collection of elements having same data type with common name.	Linked list is an ordered collection of elements which are connected by links.
Elements can be accessed randomly.	Elements cannot be accessed randomly. It can be accessed only sequentially.
Array elements can be stored in consecutive manner in memory.	Linked list elements can be stored at any available place as address of node is stored in previous node.
Insert and delete operation takes more time in array.	Insert and delete operation cannot take more time. It performs operation in fast and in easy way.
Memory is allocated at compile time.	Memory is allocated at run time.
It can be single dimensional, two dimensional or multidimensional.	It can be singly, doubly or circular linked list.
Each array element is independent and does not have a connection with previous element or with its location.	Location or address of element is stored in the link part of previous element or node.
Array elements cannot be added, deleted once it is declared.	The nodes in the linked list can be added and deleted from the list.
In array, elements can be modified easily by identifying the index value.	In linked list, modifying the node is a complex process.

Pointer cannot be used in array. So, it does not require extra space in memory for pointer.	Pointers are used in linked list. Elements are maintained using pointers or links. So, it requires extra memory space for pointers.

Types of Linked List

Following are the types of Linked List

1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List
4. Doubly Circular Linked List

1. Singly Linked List

- Each node has a single link to another node is called Singly Linked List.
- Singly Linked List does not store any pointer any reference to the previous node.
- Each node stores the contents of the node and a reference to the next node in the list.
- In a singly linked list, last node has a pointer which indicates that it is the last node. It requires a reference to the first node to store a single linked list.
- It has two successive nodes linked together in linear way and contains address of the next node to be followed.
- It has successor and predecessor. First node does not have predecessor while last node does not have successor. Last node have successor reference as NULL.
- It has only single link for the next node.
- In this type of linked list, only forward sequential movement is possible, no direct access is allowed.

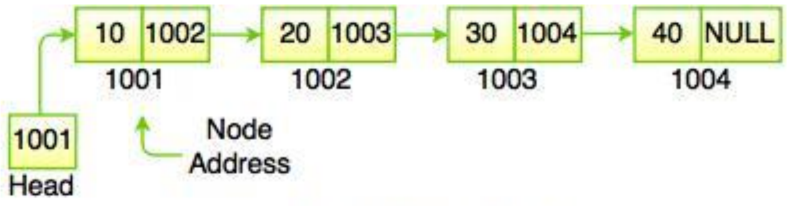


Fig. Singly Linked List

- In the above figure, the address of the first node is always store in a reference node known as Head or Front. Reference part of the last node must be null.

2. Doubly Linked List

- Doubly linked list is a sequence of elements in which every node has link to its previous node and next node.
- Traversing can be done in both directions and displays the contents in the whole list.

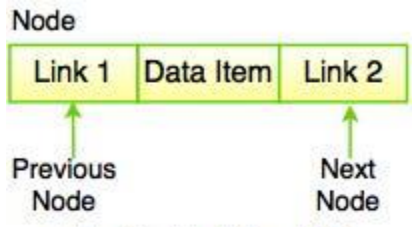


Fig. Doubly Linked List

In the above figure, Link1 field stores the address of the previous node and Link2 field stores the address of the next node. The Data Item field stores the actual value of that node. If we insert a data into the linked list, it will be look like as follows:

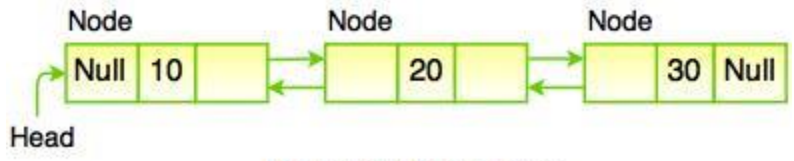


Fig. Doubly Linked List

Important Note:
First node is always pointed by head. In doubly linked list, previous field of the first node is always NULL (it must be NULL) and the next field of the last must be NULL.

In the above figure we see that, doubly linked list contains three fields. In this, link of two nodes allow traversal of the list in either direction. There is no need to traverse the list to find the previous node. We can traverse from head to tail as well as tail to head.

Advantages of Doubly Linked List

- Doubly linked list can be traversed in both forward and backward directions.

- To delete a node in singly linked list, the previous node is required, while in doubly linked list, we can get the previous node using previous pointer.
- It is very convenient than singly linked list. Doubly linked list maintains the links for bidirectional traversing.

Disadvantages of Doubly Linked List

- In doubly linked list, each node requires extra space for previous pointer.
- All operations such as Insert, Delete, Traverse etc. require extra previous pointer to be maintained.

3. Circular Linked List

- Circular linked list is similar to singly linked list. The only difference is that in circular linked list, the last node points to the first node in the list.
- It is a sequence of elements in which every element has link to its next element in the sequence and has a link to the first element in the sequence.

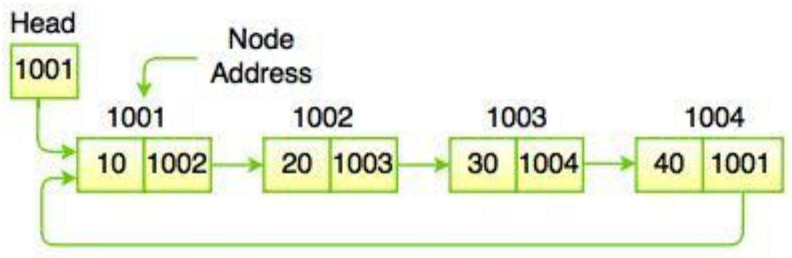


Fig. Circular Linked List

- In the above figure we see that, each node points to its next node in the sequence but the last node points to the first node in the list. The previous element stores the address of the next element and the last element stores the address of the starting element. It forms a circular chain because the element points to each other in a circular way.
- In circular linked list, the memory can be allocated when it is required because it has a dynamic size.
- Circular linked list is used in personal computers, where multiple applications are running. The operating system provides a fixed time slot for all running applications and the running applications are kept in a circular linked list until

all the applications are completed. This is a real life example of circular linked list.

- We can insert elements anywhere in circular linked list, but in the array we cannot insert elements anywhere in the list because it is in the contiguous memory.

4. Doubly Circular Linked List

- Doubly circular linked list is a linked data structure which consists of a set of sequentially linked records called nodes.
- Doubly circular linked list can be conceptualized as two singly linked lists formed from the same data items, but in opposite sequential orders.

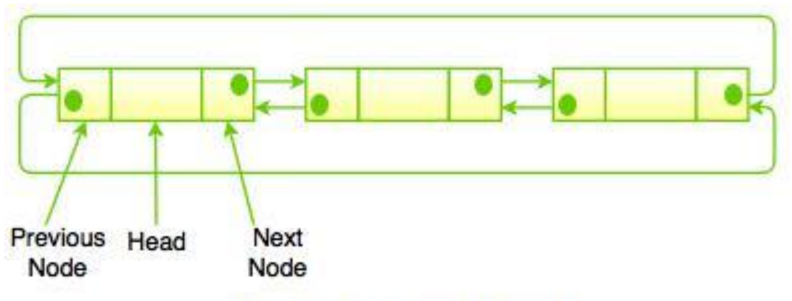


Fig. Doubly Circular Linked List

- The above diagram represents the basic structure of Doubly Circular Linked List. In doubly circular linked list, the previous link of the first node points to the last node and the next link of the last node points to the first node.
- In doubly circular linked list, each node contains two fields called links used to represent references to the previous and the next node in the sequence of nodes.

What are trees?

- 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.

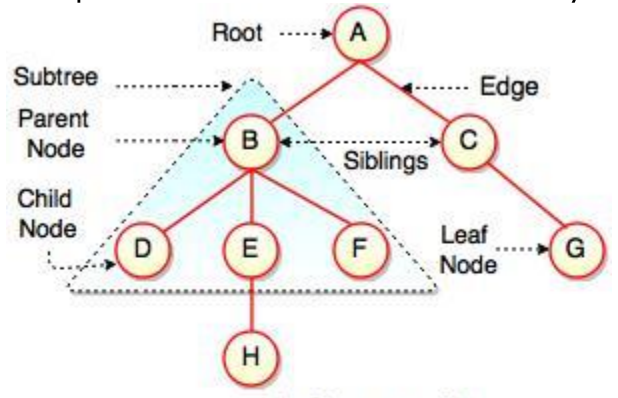


Fig. Structure of Tree

The above figure represents structure of a tree. Tree has 2 subtrees.
A is a parent of B and C.
B is called a child of A and also parent of D, E, F.

Tree is a collection of elements called Nodes, where each node can have arbitrary number of children.

Field	Description
Root	Root is a special node in a tree. The entire tree is referenced through it. It does not have a parent.
Parent Node	Parent node is an immediate predecessor of a node.
Child Node	All immediate successors of a node are its children.
Siblings	Nodes with the same parent are called Siblings.
Path	Path is a number of successive edges from source node to destination node.
Height of Node	Height of a node represents the number of edges on the longest path between that node and a leaf.

Height of Tree	Height of tree represents the height of its root node.
Depth of Node	Depth of a node represents the number of edges from the tree's root node to the node.
Degree of Node	Degree of a node represents a number of children of a node.
Edge	Edge is a connection between one node to another. It is a line between two nodes or a node and a leaf.

In the above figure, D, F, H, G are **leaves**. B and C are **siblings**. Each node excluding a root is connected by a direct edge from exactly one other node parent → children.

Levels of a node

Levels of a node represents the number of connections between the node and the root. It represents generation of a node. If the root node is at level 0, its next node is at level 1, its grand child is at level 2 and so on. Levels of a node can be shown as follows:

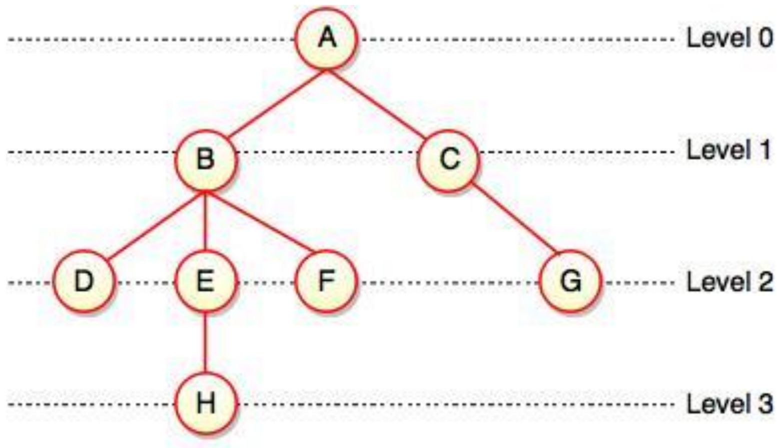


Fig. Levels of Tree

Note:

- If node has no children, it is called **Leaves** or **External Nodes.**

- Nodes which are not leaves, are called **Internal Nodes**. Internal nodes have at least one child.

- A tree can be empty with no nodes or a tree consists of one node called the **Root**.

Height of a Node

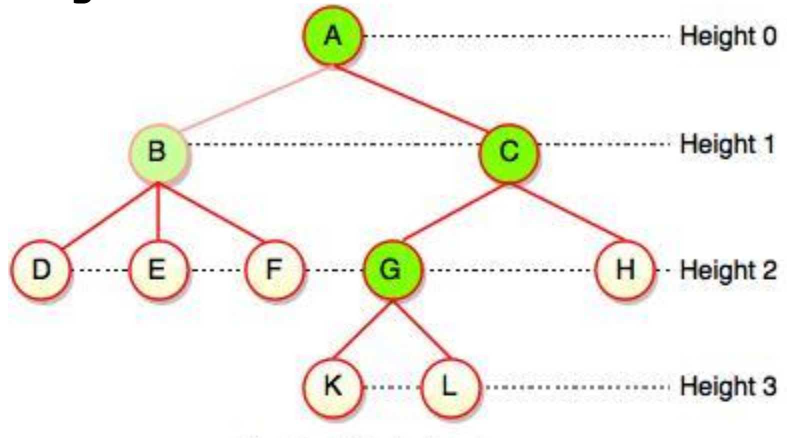


Fig. Height of a Node

As we studied, height of a node is a number of edges on the longest path between that node and a leaf. Each node has height.

In the above figure, A, B, C, D can have height. Leaf cannot have height as there will be no path starting from a leaf. Node A's height is the number of edges of the path to K not to D. And its height is 3.

Note:

- Height of a node defines the longest path from the node to a leaf.

- Path can only be downward.

Depth of a Node

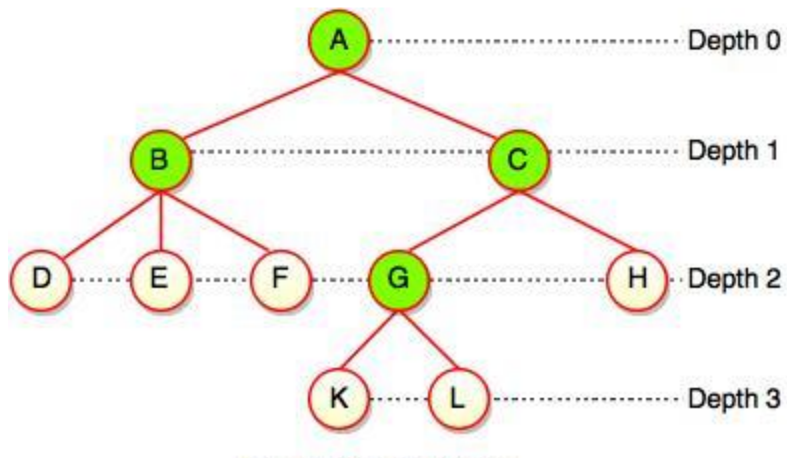


Fig. Depth of a Node

While talking about the height, it locates a node at bottom where for depth, it is located at top which is root level and therefore we call it depth of a node.

In the above figure, Node G's depth is 2. In depth of a node, we just count how many edges between the targeting node & the root and ignoring the directions.

Note: Depth of the root is 0.

Advantages of Tree

- 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 is a special type of data structure. In binary tree, every node can have a maximum of 2 children, which are known as **Left child** and **Right Child**. It is a method of placing and locating the records in a database, especially when all the data is known to be in random access memory (RAM).

Definition:

"A tree in which every node can have maximum of two children is called as Binary Tree."

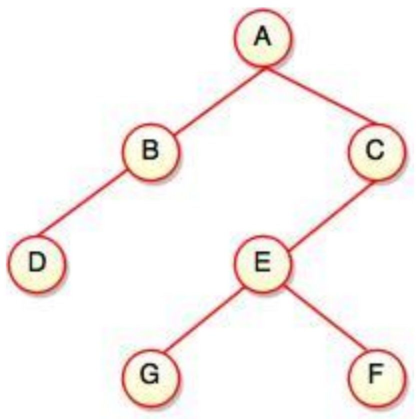


Fig. Binary Tree

The above tree represents binary tree in which node A has two children B and C. Each children have one child namely D and E respectively.

Representation of Binary Tree using Array

Binary tree using array represents a node which is numbered sequentially level by level from left to right. Even empty nodes are numbered.

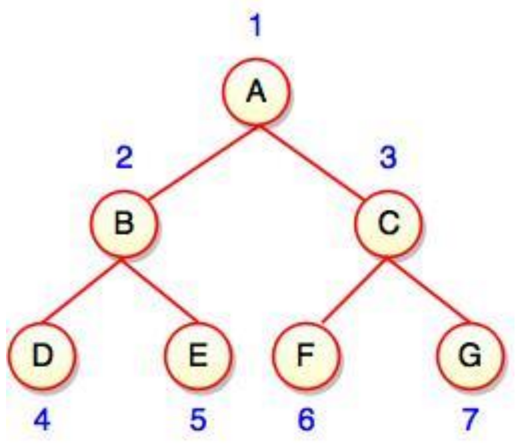


Fig. Binary Tree using Array

Array index is a value in tree nodes and array value gives to the parent node of that particular index or node. Value of the root node index is always -1 as there is no parent for root. When the data item of the tree is sorted in an array, the number appearing against the node will work as indexes of the node in an array.

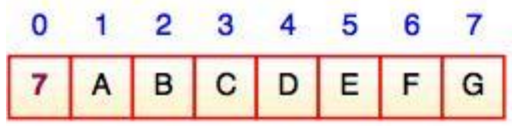


Fig. Location Number of an Array in a Tree

Location number of an array is used to store the size of the tree. The first index of an array that is '0', stores the total number of nodes. All nodes are numbered from left to right level by level from top to bottom. In a tree, each node having an index i is put into the array as its i th element.

The above figure shows how a binary tree is represented as an array. Value '7' is the total number of nodes. If any node does not have any of its child, null value is stored at the corresponding index of the array.

Binary Search Tree

- Binary search tree is a binary tree which has special property called BST.
- BST property is given as follows:
For all nodes A and B,

I. If B belongs to the left subtree of A, the key at B is less than the key at A.

II. If B belongs to the right subtree of A, the key at B is greater than the key at A.

Each node has following attributes:

I. Parent (P), left, right which are pointers to the parent (P), left child and right child respectively.

II. Key defines a key which is stored at the node.

Definition:

"Binary Search Tree is a binary tree where each node contains only smaller values in its left subtree and only larger values in its right subtree."

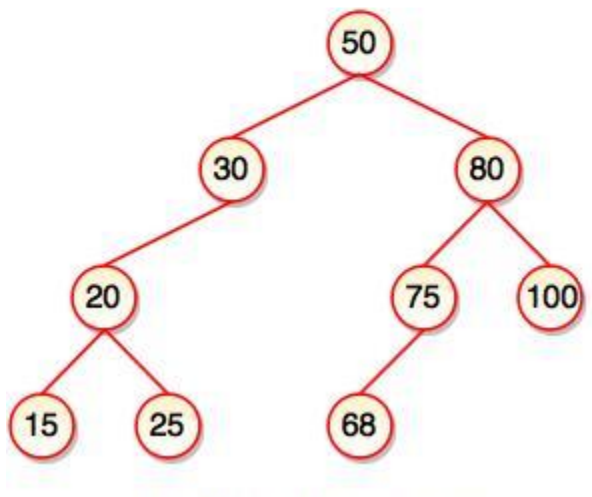


Fig. Binary Search Tree

- The above tree represents binary search tree (BST) where left subtree of every node contains smaller values and right subtree of every node contains larger value.
- Binary Search Tree (BST) is used to enhance the performance of binary tree.
- It focuses on the search operation in binary tree.

Note: Every binary search tree is a binary tree, but all the binary trees need not to be binary search trees.

Binary Search Tree Operations

Following are the operations performed on binary search tree:

1. Insert Operation

- Insert operation is performed with O(log n) time complexity in a binary search tree.
- Insert operation starts from the root node. It is used whenever an element is to be inserted.

The following algorithm shows the insert operation in binary search tree:

Step 1: Create a new node with a value and set its left and right to NULL.

Step 2: Check whether the tree is empty or not.

Step 3: If the tree is empty, set the root to a new node.

Step 4: If the tree is not empty, check whether a value of new node is smaller or larger than the node (here it is a root node).

Step 5: If a new node is smaller than or equal to the node, move to its left child.

Step 6: If a new node is larger than the node, move to its right child.

Step 7: Repeat the process until we reach to a leaf node.

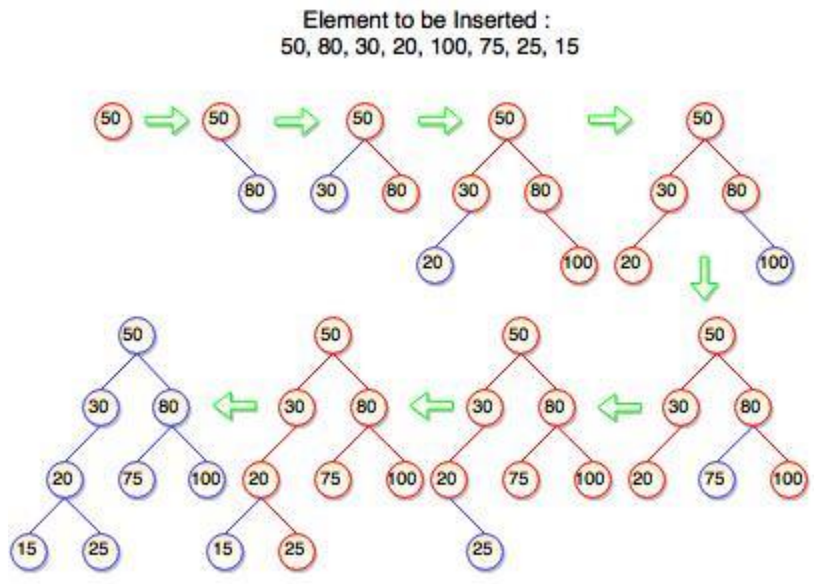


Fig. Insert Operation

The above tree is constructed a binary search tree by inserting the above elements {50, 80, 30, 20, 100, 75, 25, 15}. The diagram represents how the sequence of numbers or elements are inserted into a binary search tree.

2. Search Operation

- Search operation is performed with O(log n) time complexity in a binary search tree.
- This operation starts from the root node. It is used whenever an element is to be searched.

The following algorithm shows the search operation in binary search tree:

Step 1: Read the element from the user .

Step 2: Compare this element with the value of root node in a tree.

Step 3: If element and value are matching, display "Node is Found" and terminate the function.

Step 4: If element and value are not matching, check whether an element is smaller or larger than a node value.

Step 5: If an element is smaller, continue the search operation in left subtree.

Step 6: If an element is larger, continue the search operation in right subtree.

Step 7: Repeat the same process until we found the exact element.

Step 8: If an element with search value is found, display "Element is found" and terminate the function.

Step 9: If we reach to a leaf node and the search value is not match to a leaf node, display "Element is not found" and terminate the function.

Binary Tree Traversal

Binary tree traversing is a process of accessing every node of the tree and exactly once. A tree is defined in a recursive manner. Binary tree traversal also defined recursively.

There are three techniques of traversal:

1. Preorder Traversal
2. Postorder Traversal
3. Inorder Traversal

1. Preorder Traversal

Algorithm for preorder traversal

Step 1 : Start from the Root.

Step 2 : Then, go to the Left Subtree.

Step 3 : Then, go to the Right Subtree.

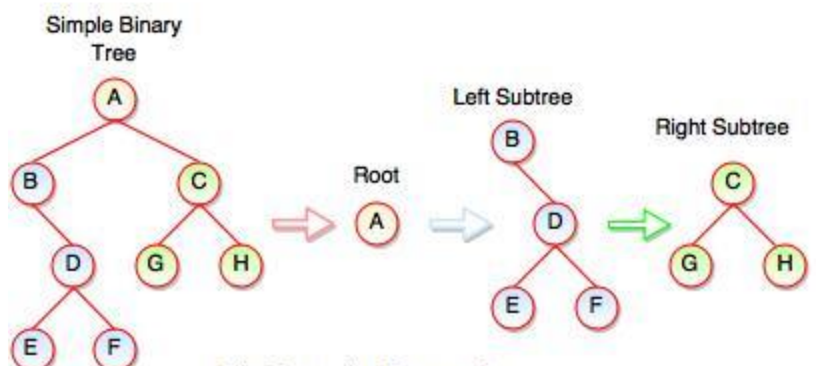


Fig. Preorder Traversal

The above figure represents how preorder traversal actually works.

Following steps can be defined the flow of preorder traversal:

Step 1 : A + B (B + Preorder on D (D + Preorder on E and F)) + C (C + Preorder on G and H)

Step 2 : A + B + D (E + F) + C (G + H)

Step 3 : A + B + D + E + F + C + G + H

Preorder Traversal : A B C D E F G H

2. Postorder Traversal

Algorithm for postorder traversal

Step 1 : Start from the Left Subtree (Last Leaf).

Step 2 : Then, go to the Right Subtree.

Step 3 : Then, go to the Root.

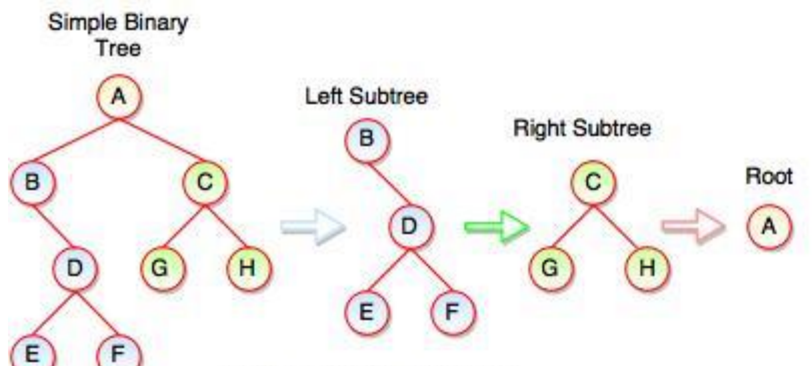


Fig. Postorder Traversal

The above figure represents how postorder traversal actually works.

Following steps can be defined the flow of postorder traversal:

Step 1 : As we know, preorder traversal starts from left subtree (last leaf) ((Postorder on E + Postorder on F) + D + B)) + ((Postorder on G + Postorder on H) + C) + (Root A)

Step 2 : (E + F) + D + B + (G + H) + C + A

Step 3 : E + F + D + B + G + H + C + A

Postorder Traversal : E F D B G H C A

3. Inorder Traversal

Algorithm for inorder traversal

Step 1 : Start from the Left Subtree.

Step 2 : Then, visit the Root.

Step 3 : Then, go to the Right Subtree.

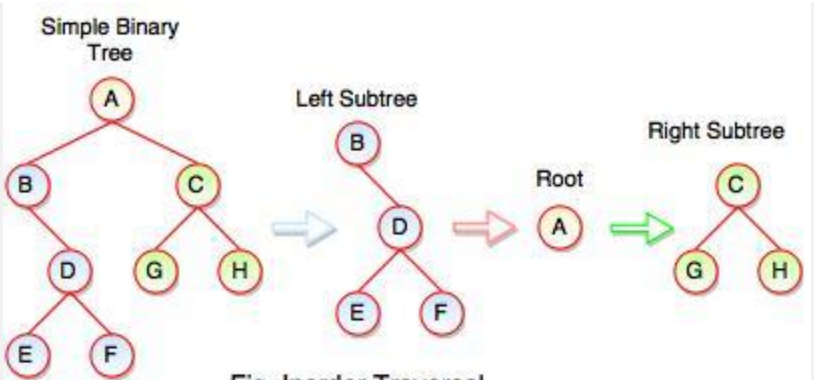


Fig. Inorder Traversal

The above figure represents how inorder traversal actually works.

Following steps can be defined the flow of inorder traversal:

Step 1 : B + (Inorder on E) + D + (Inorder on F) + (Root A) + (Inorder on G) + C (Inorder on H)

Step 2 : B + (E) + D + (F) + A + G + C + H

Step 3 : B + E + D + F + A + G + C + H

Inorder Traversal : B E D F A G C H

A tree is said to be binary tree when,

1. A binary tree has a root node. It may not have any child nodes(0 child nodes, NULL tree).

2. A root node may have one or two child nodes. Each node forms a binary tree itself.

3. The number of child nodes cannot be more than two.

4. It has a unique path from the root to every other node.

There are four types of binary tree:

1. Full Binary Tree
2. Complete Binary Tree
3. Skewed Binary Tree
4. Extended Binary Tree

1. Full Binary Tree

- If each node of binary tree has either two children or no child at all, is said to be a **Full Binary Tree**.
- Full binary tree is also called as **Strictly Binary Tree**.

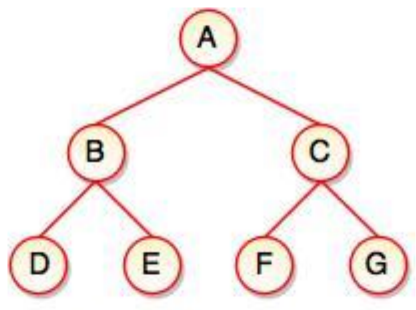


Fig. Full Binary Tree

- Every node in the tree has either 0 or 2 children.
- Full binary tree is used to represent mathematical expressions.

2. Complete Binary Tree

- If all levels of tree are completely filled except the last level and the last level has all keys as left as possible, is said to be a **Complete Binary Tree**.
- Complete binary tree is also called as **Perfect Binary Tree**.

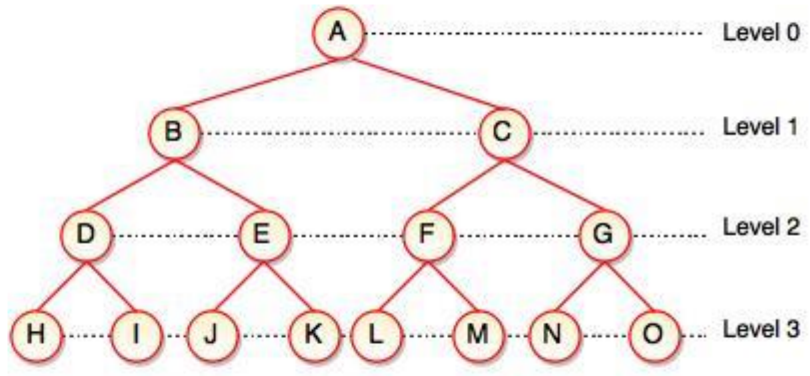


Fig. Complete Binary Tree

- In a complete binary tree, every internal node has exactly two children and all leaf nodes are at same level.
- For example, at Level 2, there must be 2^2 = 4 nodes and at Level 3 there must be 2^3 = 8 nodes.

3. Skewed Binary Tree

- If a tree which is dominated by left child node or right child node, is said to be a **Skewed Binary Tree**.
- In a skewed binary tree, all nodes except one have only one child node. The remaining node has no child.

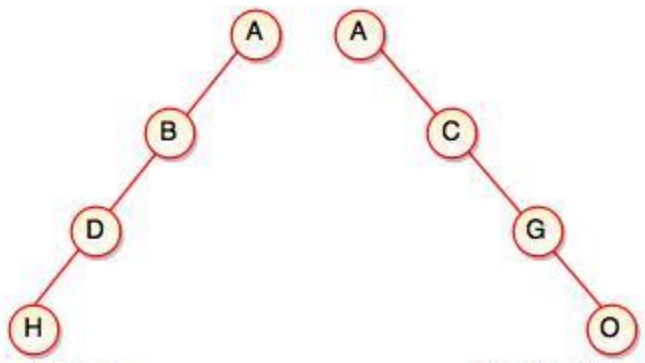


Fig. Left Skewed Binary Tree

Fig. Right Skewed Binary Tree

- In a left skewed tree, most of the nodes have the left child without corresponding right child.
- In a right skewed tree, most of the nodes have the right child without corresponding left child.

4. Extended Binary Tree

- Extended binary tree consists of replacing every null subtree of the original tree with special nodes.
- Empty circle represents internal node and filled circle represents external node.
- The nodes from the original tree are internal nodes and the special nodes are external nodes.
- Every internal node in the extended binary tree has exactly two children and every external node is a leaf. It displays the result which is a **complete binary tree**.

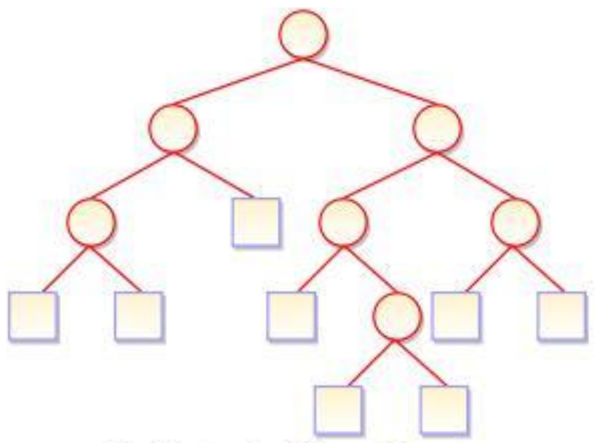


Fig. Extended Binary Tree

AVL Tree

- AVL tree is a height balanced tree.
- It is a self-balancing binary search tree.
- AVL tree is another balanced binary search tree.
- It was invented by **A**delson-**V**elskii and **L**andis.
- AVL trees have a faster retrieval.
- It takes O(logn) time for addition and deletion operation.
- In AVL tree, heights of left and right subtree cannot be more than one for all nodes.

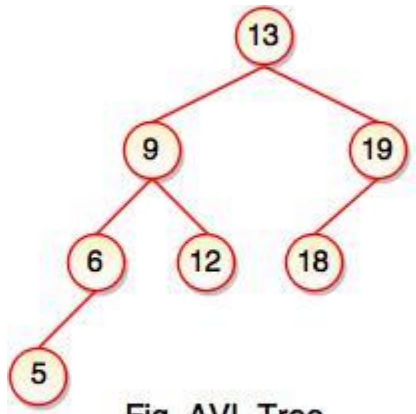
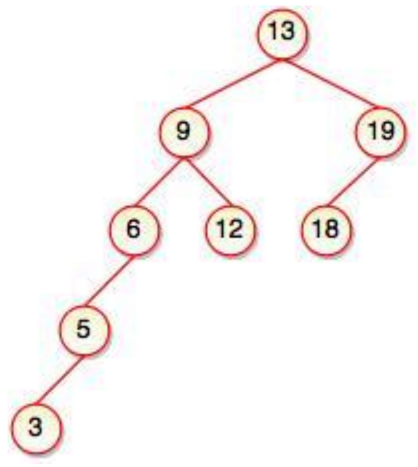


Fig. AVL Tree

- The above tree is AVL tree because the difference between heights of left and right subtrees for every node is less than or equal to 1.



- The above tree is not AVL because the difference between heights of left and right subtrees for 9 and 19 is greater than 1.
- It checks the height of the left and right subtree and assures that the difference is not more than 1. The difference is called balance factor.

What is Threaded Binary Tree??

A binary tree is threaded by making all right child pointers that would normally be null point to the inorder successor of the node (if it exists), and all left child pointers that would normally be null point to the inorder predecessor of the node**.**

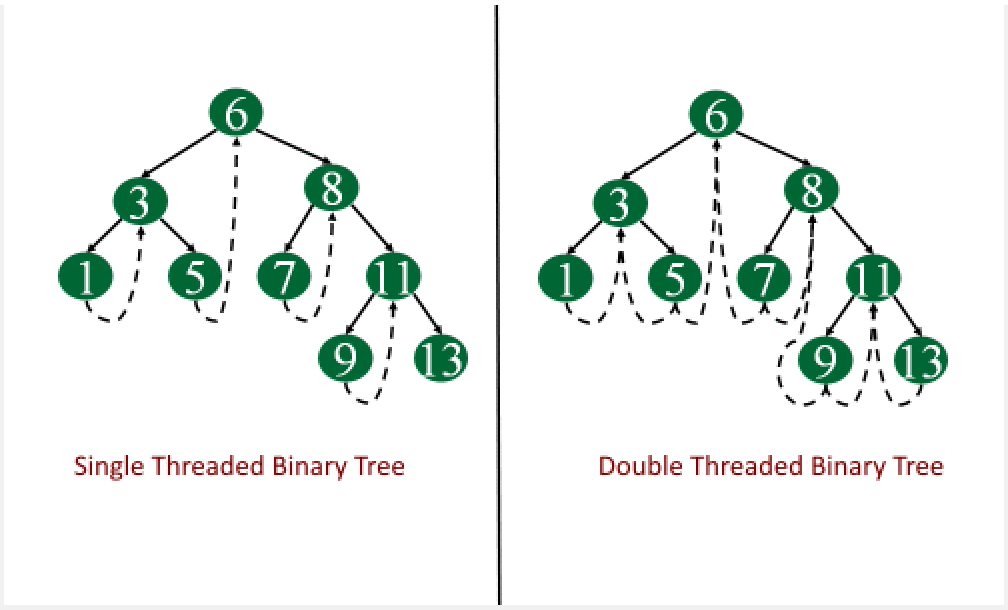
- We have the pointers reference the next node in an inorder traversal; called threads
- We need to know if a pointer is an actual link or a thread, so we keep a boolean for each pointer

Why do we need Threaded Binary Tree?

- Binary trees have a lot of wasted space: the leaf nodes each have 2 null pointers. We can use these pointers to help us in inorder traversals.
- Threaded binary tree makes the tree traversal faster since we do not need stack or recursion for traversal

Types of threaded binary trees:

- **Single Threaded**: each node is threaded towards either the in-order predecessor or successor (left **or**right) means all right null pointers will point to inorder successor **OR** all left null pointers will point to inorder predecessor.
- **Double threaded:** each node is threaded towards both the in-order predecessor and successor (left **and**right) means all right null pointers will point to inorder successor **AND** all left null pointers will point to inorder predecessor.



Single and Double threaded binary tree (1)

In earlier article "Introduction to Threaded Binary Tree" we have seen what is threaded binary tree, types of it and what advantages it has over normal binary tree. In this article we will see the complete implementation of single threaded binary tree.(Click here to read about "double threaded binary tree")

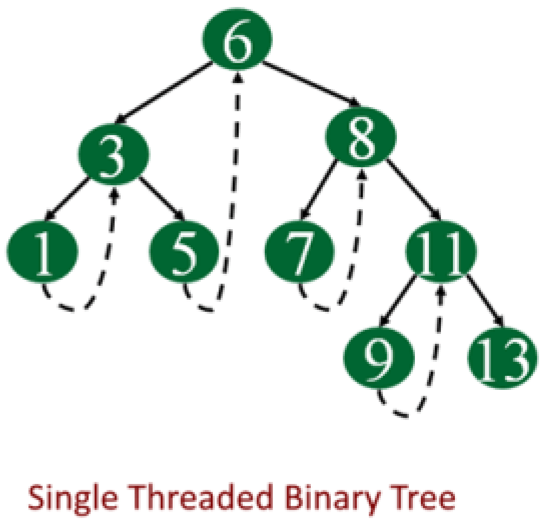
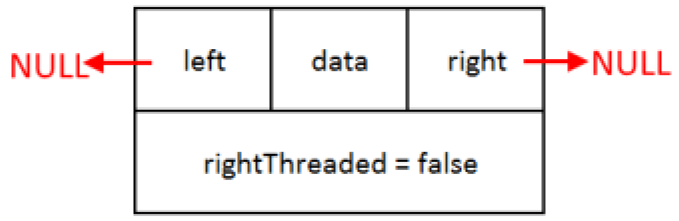


Image Source : http://web.eecs.umich.edu/~akamil/teaching/su02/080802.ppt
Single Threaded: each node is threaded towards either the in-order predecessor or successor (left **or** right) means all right null pointers will point to inorder successor **OR** all left null pointers will point to inorder predecessor.

Implementation:
Let’s see how the Node structure will look like



```
class Node{

    Node left;

    Node right;

    int data;

    boolean rightThread;

    public Node(int data){

        this.data = data;

        rightThread = false;

    }

}
```

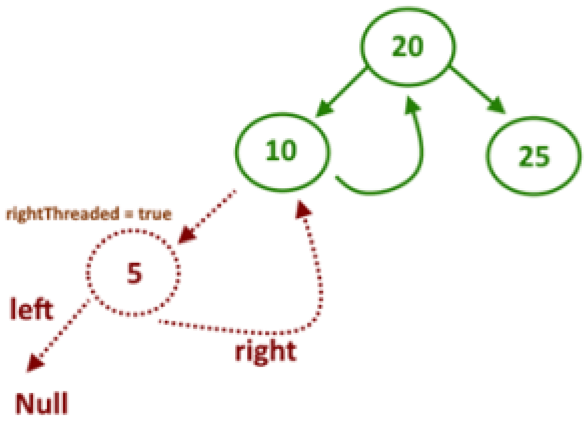
In normal BST node we have left and right references and data but in threaded binary tree we have boolean another field called “rightThreaded”. This field will tell whether node’s right pointer is pointing to its inorder successor, but how, we will see it further.

Operations:
We will discuss two primary operations in single threaded binary tree

1. Insert node into tree
2. Print or traverse the tree.(here we will see the advantage of threaded tree)

Insert():
The insert operation will be quite similar to Insert operation in Binary search tree with few modifications.

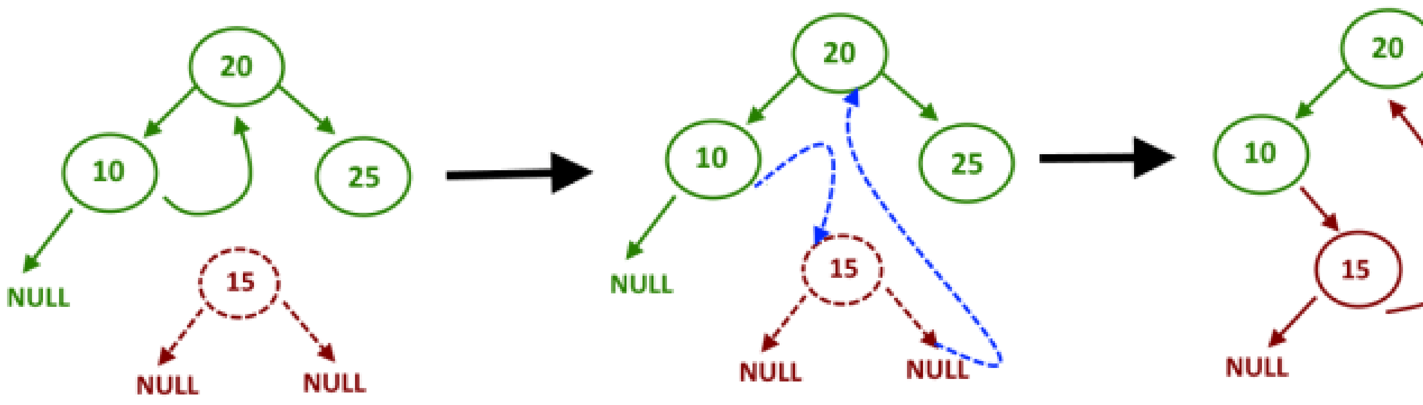
- To insert a node our first task is to find the place to insert the node.
- Take current = root .
- start from the current and compare root.data with n.
- Always keep track of parent node while moving left or right.
- if current.data is greater than n that means we go to the left of the root, if after moving to left, the current = null then we have found the place where we will insert the new node. Add the new node to the left of parent node and make the right pointer points to parent node and rightThread = true for new node.



Inserting node 5

-
- if current.data is smaller than n that means we need to go to the right of the root, while going into the right subtree, check rightThread for current node, means right thread is provided and points to the in order successor, if rightThread = false then and current reaches to null, just insert the new node else if rightThread = true then we need to detach the right pointer (store the reference, new node right reference will be point to it) of current

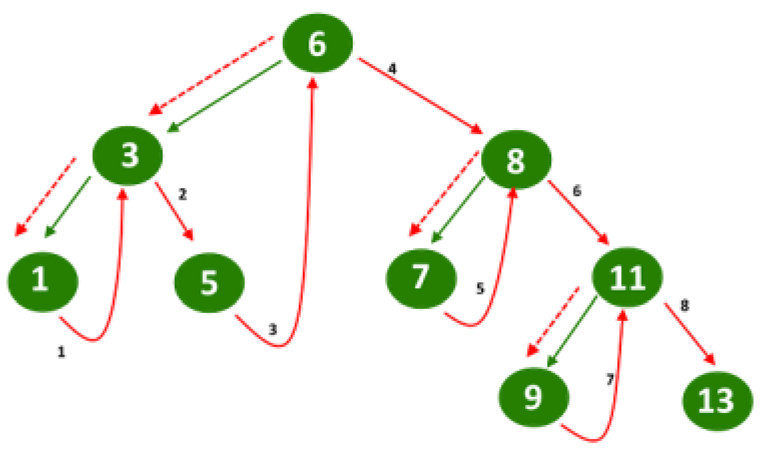
node and make it point to the new node and make the right reference point to stored reference. (See image and code for better understanding)



Inserting Node 15 into threaded binary tree

Traverse():
traversing the threaded binary tree will be quite easy, no need of any recursion or any stack for storing the node. Just go to the left most node and start traversing the tree using right pointer and whenever rightThread = false again go to the left most node in right subtree. (See image and code for better understanding)

Traversal of Single threaded binary tree



Output : 1 3 5 6 7 8 9 11 13

Follow the red arrow, dotted arrow when moving to left most node from the current node and solid arrow when using the right pointer to move it to it's inorder successor.

Complete Code:Run This Code

```
public class SingleThreadedBinaryTree {
```

```
public static Node root;

public void insert(Node root, int id){

    Node newNode = new Node(id);

    Node current = root;

    Node parent = null;

    while(true){

        parent = current;

        if(id<current.data){

            current = current.left;

            if(current==null){

                parent.left = newNode;

                newNode.right = parent;

                newNode.rightThread = true;

                return;

            }

        }else{

            if(current.rightThread==false){

                current = current.right;

                if(current==null){

                    parent.right = newNode;

                    return;

                }

            }else{

                Node temp = current.right;

                current.right = newNode;
```

```java
                    newNode.right = temp;

                    newNode.rightThread=true;

                    return;

                }

            }

        }

    }

    public void print(Node root){

        //first go to most left node

        Node current = leftMostNode(root);

        //now travel using right pointers

        while(current!=null){

            System.out.print(" " + current.data);

            //check if node has a right thread

            if(current.rightThread)

                current = current.right;

            else // else go to left most node in the right subtree

                current = leftMostNode(current.right);

        }

        System.out.println();

    }

    public Node leftMostNode(Node node){

        if(node==null){

            return null;

        }else{
            while(node.left!=null){
                node = node.left;
            }
            return node;
        }
    }
    public static void main(String[] args) {
        root = new Node(100);
        SingleThreadedBinaryTree st=new SingleThreadedBinaryTree();
        st.insert(root,50);
        st.insert(root,25);
        st.insert(root,7);
        st.insert(root,20);
        st.insert(root,75);
        st.insert(root,99);
        st.print(root);
    }
}
class Node{
    Node left;
    Node right;
    int data;
    boolean rightThread;
    public Node(int data){
        this.data = data;
```

```
            rightThread = false;
        }
    }
```

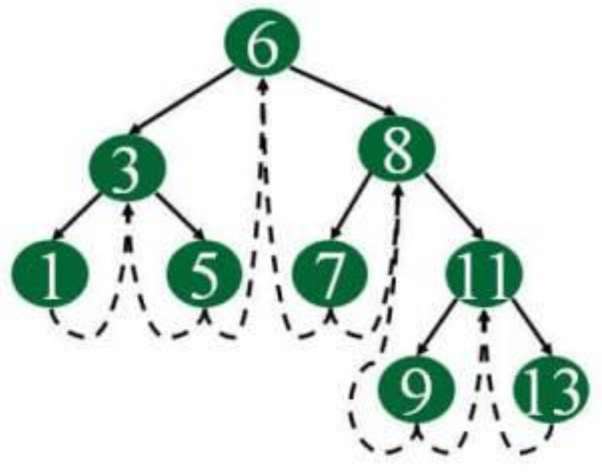
view rawSingleThreadedBinaryTree.java hosted with by GitHub

Run This Code

Output:

```
7 20 25 50 99 75 100
```

In earlier article "Introduction to Threaded Binary Tree" we have seen what is threaded binary tree, types of it and what advantages it has over normal binary tree. In this article we will see the complete implementation of double threaded binary tree. (Click here to read about "single threaded binary tree")



Double Threaded Binary Tree

Image Source : http://web.eecs.umich.edu/~akamil/teaching/su02/080802.ppt

Double threaded: each node is threaded towards both the in-order predecessor and successor (left **and** right) means all right null pointers will point to inorder successor **AND** all left null pointers will point to inorder predecessor.

Implementation:

Let’s see how the Node structure will look like

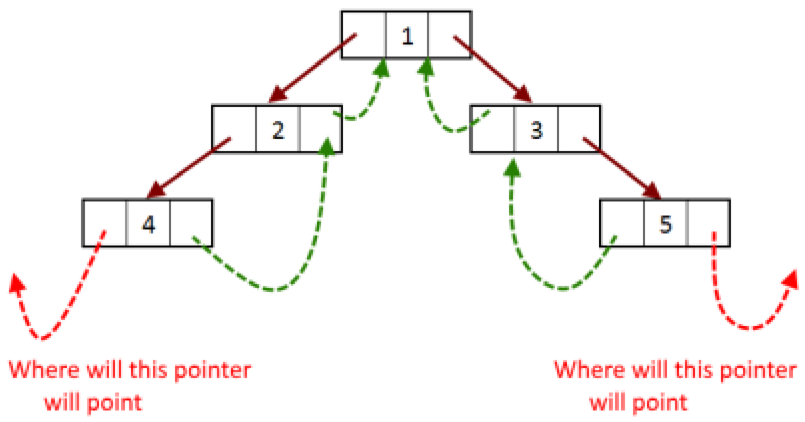
```
class Node {

   int data;

   int leftBit;

   int rightBit;

   Node left;

   Node right;



   public Node(int data) {

      this.data = data;

   }

}
```

If you notice we have two extra fields in the node than regular binary tree node. leftBit and rightBit. Let’s see what these fields represent.

leftBit=0	left reference points to the inorder predecessor
leftBit=1	left reference points to the left child

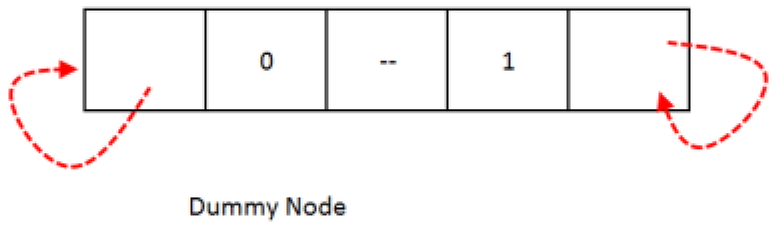
rightBit=0	right reference points to the inorder successor
righBit=1	right reference points to the right child

Let’s see why do we need these fields and why do we need a dummy node when If we try to convert the normal binary tree to threaded binary

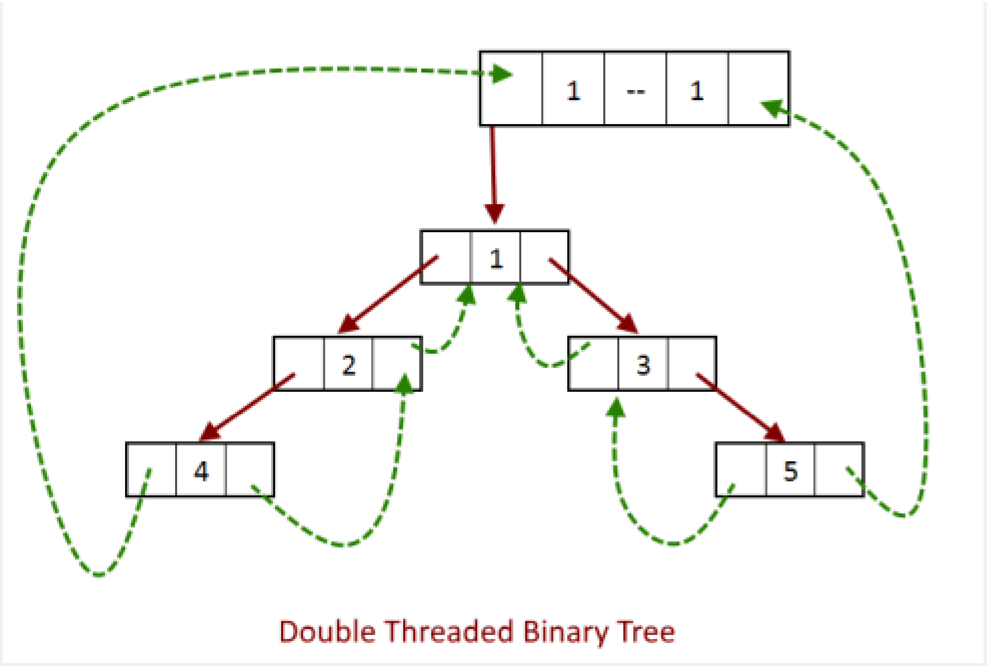

1
2
3
4
5
Where will this pointer will point
Where will this pointer will point

Now if you see the picture above , there are two references left most reference and right most reference pointers has nowhere to point to.

Need of a Dummy Node: As we saw that references left most reference and right most reference pointers has nowhere to point to so we need a dummy node and this node will always present even when tree is empty.
In this dummy node we will put ***rightBit* = *1*** and its right child will point to it self and ***leftBit* = *0***, so we will construct the threaded tree as the left child of dummy node.
Let’s see how the dummy node will look like:



Now we will see how this dummy node will solve our problem of references left most reference and right most reference pointers has nowhere to point to.



Double Threaded binary tree with dummy node

Now we will see the some operations in double threaded binary tree.

Insert():

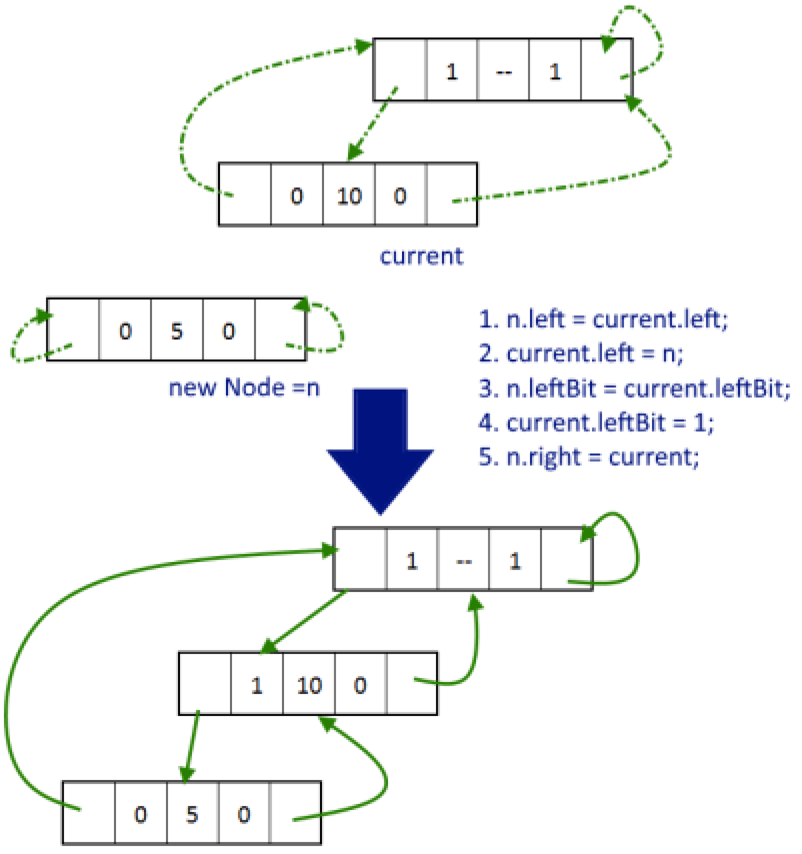
The insert operation will be quite similar to Insert operation in Binary search tree with few modifications.

1. To insert a node our first task is to find the place to insert the node.
2. First check if tree is empty, means tree has just dummy node then then insert the new node into left subtree of the dummy node.
3. If tree is not empty then find the place to insert the node, just like in normal BST.
4. If new node is smaller than or equal to current node then check if ***leftBit =0***, if yes then we have found the place to insert the node, it will be in the left of the subtree and if ***leftBit=1*** then go left.
5. If new node is greater than current node then check if ***rightBit =0***, if yes then we have found the place to insert the node, it will be in the right of the subtree and if ***rightBit=1*** then go right.
6. Repeat step 4 and 5 till the place to be inserted is not found.
7. Once decided where the node will be inserted, next task would be to insert the node. first we will see how the node will be inserted as left child.

```
n.left = current.left;
```

```
current.left = n;

n.leftBit = current.leftBit;

current.leftBit = 1;

n.right = current;
```

see the image below for better understanding



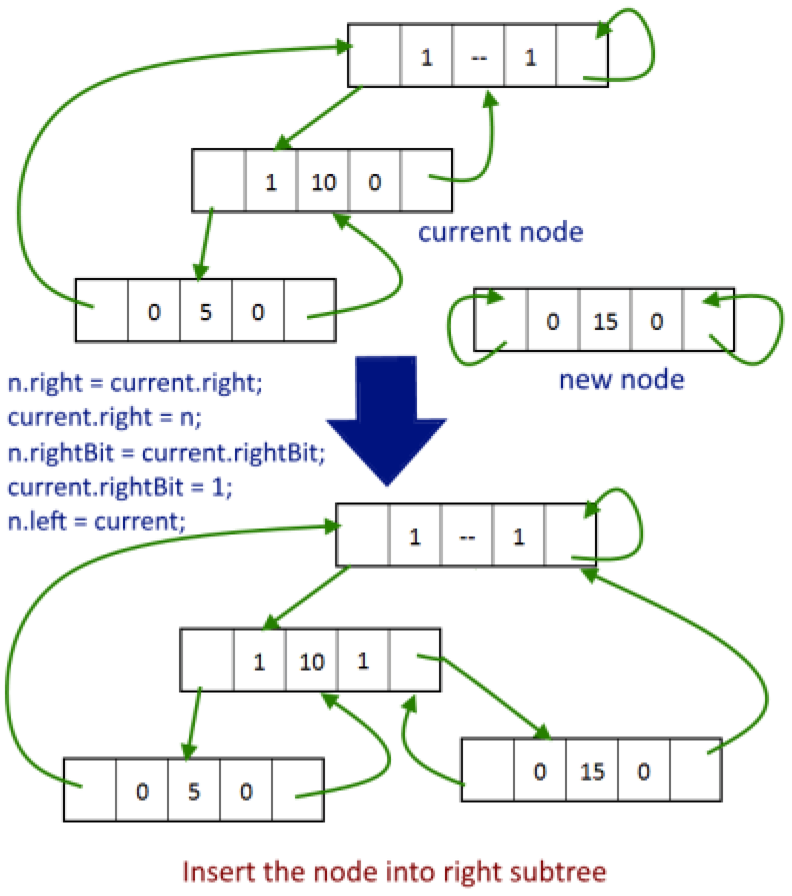
Insert the node in the left sub tree

8. To insert the node as right child.

```
n.right = current.right;
```

```
current.right = n;

n.rightBit = current.rightBit;

current.rightBit = 1;

n.left = current;
```

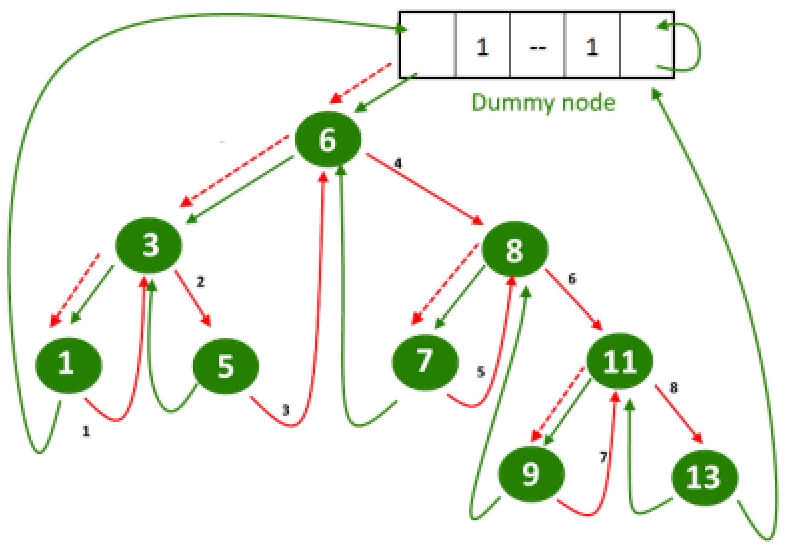
see the image below for better understanding.



Traverse():
Now we will see how to traverse in the double threaded binary tree, we do not need a recursion to do that which means it won’t require stack, it will be done n one single traversal in O(n).
Starting from left most node in the tree, keep traversing the inorder successor and

print it.(click here to read more about inorder successor in a tree). See the image below for more understanding.

Traversal in double threaded binary tree



Output : 1 3 5 6 7 8 9 11 13

Follow the red arrow, dotted arrow when moving to left most node from the current node and solid arrow when using the right pointer to move it to it's inorder successor.

Complete Code: Run This Code

```
public class DoubleThreadedBinaryTree {

    public static Node root;

    public static boolean directionLeft;

    public static boolean directionRight;

    public DoubleThreadedBinaryTree() {

        //create the dummy node

        root = new Node(Integer.MAX_VALUE);

        root.leftBit = 0;

        root.rightBit = 1;

        root.left = root.right = root;

    }
```

```
public void insert(int data) {

        Node n = new Node(data);

        //check if new node is going to be first actual node in the tree.

        if (root == root.left && root.right == root) {

                n.left = root;

                root.left = n;

                n.leftBit = root.leftBit;

                root.leftBit = 1;

                n.right = root;

        } else {

                Node current = root.left;

                while (true) {

                        if (current.data > n.data) {

                                if (current.leftBit == 0) {

                                        //node will be added as left child

                                        directionLeft = true;

                                        directionRight = false;

                                        break;

                                } else {

                                        current = current.left;

                                }

                        } else {

                                if (current.rightBit == 0) {

                                        //node will be added as right child

                                        directionLeft = false;
                                        directionRight = true;
                                        break;
                                    } else {
                                        current = current.right;
                                    }
                                }
                            }
                            if (directionLeft) {
                                //add the node as left child
                                n.left = current.left;
                                current.left = n;
                                n.leftBit = current.leftBit;
                                current.leftBit = 1;
                                n.right = current;
                            } else if (directionRight) {
                                //add the node as right child
                                n.right = current.right;
                                current.right = n;
                                n.rightBit = current.rightBit;
                                current.rightBit = 1;
                                n.left = current;
                            }
                        }
                    }
                    public void inorder() {
```

```
        //start from the left child of the dummy node
        Node current = root.left;
        //go to the left most node
        while (current.leftBit == 1) {
            current = current.left;
        }
        //now keep traversing the next inorder successor and print it
        while (current != root) {
            System.out.print("  " + current.data);
            current = findNextInorder(current);
        }
    }
    public Node findNextInorder(Node current) {
        //if rightBit of current node is 0 means current node does not
        //have right child so use the right pointer to move to its
        // inorder successor.
        if (current.rightBit == 0) {
            return current.right;
        }
        //if //if rightBit of current node is 0 means current node does
        //have right child so go to the left most node in right sub tree.
        current = current.right;
        while (current.leftBit != 0) {
            current = current.left;
        }
        return current;
    }
    public static void main(String args[]){
        DoubleThreadedBinaryTree i = new DoubleThreadedBinaryTree();
        i.insert(10);
        i.insert(12);
        i.insert(15);
        i.insert(2);
        i.insert(13);
        i.insert(1);
        i.insert(4);
        i.inorder();
    }
}
    class Node {
        int data;
        int leftBit;
        int rightBit;
        Node left;
        Node right;
        public Node(int data) {
            this.data = data;
        }
    }
```

Run This Code

Output:

```
1  2  4  10  12  13  15
```

What is Graph?

- Graph is an abstract data type.
- It is a pictorial representation of a set of objects where some pairs of objects are connected by links.
- Graph is used to implement the undirected graph and directed graph concepts from mathematics.
- It represents many real life application. Graphs are used to represent the networks. Network includes path in a city, telephone network etc.
- It is used in social networks like Facebook, LinkedIn etc.

  Graph consists of two following components:

  1. Vertices
  2. Edges
- Graph is a set of vertices (V) and set of edges (E).
- V is a finite number of vertices also called as nodes.
- E is a set of ordered pair of vertices representing edges.
- For example, in Facebook, each person is represented with a vertex or a node. Each node is a structure and contains the information like user id, user name, gender etc.

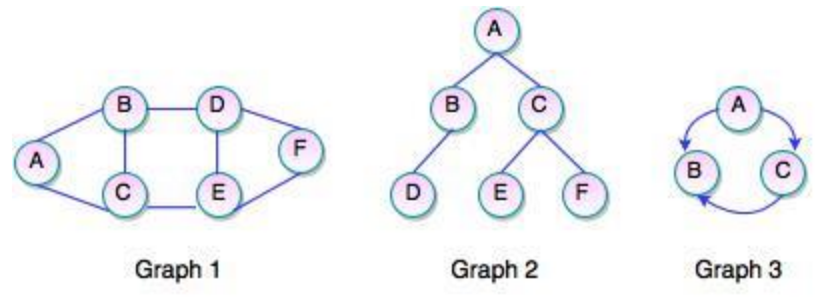


Fig. Graphs

- The above figures represent the graphs. The set representation for each of these graphs are as follows:

  Graph 1:

  V = {A, B, C, D, E, F}
  E = {(A, B), (A, C), (B, C), (B, D), (D, E), (D, F), (E, F)}

  Graph 2:

  V = {A, B, C, D, E, F}
  E = {(A, B), (A, C), (B, D), (C, E), (C, F)}

  Graph 3:

  V = {A, B, C}
  E = {(A, B), (A, C), (C, B)}

Directed Graph

- If a graph contains ordered pair of vertices, is said to be a Directed Graph.
- If an edge is represented using a pair of vertices (V_1, V_2), the edge is said to be directed from V_1 to V_2.
- The first element of the pair V_1 is called the start vertex and the second element of the pair $V_2$is called the end vertex.

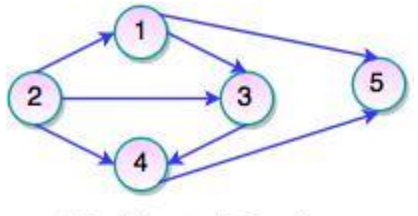


Fig. Directed Graph

Set of Vertices V = {1, 2, 3, 4, 5, 5}

Set of Edges W = {(1, 3), (1, 5), (2, 1), (2, 3), (2, 4), (3, 4), (4, 5)}

Undirected Graph

- If a graph contains unordered pair of vertices, is said to be an Undirected Graph.
- In this graph, pair of vertices represents the same edge.

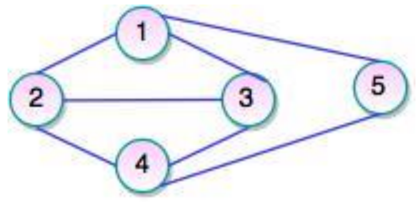


Fig. Undirected Graph

Set of Vertices V = {1, 2, 3, 4, 5}

Set of Edges E = {(1, 2), (1, 3), (1, 5), (2, 1), (2, 3), (2, 4), (3, 4), (4, 5)}

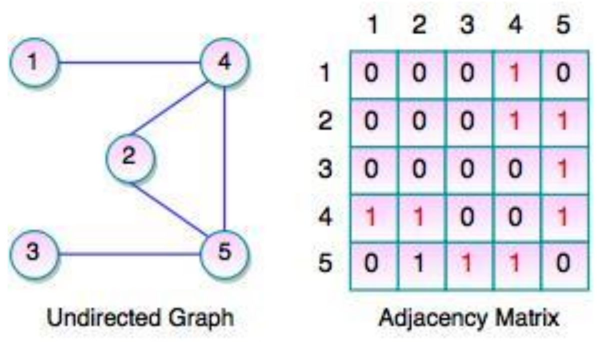
- In an undirected graph, the nodes are connected by undirected arcs.
- It is an edge that has no arrow. Both the ends of an undirected arc are equivalent, there is no head or tail.

Representation of Graphs

Adjacency Matrix

- Adjacency matrix is a way to represent a graph.
- It shows which nodes are adjacent to one another.

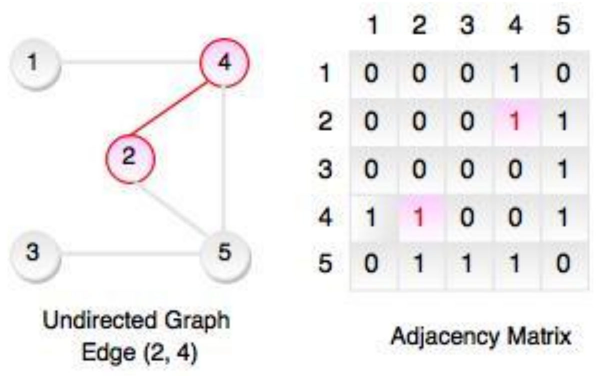
- Graph is represented using a square matrix.
- **Graph can be divided into two categories:**
  a. Sparse Graph
  b. Dense Graph
  a. Sparse graph contains less number of edges.
  b. Dense graph contains number of edges as compared to sparse graph.
- Adjacency matrix is best for dense graph, but for sparse graph, it is not required.
- Adjacency matrix is good solution for dense graph which implies having constant number of vertices.
- Adjacency matrix of an undirected graph is always a symmetric matrix which means an edge (i, j) implies the edge (j, i).



	1	2	3	4	5
1	0	0	0	1	0
2	0	0	0	1	1
3	0	0	0	0	1
4	1	1	0	0	1
5	0	1	1	1	0

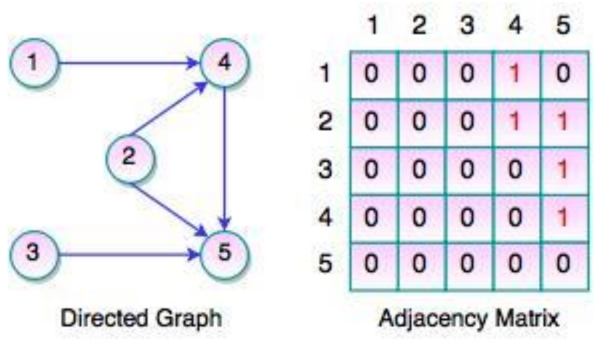
Fig. Adjacency Matrix Representation of Undirected Graph

- The above graph represents undirected graph with the adjacency matrix representation. It shows adjacency matrix of undirected graph is symmetric. If there is an edge (2, 4), there is also an edge (4, 2).



	1	2	3	4	5
1	0	0	0	1	0
2	0	0	0	1	1
3	0	0	0	0	1
4	1	1	0	0	1
5	0	1	1	1	0

- Adjacency matrix of a directed graph is never symmetric adj[i][j] = 1, indicated a directed edge from vertex i to vertex j.



	1	2	3	4	5
1	0	0	0	1	0
2	0	0	0	1	1
3	0	0	0	0	1
4	0	0	0	0	1
5	0	0	0	0	0

Fig. Adjacency Matrix Representation of Directed Graph

- The above graph represents directed graph with the adjacency matrix representation. It shows adjacency matrix of directed graph which is never symmetric. If there is an edge (2, 4), there is not an edge (4, 2). It indicates direct edge from vertex i to vertex j.

Advantages of Adjacency Matrix

- Adjacency matrix representation of graph is very simple to implement.
- Adding or removing time of an edge can be done in O(1) time. Same time is required to check, if there is an edge between two vertices.
- It is very convenient and simple to program.

Disadvantages of Adjacency Matrix

- It consumes huge amount of memory for storing big graphs.
- It requires huge efforts for adding or removing a vertex. If you are constructing a graph in dynamic structure, adjacency matrix is quite slow for big graphs.

Adjacency List

- Adjacency list is another representation of graphs.
- It is a collection of unordered list, used to represent a finite graphs.
- Each list describes the set of neighbors of a vertex in the graph.
- Adjacency list requires less amount of memory.
- For every vertex, adjacency list stores a list of vertices, which are adjacent to the current one.

- In adjacency list, an array of linked list is used. Size of the array is equal to the number of vertices.

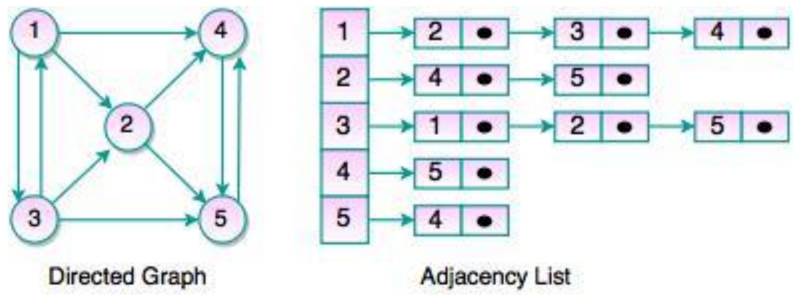


Fig. Adjacency List Representation of Directed Graph

- In adjacency list, an entry array[i] represents the linked list of vertices adjacent to the ithvertex.
- Adjacency list allows to store the graph in more compact form than adjacency matrix.
- It allows to get the list of adjacent vertices in O(1) time.

Disadvantages of Adjacency List

- It is not easy for adding or removing an edge to/from adjacent list.
- It does not allow to make an efficient implementation, if dynamically change of vertices number is required.

Important Note:

Vertex: Each node of the graph is represented as a vertex.
Edge: It represents a path between two vertices or a line between two vertices.
Path: It represents a sequence of edges between the two vertices.
Adjacency: If two nodes or vertices are connected to each other through an edge, it is said to be an adjacency.

Graph Traversal

- Graph traversal is a process of checking or updating each vertex in a graph.
- It is also known as Graph Search.
- Graph traversal means visiting each and exactly one node.
- Tree traversal is a special case of graph traversal.

There are two techniques used in graph traversal:

1. Depth First Search
2. Breadth First Search

1. Depth First Search

- Depth first search (DFS) is used for traversing a finite graph.
- DFS traverses the depth of any particular path before exploring its breadth.
- It explores one subtree before returning to the current node and then exploring the other subtree.
- DFS uses stack instead of queue.
- It traverses a graph in a depth-ward motion and gets the next vertex to start a search when a dead end occurs in any iteration.

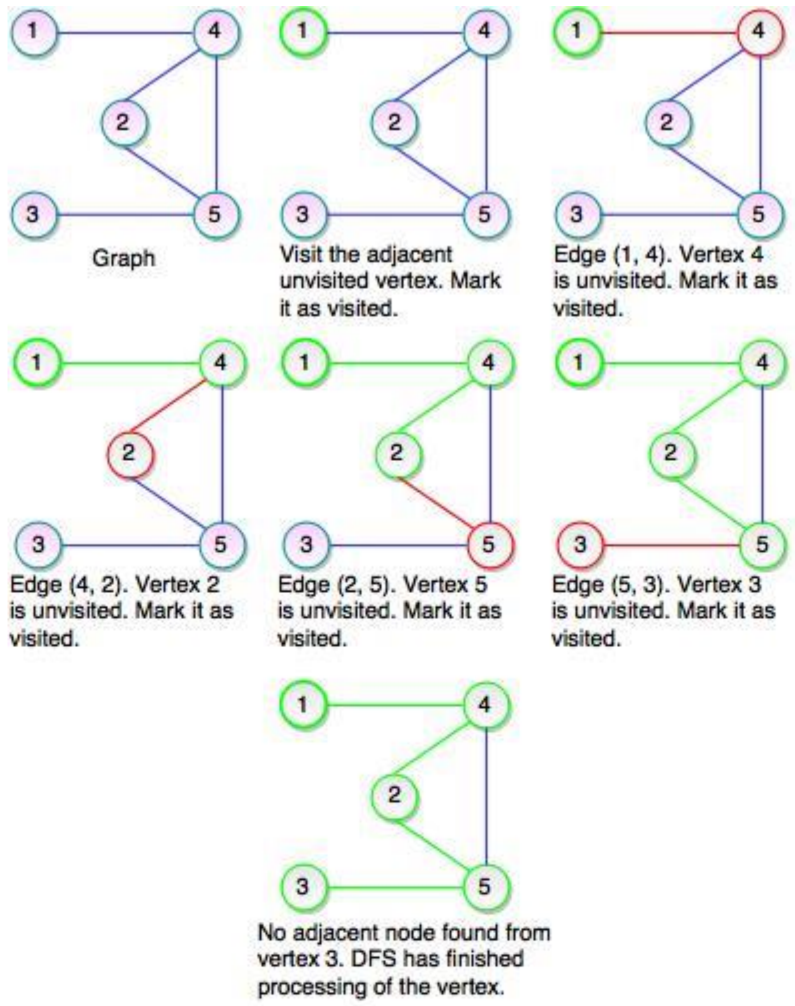

Graph
Visit the adjacent unvisited vertex. Mark it as visited.
Edge (1, 4). Vertex 4 is unvisited. Mark it as visited.
Edge (4, 2). Vertex 2 is unvisited. Mark it as visited.
Edge (2, 5). Vertex 5 is unvisited. Mark it as visited.
Edge (5, 3). Vertex 3 is unvisited. Mark it as visited.
No adjacent node found from vertex 3. DFS has finished processing of the vertex.

Fig. Depth First Search (DFS)

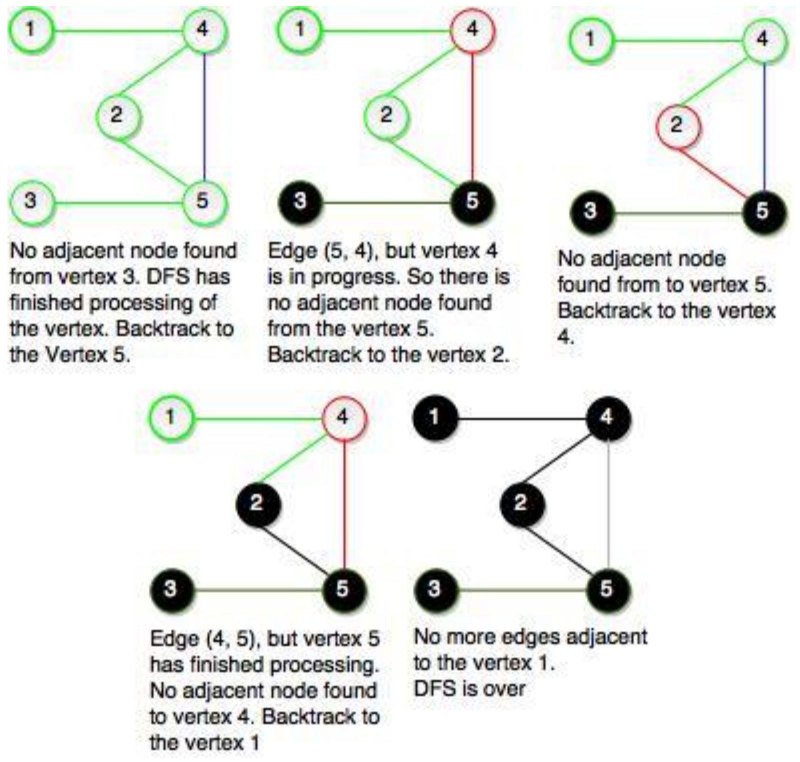

1
4
2
3
5
No adjacent node found from vertex 3. DFS has finished processing of the vertex. Backtrack to the Vertex 5.
Edge (5, 4), but vertex 4 is in progress. So there is no adjacent node found from the vertex 5. Backtrack to the vertex 2.
No adjacent node found from to vertex 5. Backtrack to the vertex 4.
Edge (4, 5), but vertex 5 has finished processing. No adjacent node found to vertex 4. Backtrack to the vertex 1
No more edges adjacent to the vertex 1. DFS is over

Fig. Depth First Search (DFS) Recursive

- As you see both the figures, DFS does not go though all the edges. The tree contains all the vertices of the graph if it is connected to the nodes and is called as Graph Spanning Tree. Graph spanning tree is exactly corresponds to the recursive calls of DFS.
- If a graph is disconnected then DFS will not be able to visit all of its vertices.
- DFS will pop up all the vertices from the stack which do not have adjacent nodes. The process is going on until we find a node that has unvisited adjacent node and if there is no more adjacent node, DFS is over.

2. Breadth First Search

- Breadth first search is used for traversing a finite graph.
- It visits the neighbor vertices before visiting the child vertices.
- BFS uses a queue for search process and gets the next vertex to start a search when a dead end occurs in any iteration.
- It traverses a graph in a breadth-ward motion.
- It is used to find the shortest path from one vertex to another.

- The main purpose of BFS is to traverse the graph as close as possible to the root node.
- BFS is a different approach for traversing the graph nodes.

What is Hashing?

- 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**.
- It is a technique to convert a range of key values into a range of indexes of an array.
- It is used to facilitate the next level searching method when compared with the linear or binary search.
- Hashing allows to update and retrieve any data entry in a constant time O(1).
- Constant time O(1) means the operation does not depend on the size of the data.
- Hashing is used with a database to enable items to be retrieved more quickly.
- It is used in the encryption and decryption of digital signatures.

What is 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.

What is Hash Table?

- 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.

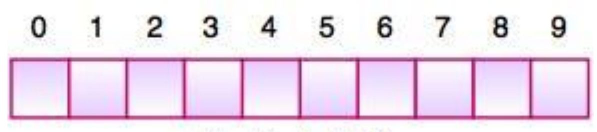


Fig. Hash Table

- The above figure shows the hash table with the size of n = 10. Each position of the hash table is called as **Slot**. In the above hash table, there are n slots in the table, names = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. Slot 0, slot 1, slot 2 and so on. Hash table contains no items, so every slot is empty.
- As we know the mapping between an item and the slot where item belongs in the hash table is called the hash function. The hash function takes any item in the collection and returns an integer in the range of slot names between 0 to n-1.
- Suppose we have integer items {26, 70, 18, 31, 54, 93}. One common method of determining a hash key is the division method of hashing and the formula is :

  Hash Key = Key Value % Number of Slots in the Table

- Division method or reminder method takes an item and divides it by the table size and returns the remainder as its hash value.

Data Item	Value % No. of Slots	Hash Value
26	26 % 10 = 6	6
70	70 % 10 = 0	0
18	18 % 10 = 8	8

31	31 % 10 = 1	1
54	54 % 10 = 4	4
93	93 % 10 = 3	3

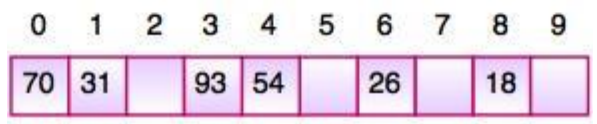


Fig. Hash Table

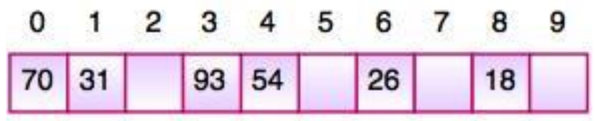
- After computing the hash values, we can insert each item into the hash table at the designated position as shown in the above figure. In the hash table, 6 of the 10 slots are occupied, it is referred to as the load factor and denoted by, λ = No. of items / table size. For example , λ = 6/10.
- It is easy to search for an item using hash function where it computes the slot name for the item and then checks the hash table to see if it is present.
- Constant amount of time O(1) is required to compute the hash value and index of the hash table at that location.

Linear Probing

- Take the above example, if we insert next item 40 in our collection, it would have a hash value of 0 (40 % 10 = 0). But 70 also had a hash value of 0, it becomes a problem. This problem is called as **Collision** or **Clash**. Collision creates a problem for hashing technique.
- **Linear probing is used for resolving the collisions in hash table**, data structures for maintaining a collection of key-value pairs.
- Linear probing was invented by Gene Amdahl, Elaine M. McGraw and Arthur Samuel in 1954 and analyzed by Donald Knuth in 1963.
- It is a component of open addressing scheme for using a hash table to solve the dictionary problem.
- The simplest method is called Linear Probing. Formula to compute linear probing is:

P = (1 + P) % (MOD) Table_size

For example,



0	1	2	3	4	5	6	7	8	9
70	31		93	54		26		18	

Fig. Hash Table

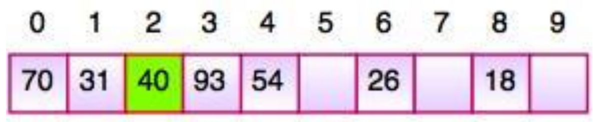
If we insert next item 40 in our collection, it would have a hash value of 0 (40 % 10 = 0). But 70 also had a hash value of 0, it becomes a problem.

Linear probing solves this problem:

P = H(40)
44 % 10 = 0
Position 0 is occupied by 70. so we look elsewhere for a position to store 40.

Using Linear Probing:
P= (P + 1) % table-size
0 + 1 % 10 = 1
But, position 1 is occupied by 31, so we look elsewhere for a position to store 40.

Using linear probing, we try next position : 1 + 1 % 10 = 2
Position 2 is empty, so 40 is inserted there.



0	1	2	3	4	5	6	7	8	9
70	31	40	93	54		26		18	

Fig. Hash Table

What is File?

File is a collection of records related to each other. The file size is limited by the size of memory and storage medium.

There are two important features of file:

1. File Activity
2. File Volatility

File activity specifies percent of actual records which proceed in a single run.

File volatility addresses the properties of record changes. It helps to increase the efficiency of disk design than tape.

File Organization

File organization ensures that records are available for processing. It is used to determine an efficient file organization for each base relation.

For example, if we want to retrieve employee records in alphabetical order of name. Sorting the file by employee name is a good file organization. However, if we want to retrieve all employees whose marks are in a certain range, a file is ordered by employee name would not be a good file organization.

Types of File Organization

There are three types of organizing the file:

1. Sequential access file organization
2. Direct access file organization
3. Indexed sequential access file organization

1. Sequential access file organization

- Storing and sorting in contiguous block within files on tape or disk is called as **sequential access file organization**.

- In sequential access file organization, all records are stored in a sequential order. The records are arranged in the ascending or descending order of a key field.
- Sequential file search starts from the beginning of the file and the records can be added at the end of the file.
- In sequential file, it is not possible to add a record in the middle of the file without rewriting the file.

Advantages of sequential file

- It is simple to program and easy to design.
- Sequential file is best use if storage space.

Disadvantages of sequential file

- Sequential file is time consuming process.
- It has high data redundancy.
- Random searching is not possible.

2. Direct access file organization

- Direct access file is also known as random access or relative file organization.
- In direct access file, all records are stored in direct access storage device (DASD), such as hard disk. The records are randomly placed throughout the file.
- The records does not need to be in sequence because they are updated directly and rewritten back in the same location.
- This file organization is useful for immediate access to large amount of information. It is used in accessing large databases.
- It is also called as hashing.

Advantages of direct access file organization

- Direct access file helps in online transaction processing system (OLTP) like online railway reservation system.
- In direct access file, sorting of the records are not required.
- It accesses the desired records immediately.
- It updates several files quickly.
- It has better control over record allocation.

Disadvantages of direct access file organization

- Direct access file does not provide back up facility.

- It is expensive.
- It has less storage space as compared to sequential file.

3. Indexed sequential access file organization

- Indexed sequential access file combines both sequential file and direct access file organization.
- In indexed sequential access file, records are stored randomly on a direct access device such as magnetic disk by a primary key.
- This file have multiple keys. These keys can be alphanumeric in which the records are ordered is called primary key.
- The data can be access either sequentially or randomly using the index. The index is stored in a file and read into memory when the file is opened.

Advantages of Indexed sequential access file organization

- In indexed sequential access file, sequential file and random file access is possible.
- It accesses the records very fast if the index table is properly organized.
- The records can be inserted in the middle of the file.
- It provides quick access for sequential and direct processing.
- It reduces the degree of the sequential search.

Disadvantages of Indexed sequential access file organization

- Indexed sequential access file requires unique keys and periodic reorganization.
- Indexed sequential access file takes longer time to search the index for the data access or retrieval.
- It requires more storage space.
- It is expensive because it requires special software.
- It is less efficient in the use of storage space as compared to other file organizations.

What is Sorting?

Sorting is a process of ordering or placing a list of elements from a collection in some kind of order. It is nothing but storage of data in sorted order. Sorting can be done in ascending and descending order. It arranges the data in a sequence which makes searching easier.

For example, suppose we have a record of employee. It has following data:

Employee No.
Employee Name
Employee Salary
Department Name

Here, employee no. can be takes as key for sorting the records in ascending or descending order. Now, we have to search a Employee with employee no. 116, so we don't require to search the complete record, simply we can search between the Employees with employee no. 100 to 120.

Sorting Techniques

Sorting technique depends on the situation. It depends on two parameters.

1. Execution time of program that means time taken for execution of program.
2. Space that means space taken by the program.

Sorting techniques are differentiated by their efficiency and space requirements.

Sorting can be performed using several techniques or methods, as follows:

1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Quick Sort
5. Heap Sort

1. Bubble Sort

- Bubble sort is a type of sorting.

- It is used for sorting 'n' (number of items) elements.
- It compares all the elements one by one and sorts them based on their values.

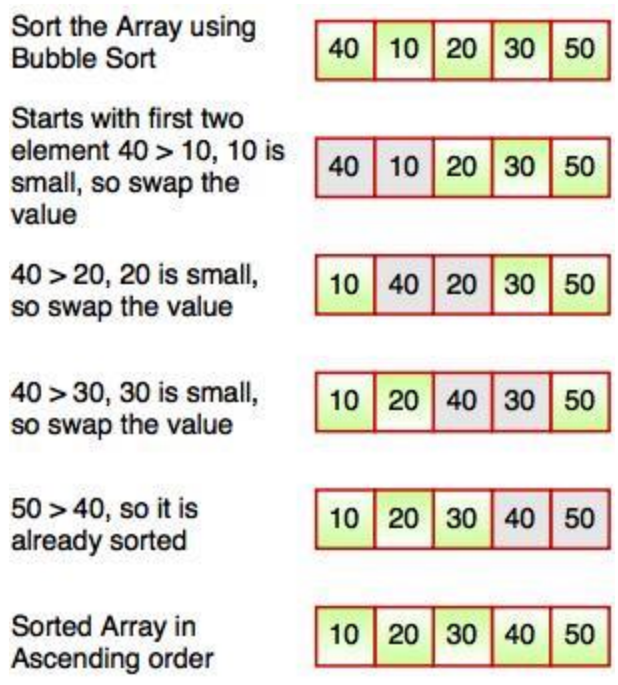

Sort the Array using Bubble Sort
40 10 20 30 50
Starts with first two element 40 > 10, 10 is small, so swap the value
40 10 20 30 50
40 > 20, 20 is small, so swap the value
10 40 20 30 50
40 > 30, 30 is small, so swap the value
10 20 40 30 50
50 > 40, so it is already sorted
10 20 30 40 50
Sorted Array in Ascending order
10 20 30 40 50

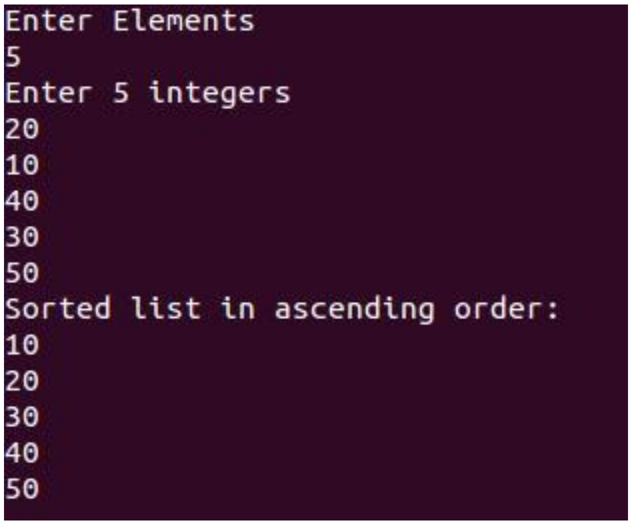
Fig. Working of Bubble Sort

- The above diagram represents how bubble sort actually works. This sort takes O(n^2) time. It starts with the first two elements and sorts them in ascending order.
- Bubble sort starts with first two elements. It compares the element to check which one is greater.
- In the above diagram, element 40 is greater than 10, so these values must be swapped. This operation continues until the array is sorted in ascending order.

Example: Program for Bubble Sort

```
#include <stdio.h>
void bubble_sort(long [], long);

int main()
{
  long array[100], n, c, d, swap;
  printf("Enter Elements\n");
  scanf("%ld", &n);
  printf("Enter %ld integers\n", n);
  for (c = 0; c < n; c++)
    scanf("%ld", &array[c]);
  bubble_sort(array, n);
  printf("Sorted list in ascending order:\n");
  for ( c = 0 ; c < n ; c++ )
     printf("%ld\n", array[c]);
  return 0;
}
void bubble_sort(long list[], long n)
{
  long c, d, t;
  for (c = 0 ; c < ( n - 1 ); c++)
  {
    for (d = 0 ; d < n - c - 1; d++)
    {
      if (list[d] > list[d+1])
      {
        /* Swapping */
        t         = list[d];
        list[d]   = list[d+1];
        list[d+1] = t;
      }
    }
  }
}
```

Output:


```
Enter Elements
5
Enter 5 integers
20
10
40
30
50
Sorted list in ascending order:
10
20
30
40
50
```

2. Insertion Sort

- Insertion sort is a simple sorting algorithm.
- This sorting method sorts the array by shifting elements one by one.
- It builds the final sorted array one item at a time.
- Insertion sort has one of the simplest implementation.
- This sort is efficient for smaller data sets but it is insufficient for larger lists.
- It has less space complexity like bubble sort.
- It requires single additional memory space.
- Insertion sort does not change the relative order of elements with equal keys because it is stable.

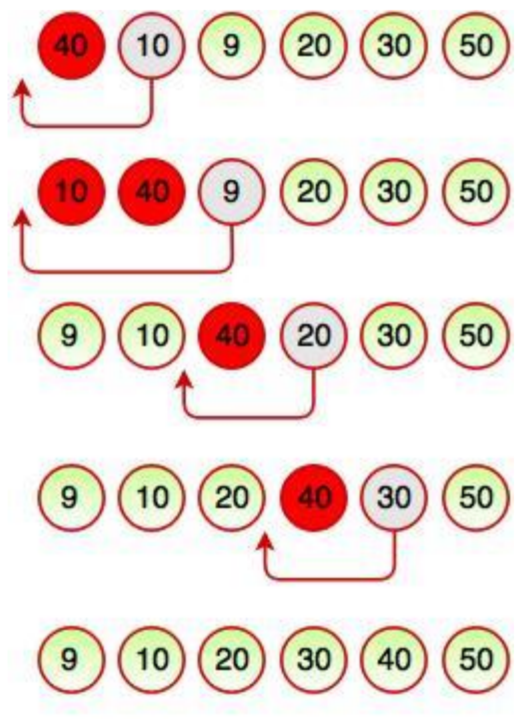


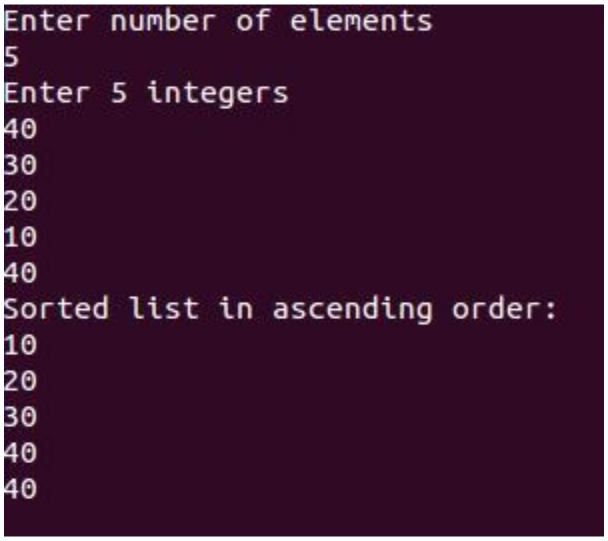
Fig. Working of Insertion Sort

- The above diagram represents how insertion sort works. Insertion sort works like the way we sort playing cards in our hands. It always starts with the second element as key. The key is compared with the elements ahead of it and is put it in the right place.
- In the above figure, 40 has nothing before it. Element 10 is compared to 40 and is inserted before 40. Element 9 is smaller than 40 and 10, so it is inserted before 10 and this operation continues until the array is sorted in ascending order.

Example: Program for Insertion Sort

```
#include <stdio.h>

int main()
{
  int n, array[1000], c, d, t;
  printf("Enter number of elements\n");
  scanf("%d", &n);
  printf("Enter %d integers\n", n);
  for (c = 0; c < n; c++)
  {
    scanf("%d", &array[c]);
  }
  for (c = 1 ; c <= n - 1; c++)
  {
    d = c;
    while ( d > 0 && array[d] < array[d-1])
    {
        t            = array[d];
        array[d]    = array[d-1];
        array[d-1] = t;
        d--;
    }
  }
  printf("Sorted list in ascending order:\n");
  for (c = 0; c <= n - 1; c++)
  {
     printf("%d\n", array[c]);
  }
  return 0;
}
```

Output:



```
Enter number of elements
5
Enter 5 integers
40
30
20
10
40
Sorted list in ascending order:
10
20
30
40
40
```

•

Selection Sort

- Selection sort is a simple sorting algorithm which finds the smallest element in the array and exchanges it with the element in the first position. Then finds the second smallest element and exchanges it with the element in the second position and continues until the entire array is sorted.

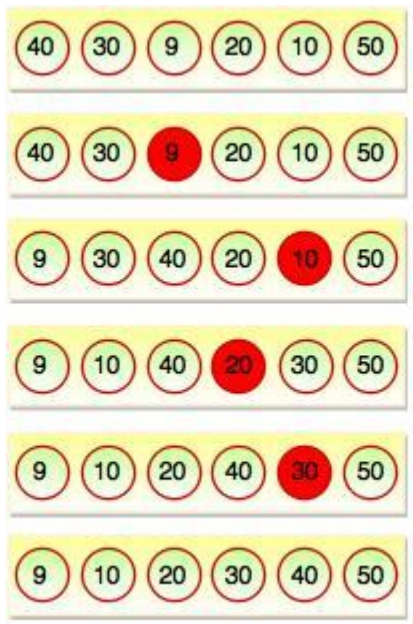


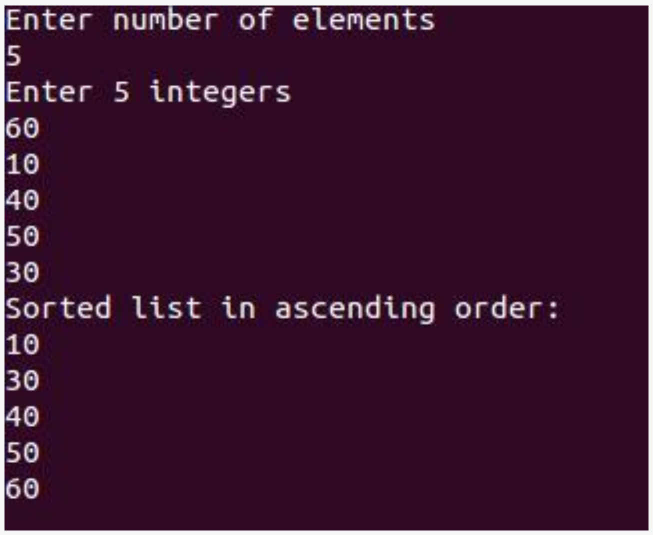
Fig. Working of Selection Sort

- In the above diagram, the smallest element is found in first pass that is 9 and it is placed at the first position. In second pass, smallest element is searched from the rest of the element excluding first element. Selection sort keeps doing this, until the array is sorted.

Example: Program for Selection Sort

```
#include <stdio.h>

int main()
{
  int array[100], n, c, d, position, swap;
  printf("Enter number of elements\n");
  scanf("%d", &n);
  printf("Enter %d integers\n", n);
  for ( c = 0 ; c < n ; c++ )
    scanf("%d", &array[c]);
  for ( c = 0 ; c < ( n - 1 ) ; c++ )
  {
    position = c;
    for ( d = c + 1 ; d < n ; d++ )
    {
      if ( array[position] > array[d] )
        position = d;
    }
    if ( position != c )
    {
      swap = array[c];
      array[c] = array[position];
      array[position] = swap;
    }
  }
  printf("Sorted list in ascending order:\n");
  for ( c = 0 ; c < n ; c++ )
    printf("%d\n", array[c]);
  return 0;
}
```

Output:



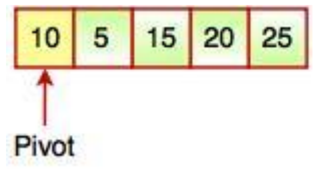
```
Enter number of elements
5
Enter 5 integers
60
10
40
50
30
Sorted list in ascending order:
10
30
40
50
60
```

Quick Sort

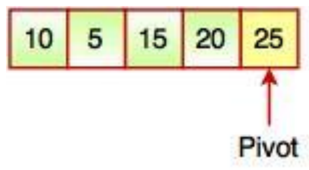
- Quick sort is also known as **Partition-exchange sort** based on the rule of **Divide and Conquer.**
- It is a highly efficient sorting algorithm.
- Quick sort is the quickest comparison-based sorting algorithm.
- It is very fast and requires less additional space, only O(n log n) space is required.
- Quick sort picks an element as pivot and partitions the array around the picked pivot.

There are different versions of quick sort which choose the pivot in different ways:

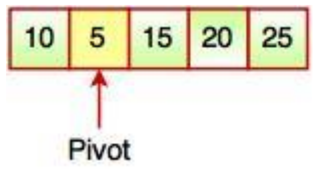
1. First element as pivot



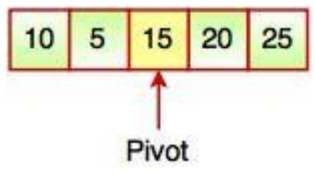
2. Last element as pivot



3. Random element as pivot



4. Median as pivot



Algorithm for Quick Sort

Step 1: Choose the highest index value as pivot.

Step 2: Take two variables to point left and right of the list excluding pivot.

Step 3: Left points to the low index.

Step 4: Right points to the high index.

Step 5: While value at left < (Less than) pivot move right.

Step 6: While value at right > (Greater than) pivot move left.

Step 7: If both Step 5 and Step 6 does not match, swap left and right.

Step 8: If left = (Less than or Equal to) right, the point where they met is new pivot.

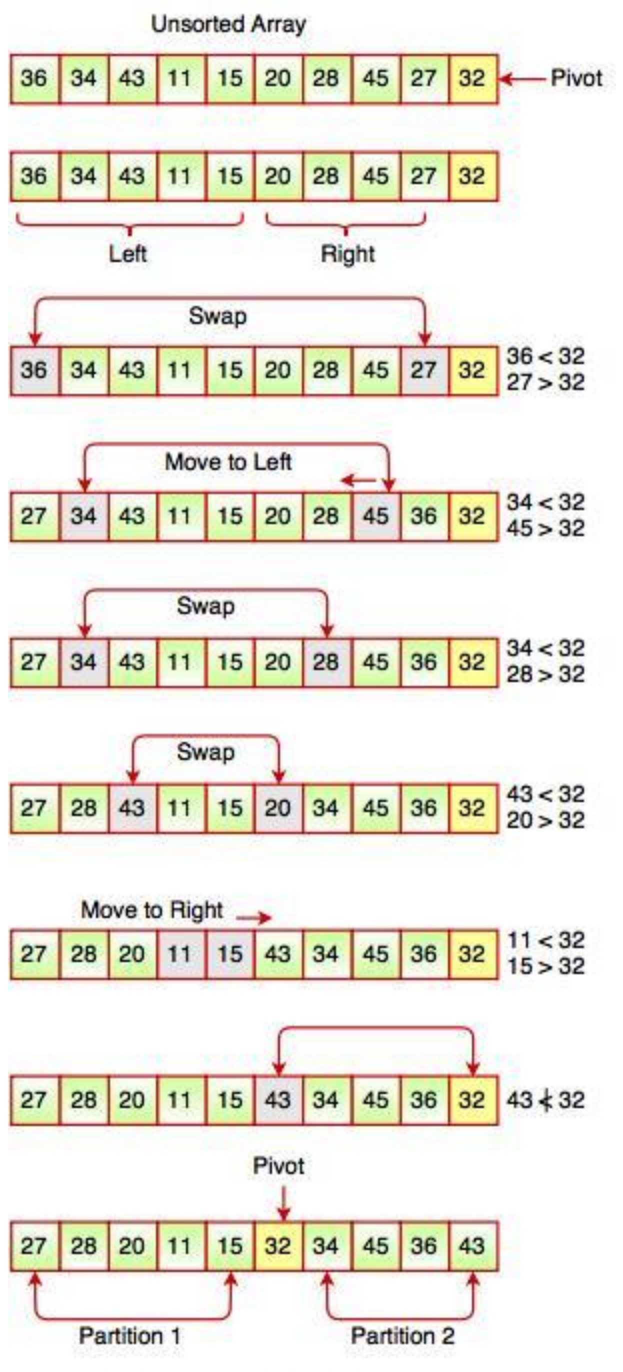


Fig. Finding Pivot Value in an Array

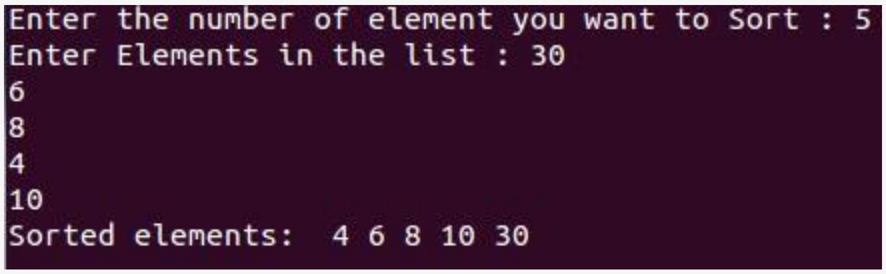
The above diagram represents how to find the pivot value in an array. As we see, pivot value divides the list into two parts (partitions) and then each part is processed for quick sort. Quick sort is a recursive function. We can call the partition function again.

Example: Demonstrating Quick Sort

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

//quick Sort function to Sort Integer array list
void quicksort(int array[], int firstIndex, int lastIndex)
{
   //declaaring index variables
   int pivotIndex, temp, index1, index2;
   if(firstIndex < lastIndex)
   {
      //assigninh first element index as pivot element
      pivotIndex = firstIndex;
      index1 = firstIndex;
      index2 = lastIndex;
      //Sorting in Ascending order with quick sort
      while(index1 < index2)
      {
         while(array[index1] <= array[pivotIndex] && index1 < lastIndex)
         {
            index1++;
         }
         while(array[index2]>array[pivotIndex])
         {
            index2--;
         }
         if(index1<index2)
         {
            //Swapping opertation
            temp = array[index1];
            array[index1] = array[index2];
            array[index2] = temp;
         }
      }
```

```
        //At the end of first iteration, swap pivot element with index2 element
        temp = array[pivotIndex];
        array[pivotIndex] = array[index2];
        array[index2] = temp;
        //Recursive call for quick sort, with partiontioning
        quicksort(array, firstIndex, index2-1);
        quicksort(array, index2+1, lastIndex);
    }
}
int main()
{
    //Declaring variables
    int array[100],n,i;
    //Number of elements in array form user input
    printf("Enter the number of element you want to Sort : ");
    scanf("%d",&n);
    //code to ask to enter elements from user equal to n
    printf("Enter Elements in the list : ");
    for(i = 0; i < n; i++)
    {
        scanf("%d",&array[i]);
    }
    //calling quickSort function defined above
    quicksort(array,0,n-1);
    //print sorted array
    printf("Sorted elements: ");
    for(i=0;i<n;i++)
       printf(" %d",array[i]);
    getch();
    return 0;
}
```

Output:



```
Enter the number of element you want to Sort : 5
Enter Elements in the list : 30
6
8
4
10
Sorted elements:  4 6 8 10 30
```

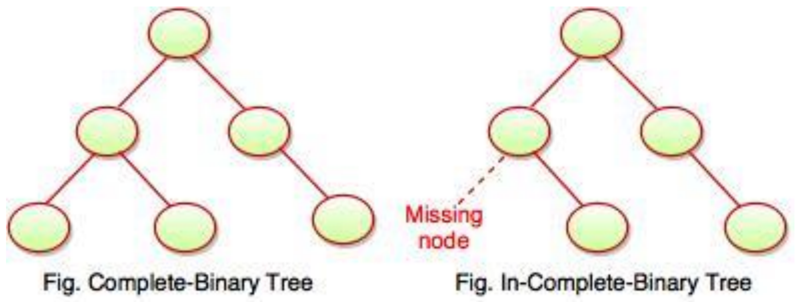
Heap Sort

- Heap sort is a comparison based sorting algorithm.
- It is a special tree-based data structure.
- Heap sort is similar to selection sort. The only difference is, it finds largest element and places the it at the end.
- This sort is not a stable sort. It requires a constant space for sorting a list.
- It is very fast and widely used for sorting.

It has following two properties:

1. Shape Property
2. Heap Property

1. Shape property represents all the nodes or levels of the tree are fully filled. Heap data structure is a complete binary tree.



2. Heap property is a binary tree with special characteristics. It can be classified into two types:

I. Max-Heap
II. Min Heap

I. Max Heap: If the parent nodes are greater than their child nodes, it is called a **Max-Heap.**

II. Min Heap: If the parent nodes are smaller than their child nodes, it is called a **Min-Heap.**

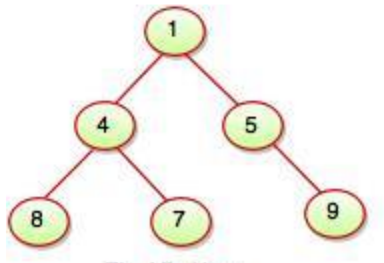


Fig. Min Heap

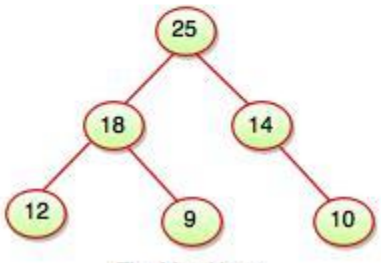


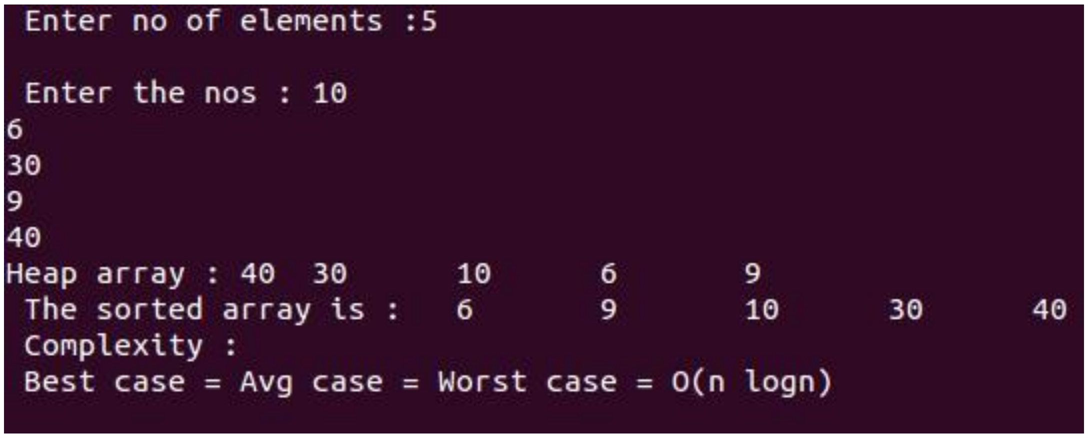
Fig. Max Heap

Example: Program for Heap Sort

```
#include <stdio.h>
void main()
{
   int heap[10], no, i, j, c, root, temp;
   printf("\n Enter no of elements :");
   scanf("%d", &no);
   printf("\n Enter the nos : ");
   for (i = 0; i < no; i++)
      scanf("%d", &heap[i]);
   for (i = 1; i < no; i++)
   {
      c = i;
      do
      {
         root = (c - 1) / 2;
         if (heap[root] < heap[c])   /* to create MAX heap array */
```

```
            {
                temp = heap[root];
                heap[root] = heap[c];
                heap[c] = temp;
            }
            c = root;
        } while (c != 0);
    }
    printf("Heap array : ");
    for (i = 0; i < no; i++)
        printf("%d\t ", heap[i]);
    for (j = no - 1; j >= 0; j--)
    {
        temp = heap[0];
        heap[0] = heap[j];    /* swap max element with rightmost leaf element
*/
        heap[j] = temp;
        root = 0;
        do
        {
            c = 2 * root + 1;    /* left node of root element */
            if ((heap[c] < heap[c + 1]) && c < j-1)
                c++;
            if (heap[root]<heap[c] && c<j)    /* again rearrange to max heap array
*/
            {
                temp = heap[root];
                heap[root] = heap[c];
                heap[c] = temp;
            }
            root = c;
        } while (c < j);
    }
    printf("\n The sorted array is : ");
    for (i = 0; i < no; i++)
       printf("\t %d", heap[i]);
    printf("\n Complexity : \n Best case = Avg case = Worst case = O(n logn)
\n");
}
```

Output:



```
 Enter no of elements :5

 Enter the nos : 10
6
30
9
40
Heap array : 40  30      10      6       9
 The sorted array is :   6       9       10      30      40
 Complexity :
 Best case = Avg case = Worst case = O(n logn)
```

What is Searching?

- Searching is the process of finding a given value position in a list of values.
- It decides whether a search key is present in the data or not.
- It is the algorithmic process of finding a particular item in a collection of items.
- It can be done on internal data structure or on external data structure.

Searching Techniques

To search an element in a given array, it can be done in following ways:

1. Sequential Search
2. Binary Search

1. Sequential Search

- Sequential search is also called as Linear Search.

- Sequential search starts at the beginning of the list and checks every element of the list.
- It is a basic and simple search algorithm.
- Sequential search compares the element with all the other elements given in the list. If the element is matched, it returns the value index, else it returns -1.

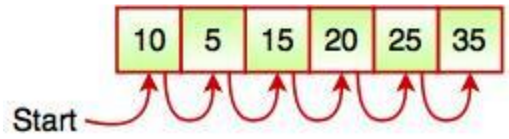


Fig. Sequential Search

The above figure shows how sequential search works. It searches an element or value from an array till the desired element or value is not found. If we search the element 25, it will go step by step in a sequence order. It searches in a sequence order. Sequential search is applied on the unsorted or unordered list when there are fewer elements in a list.

The following code snippet shows the sequential search operation:

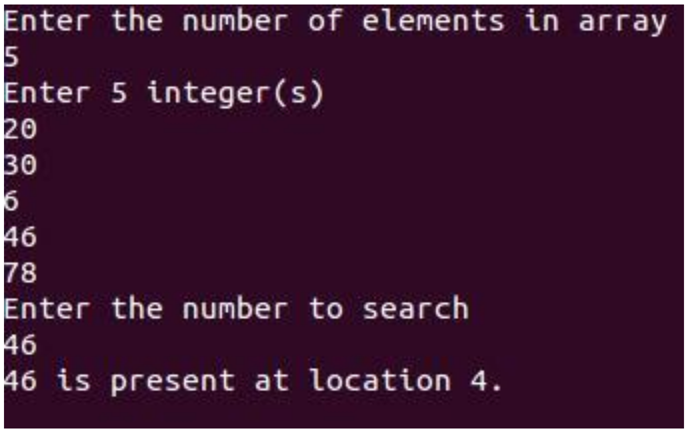
```
function searchValue(value, target)
{
    for (var i = 0; i < value.length; i++)
    {
        if (value[i] == target)
        {
            return i;
        }
    }
    return -1;
}
searchValue([10, 5, 15, 20, 25, 35] , 25);   // Call the function with array and
number to be searched
```

Example: Program for Sequential Search

```
#include <stdio.h>
int main()
```

```
{
  int arr[50], search, cnt, num;

  printf("Enter the number of elements in array\n");
  scanf("%d",&num);

  printf("Enter %d integer(s)\n", num);

  for (cnt = 0; cnt < num; cnt++)
  scanf("%d", &arr[cnt]);

  printf("Enter the number to search\n");
  scanf("%d", &search);

  for (cnt = 0; cnt < num; cnt++)
  {
    if (arr[cnt] == search)     /* if required element found */
    {
      printf("%d is present at location %d.\n", search, cnt+1);
      break;
    }
  }
  if (cnt == num)
    printf("%d is not present in array.\n", search);

  return 0;
}
```

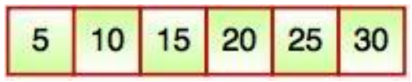
Output:



```
Enter the number of elements in array
5
Enter 5 integer(s)
20
30
6
46
78
Enter the number to search
46
46 is present at location 4.
```

2. Binary Search

- Binary Search is used for searching an element in a sorted array.
- It is a fast search algorithm with run-time complexity of O(log n).
- Binary search works on the principle of divide and conquer.
- This searching technique looks for a particular element by comparing the middle most element of the collection.
- It is useful when there are large number of elements in an array.



5	10	15	20	25	30

- The above array is sorted in ascending order. As we know binary search is applied on sorted lists only for fast searching.

For example, if searching an element 25 in the 7-element array, following figure shows how binary search works:

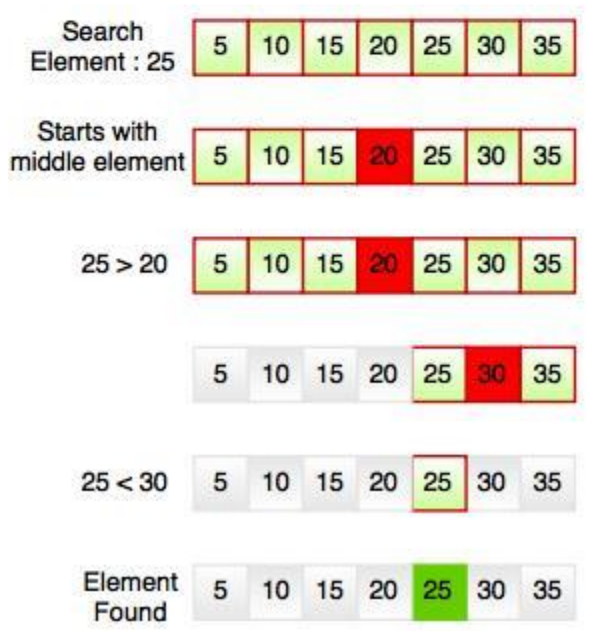


Fig. Working Structure of Binary Search

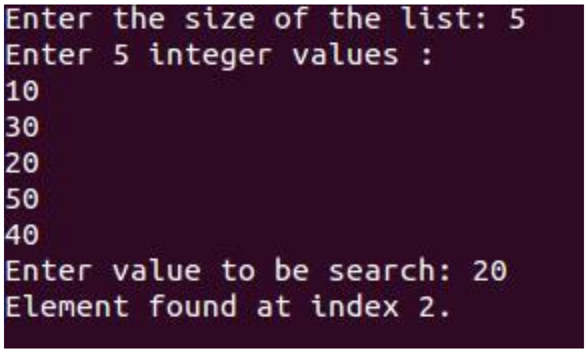
Binary searching starts with middle element. If the element is equal to the element that we are searching then return true. If the element is less than then move to the right of the list or if the element is greater than then move to the left of the list. Repeat this, till you find an element.

Example: Program for Binary Search

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

void main()
{
   int f, l, m, size, i, sElement, list[50]; //int f, l ,m : First, Last, Middle
   clrscr();

   printf("Enter the size of the list: ");
   scanf("%d",&size);

   printf("Enter %d integer values : \n", size);
```

```
  for (i = 0; i < size; i++)
    scanf("%d",&list[i]);

  printf("Enter value to be search: ");
  scanf("%d", &sElement);

  f = 0;
  l = size - 1;
  m = (f+l)/2;

  while (f <= l) {
    if (list[m] < sElement)
      f = m + 1;
    else if (list[m] == sElement) {
      printf("Element found at index %d.\n",m);
      break;
    }
    else
      l = m - 1;
    m = (f + l)/2;
  }
  if (f > l)
    printf("Element Not found in the list.");
  getch();
}
```

Output:



```
Enter the size of the list: 5
Enter 5 integer values :
10
30
20
50
40
Enter value to be search: 20
Element found at index 2.
```