214447: DATA STRUCTURE AND ALGORITHMS LABORATORY

Teaching Scheme: Practical: 4Hours/Week02 **Credits: 02**
Examination Scheme: Term Work: 25Marks
Practical : 25 Marks
Prerequisites: Fundamental knowledge of programming language and basics of Algorithms

Sr. No	List of Assignments
1	Consider a student database of SEIT class (at least 15 records). Database contains different fields of every student like Roll No, Name and SGPA.(array of structure) a) Design a roll call list, arrange list of students according to roll numbers in ascending order (Use Bubble Sort) b) Arrange list of students alphabetically. (Use Insertion sort) c) Arrange list of students to find out first ten toppers from a class. (Use Quicksort) d) Search students according to SGPA. If more than one student having sameSGPA, then print list of all students having same SGPA. e) Search a particular student according to name using binary searchwithout recursion. (all the student records having the presence of search key should be displayed) (Note: Implement either Bubble sort or Insertion Sort.)
2	Implement stack as an abstract data type using singly linked list, use this ADT for conversion of infix expression to postfix, prefix and evaluation of postfix, and prefix expression.
3	Implement Circular Queue using Array. Perform following operations on it. a) Insertion (Enqueue) b) Deletion (Dequeue) c) Display (Note: Handle queue full condition by considering a fixed size of a queue.)
4	Construct an Expression Tree from postfix and prefix expression. Performrecursive and non- recursive In-order, pre-order and post-order traversals.

5	Implement binary search tree and perform following operations: a) Insert (Handle insertion of duplicate entry) b) Delete c) Search d) Display tree (Traversal) e) Display - Depth of tree f) Display - Mirror image g) Create a copy h) Display all parent nodes with their child nodes i) Display leaf nodes j) Display tree level wise (Note: Insertion, Deletion, Search and Traversal are compulsory, from rest ofoperations, perform Any three)
6	Implement In-order Threaded Binary Tree and traverse it in In-order and Pre-order.
7	Represent a graph of your college campus using adjacency list /adjacency matrix.Nodes should represent the various departments/institutes and links should represent the distance between them. Find minimum spanning tree a) Using Kruskal's algorithm. b) Using Prim's algorithm.
8	Represent a graph of city using adjacency matrix /adjacency list. Nodes should represent the various landmarks and links should represent the distance betweenthem. Find the shortest path using Dijkstra's algorithm from single source to all destination.
9	Implement Heap sort to sort given set of values using max or min heap.
10	Department maintains student's database. The file contains roll number, name,division and address. Write a program to create a sequential file to store and maintain student data. It should allow the user to add, delete information of student. Display information of particular student. If record of student does notexist an appropriate message is displayed. If student record is found it should display the student details.

Assignment 1-Searching and Sorting

AIM: To implement various searching and sorting algorithms on database of students implemented using array of structure.

DETAILED PROBLEM STATEMENT :

Consider a student database of SEIT class (at least 15 records). Database contains different Fields of every student like Roll No, Name and SGPA.(array of structure)
a. Design a roll call list, arrange list of students according to roll numbers in ascending order (Use Bubble Sort)
b. Arrange list of students alphabetically. (Use Insertion sort)
c. Arrange list of students to find out first ten toppers from a class. (Use Quick sort)
d. Search students according to SGPA. If more than one student having same SGPA, then print list of all students having same SGPA.
e. Search a particular student according to name using binary search without recursion. (all the student records having the presence of search key should be displayed)
(Note: Implement either Bubble sort or Insertion Sort.)

OBJECTIVE:

1. To study the concepts of array of structure.
2. To understand the concepts, algorithms and applications of Sorting.
3. To understand the concepts, algorithms and applications of Searching.

OUTCOME:

1. Understand linear data structure - array of structure.
2. Apply different sorting techniques on array of structure (Bubble,
3. Insertion and Quicksort) and display the output for every pass.
4. Apply different searching techniques on array of structure (Linear Search, Binary Search) and display the output for every pass.
5. Calculate time complexity

THEORY:

1. Structure:

Structure is collection of heterogeneous types of data. In C++, a structure collects different data items in such a way that they can be referenced as a single unit. Data items in a structure are called fields or members of the structure.

Creating a Structures:

```
struct student
{
        int roll_no;
            char name[15];
        float sgpa;
```

};
Here struct is keyword used to declare a structure. student is the name of the structure. In this example, there are three types of variables int, char and float. The variables have their own names, such as roll_no, name, sgpa.

Declaring Structure Variables:

After declaring a structure, you can define the structure variables. For instance, the following structure variables are defined with the structure data type of automobile from the previous section:

struct student s1, s2, s3;

Here three structure variables, s1, s2, s3 are defined by the structure of student. All three structure variables contain the three members of the structure student.

Referencing Structure Members with the Dot Operator:

Given the structure student and s1 is variable of type struct student we can access the fields in following way:

s1.roll_no
s1.name
s1.sgpa

Here the structure name and its member's name are separated by the dot (.) operator.

Pointer to Structure:

We can declare a pointer variable which is capable of storing the address of a structure variable as follows:

struct student *ptr;

Suppose if we have structure variable declared as:

struct student s1;

Then ptr can store the address of s1 by:

ptr = &s1;

Referencing a Structure Member with -> (arrow operator)

Using ptr we can access the fields of s1 as follows:

ptr->name or (*ptr).name

Because of its clearness, the -> operator is more frequently used in programs than the dot operator.

2. **Array of Structure:**

To declare an array of a structures, we first define a structure and then declare an array variable of that type. To declare an 20 element array of structures of type student define earlier, we write,

struct student s[20];

This creates 20 sets of variables that are organized as defined in the structure student.

To access a specific structure, index the array name. For example, to print the name of 4th customer, we write

cout<< s[3].name;

3. **Sorting:**

It is the process of arranging or ordering information in the ascending or descending order of the key

values.

- **Bubble Sort**

This is one of the simplest and most popular sorting methods. The basic idea is to pass through the array sequentially several times. In each pass we compare successive pairs of elements (A[i] with A[i+1]) and interchange the two if they are not in the required order. One element is placed in its correct position in each pass. In the first pass, the largest element will sink to the bottom, second largest in the second pass and so on. Thus, a total of n-1 passes are required to sort 'n' keys.

Analysis:

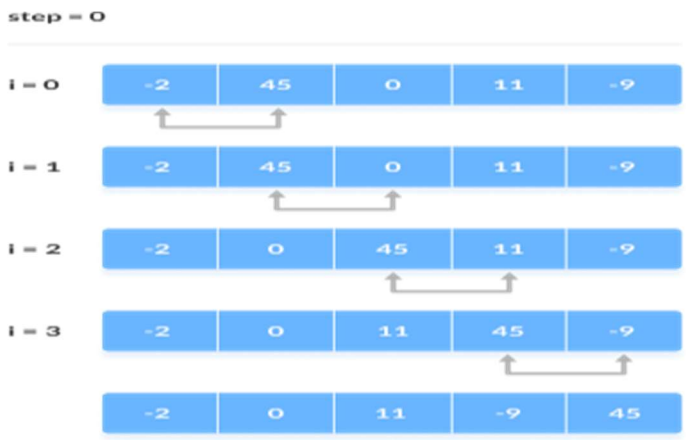
In this sort, n-1 comparisons takes place in 1^{st} pass , n-2 in 2^{nd} pass , n-3 in 3^{rd} pass , and so on.
Therefore total number of comparisons will be

$(n-1) + (n-2) + (n-3) + \ldots\ldots + 3 + 2 + 1$

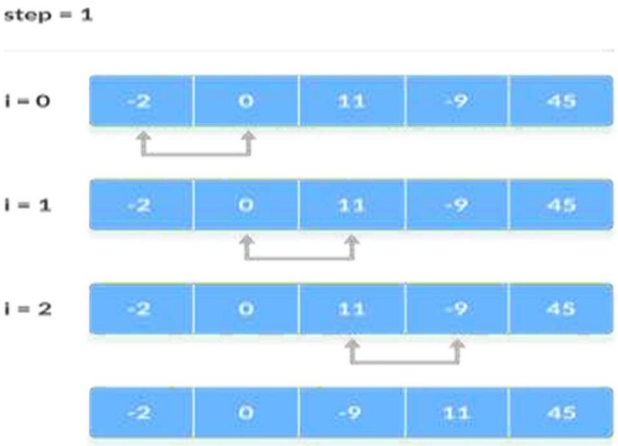
This is an Arithmetic series in decreasing order
Sum = n/2 [2a + (n-1)d]
Where, n = no. of elements in series , a = first element in the series
d = difference between second element and first element.
Sum = (n-1) / 2 [2 * 1 + ((n-1) - 1) * 1]
= n(n-1) / 2
which is of order n^2 i.e $O(n^2)$
The main advantage is simplicity of algorithm and it behaves as O(n) for sorted array of element. Additional space requirement is only one temporary variable.

Example:

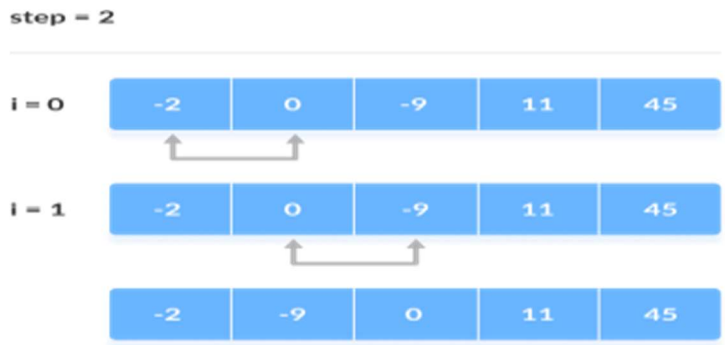
Compare two consecutive elements, if first element is greater than second then interchange the elements else no change.
Pass 1:



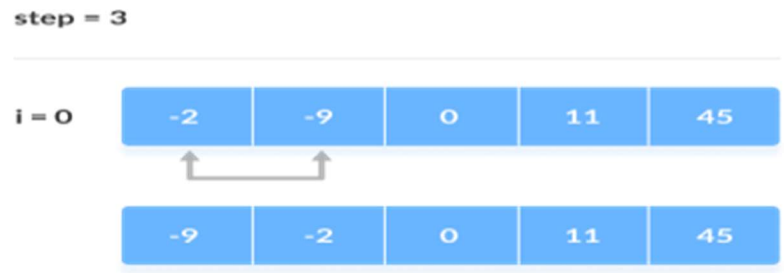
Pass 2:



Pass 3:



Pass 4:



➢ Insertion Sort

Insertion sort works similarly as we sort cards in our hand in a card game.
We assume that the first card is already sorted then, we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left. In the same way, other unsorted cards are taken and put at their right place. A similar approach is used by insertion sort.
Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in every iteration.

Analysis:

Worst Case Complexity: O(n2)
Suppose, an array is in ascending order, and you want to sort it in descending order. In this case, worst case complexity occurs.
Each element has to be compared with each of the other elements so, for every nth element, (n-1) number of comparisons are made.
Thus, the total number of comparisons = n*(n-1) ~ n2

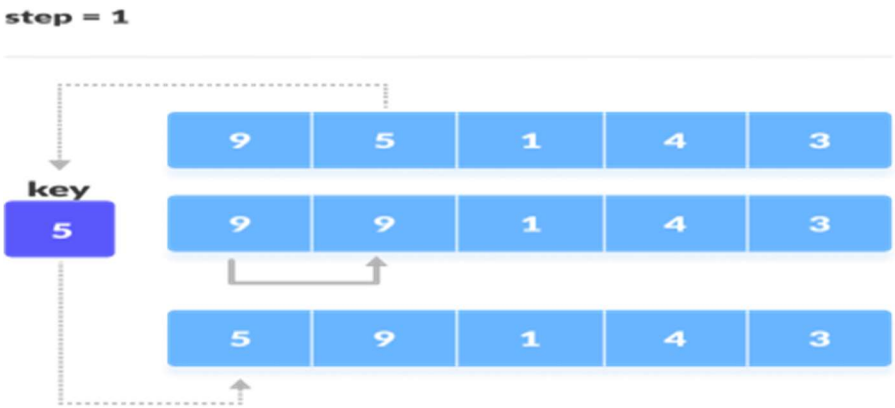
Best Case Complexity: O(n)
When the array is already sorted, the outer loop runs for n number of times whereas the inner loop does not run at all. So, there are only n number of comparisons. Thus, complexity is linear.

Average Case Complexity: O(n2)
It occurs when the elements of an array are in jumbled order (neither ascending nor descending).

Example:

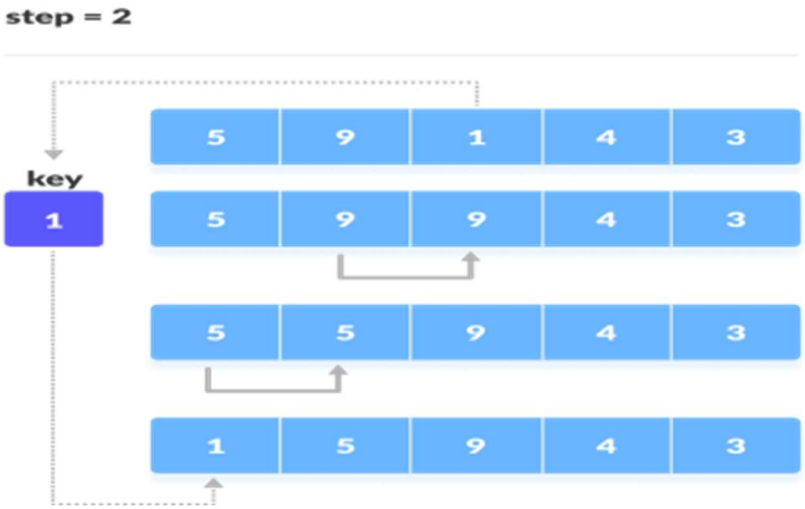
Suppose we need to sort the following array.

1. The first element in the array is assumed to be sorted. Take the second element and store it separately in key.
   Compare key with the first element. If the first element is greater than key, then key is placed in front of the first element.

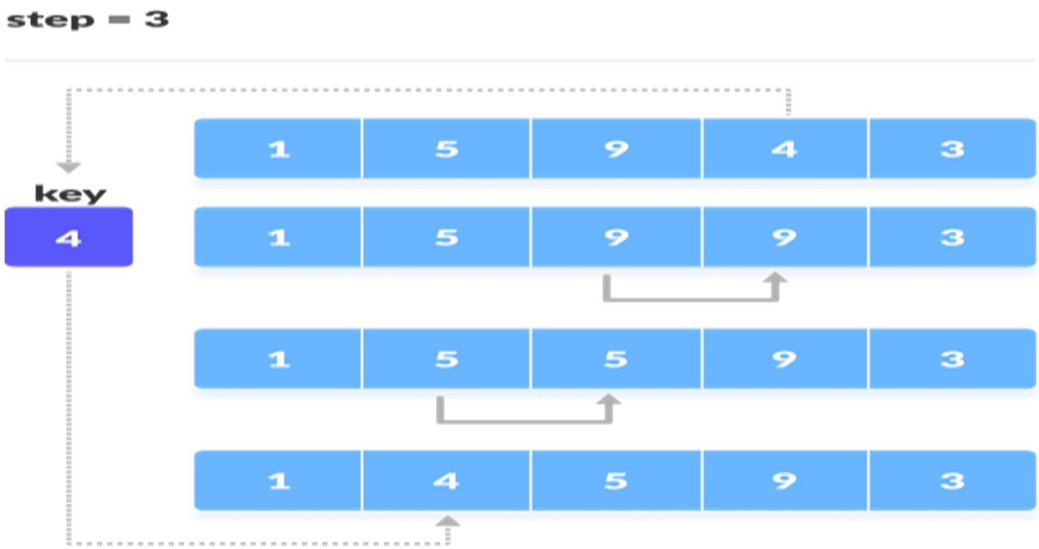


2. Now, the first two elements are sorted.

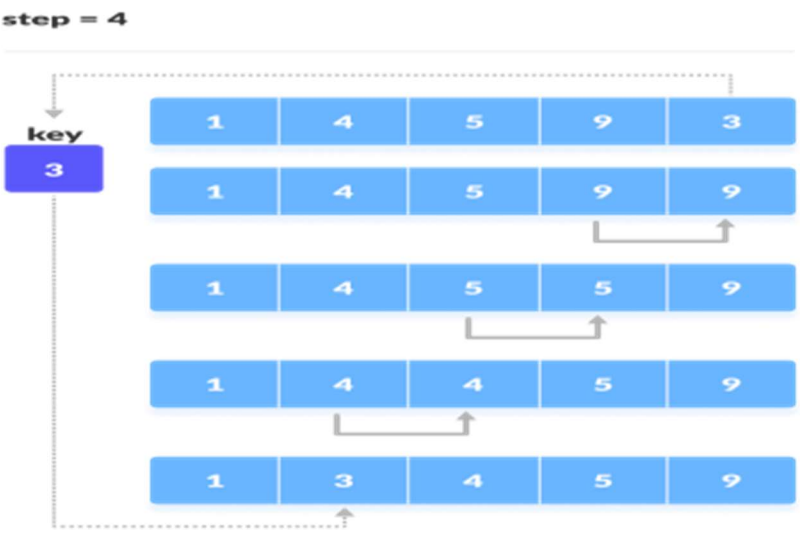
Take the third element and compare it with the elements on the left of it. Placed it just behind the element smaller than it. If there is no element smaller than it, then place it at the beginning of the array.



3. Similarly, place every unsorted element at its correct position.
   Place 4 behind 1



Place 3 behind 1 and the array is sorted:



- **Quick Sort**

Quick sort is an algorithm based on divide and conquers approach in which the array is split into sub arrays and these sub-arrays are recursively called to sort the elements.

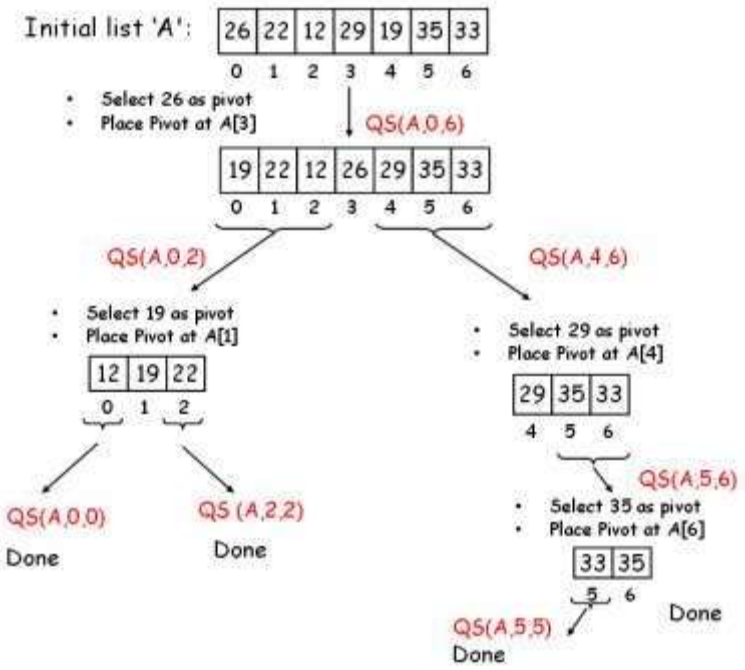
Analysis:

Worst Case Complexity [Big-O]: O(n2)
It occurs when the pivot element picked is either the greatest or the smallest element. This condition leads to the case in which the pivot element lies in an extreme end of the sorted array. One sub-array is always empty and another sub-array contains n - 1 elements. Thus, quick sort is called only on this sub-array. However, the quick sort algorithm has better performance for scattered pivots.

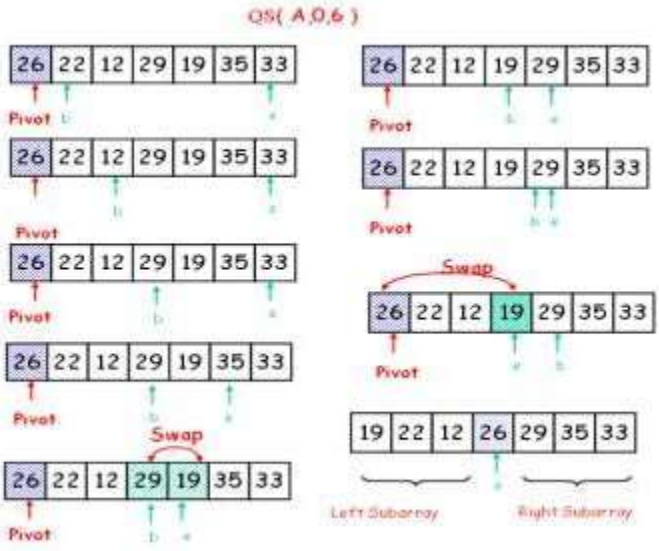
Best Case Complexity [Big-omega]: O(n*log n)
It occurs when the pivot element is always the middle element or near to the middle element.

Average Case Complexity [Big-theta]: O(n*log n)
It occurs when the above conditions do not occur.

Example:



Step by Step Process for QS (A, 0, 6)



- **Searching:**
  Searching is a process of retrieving or locating a record with a particular key value.

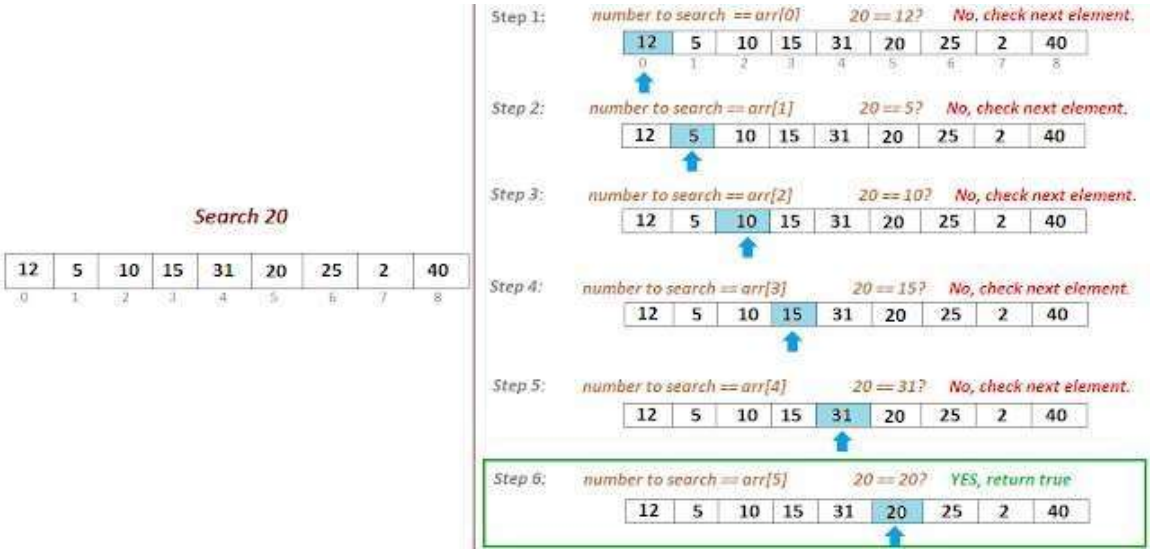
➢ **Linear Search**

Linear search is the simplest searching algorithm that searches for an element in a list in sequential order. We start at one end and check every element until the desired element is not found.

Analysis:

Time Complexity: O(n)
Space Complexity: O(1)

Example:



- **Binary Search**

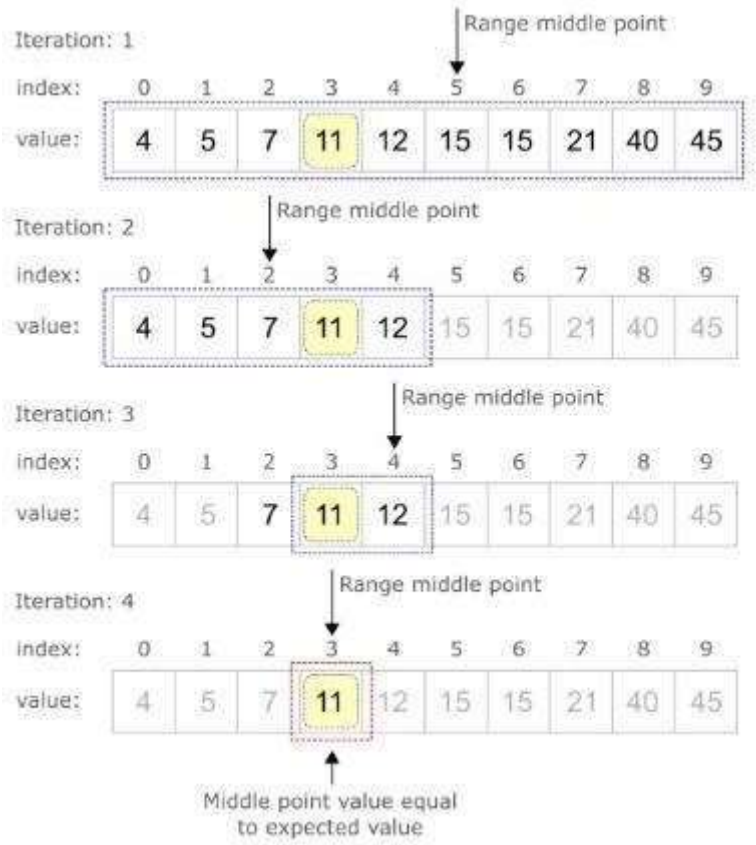
This is a very efficient searching method used for linear / sequential data. In this search, Data has to be in the sorted order either ascending or descending. Sorting has to be done based on the key values.
In this method, the required key is compared with the key of the middle record. If a match is found, the search terminates. If the key is less than the record key, the search proceeds in the left half of the table. If the key is greater than record key, search proceeds in the same way in the right half of the table. The process continues till no more partitions are possible. Thus every time a match is not found, the remaining table size to be searched reduces to half.
If we compare the target with the element that is in the middle of a sorted list, we have three possible results: the target matches, the target is less than the element, or the target is greater than the element. In the first and best case, we are done. In the other two cases, we learn that half of the list can be eliminated from consideration.
When the target is less than the middle element, we know that if the target is in this ordered list, it must be in the list before the middle element. When the target is greater than the middle element, we know that if the target is in this ordered list, it must be in the list after the middle element. These facts allow this one comparison to eliminate one-half of the list from consideration. As the process continues, we will eliminate from consideration, one-half of what is left of the list with each comparison. This technique is called as binary search.

Analysis:

After 0 comaparisons □ Remaining file size = n
After 1 comaparisons □ Remaining file size = n / 2 = n / 2^1
After 2 comaparisons □ Remaining file size = n / 4 = n / 2^2
After 3 comaparisons □ Remaining file size = n / 8 = n / 2^3
……
After k comaparisons □ Remaining file size = n / 2^k

The process terminates when no more partitions are possible i.e. remaining file size = 1
n / 2^k = 1
k = $\log_2 n$
Thus, the time complexity is $O(\log_2 n)$. which is very efficient.

Example:



ALGORITHM / PSEUDOCODE :

- **Create a structure**

Create_database(struct student s[])
Step 1: Accept how many records user need to add, say, no of records as n
Step 2: For i = 0 to n – 1
i. Accept the record and store it in s[i]
Step 3: End For
Step 4: stop

Display_database(struct student s[] ,int n)
Step 1: For i = 0 to n – 1
i. Display the fields s.roll_no, s.name, s.sgpa
Step 2: End For
Step 3: Stop

Bubble Sort student according to sort to roll numbers
BubbleSort(Student s[], n)
Step 1: For Pass = 1 to n-1
Step 2: For i = 0 to (n – pass – 1)
i. If s[i].roll_no < s[i+1].roll_no

a. Swap (s[i]. s[i+1])

iii. End if

Step 3: End for

Step 4: End For

Step 5 : Stop

- **Insertion Sort to sort student on the basis of names**

insertion_Sort (Struct student S[], int n)

Step 1: For i = 1 to n-1

i. Set key to s[i]

ii. Set j to i-1

iii. While j>=0 AND strcmp(s[i].name,key.name)>0

a. Assign s[j] to s[j+1]

b. Decrement j

iv. End While

Step 2: Assign key to s[j+1]

Step 3: End for

Step 4: end of insertion sort

- **Quick Sort to sort students on the basis of their sgpa.**

partition (struct student s[], int l, int h)

// where s is the array of structure , l is the index of starting element

// and h is the index of last element.

Step 1: Select s[l].sgpa as the pivot element

Step 2: Set i = l

Step 3: Set j = h-1

Step 4: While $i \leq j$

i. Increment i till s[i].sgpa $\leq$ pivot element

ii. Decrement j till s[j].sgpa > pivot element

iii. If i < j

iv. Swap(s[i], s[j])

v. End if

Step 5: End while

Step 6: Swap(s[j],s[l])

Step 7: return j

Step 8 :end of Partition

quicksort(struct student s[], int l, int h)

//where s is the array of structure , l is the index of starting element

//and h is the index of last element.

Step 1: If l<h

i. P=partition(s,l,h)

ii. quicksort (s,l,p-1)

iii. quicksort (s,p+1,h)

Step 2: End if

Step 3: end of quicksort

- **Algorithm for Line ar Search to search students with sgpa given and display all of them**
  Linear_search (struct student s[], float key, int n)
  //Here s is array of structure student, key is sgpa of student to be searched and
  // displayed, n is total number of students in record
  Step 1: Set i to 0 and flag to 0
  Step 2: While i<n
  i. If s[i].sgpa==key
  a. Print s[i].roll_no, s[i].name
  b. Set flag to 1
  c. i++
  Step 3: End while
  Step 4: If flag==0
  i. Print No student found with sgpa=value of key
  Step 5: End if
  Step 6: End of linear_ sear
- **Algorithm for Binary Search to search students having given string in their names**
  Binary_Search (s, n , Key)
  // Where s is an array of structure , n is the no of records, and key is element to be searched
  Step 1: Set l = 0 & h = n-1
  Step 2: While l ≤ h
  i. mid = (l + h) / 2
  ii. If strcmp (s[mid].name, key)==0)
  a. foun
  b. stop
  iii. Else
  a. if (strcmp (key, s[mid].name)<0
  i. h = mid – 1
  b. Else
  ii. l = mid + 1
  c. End if
  iv. End if
  Step 4: End while
  Step 5: not found // search is unsuccessful

Validation:

- Limit of the array should not be –ve, and should not cross the lower and upper bound.
- Roll numbers should not repeat, should not -ve
- Name should only contain alphabets, space and .
- Should not allowed any other operations before the input list is entered by the user .
- Before going to (binary search) records should be sorted according a names.

Test Cases :

- **Sorting :**
  Four test cases :

i. Already Sorted according to the requirement
ii. Sorted in reverse order
iii. Partially sorted
iv. Completely Random List

Expected output /analysis is :

i. Test algorithm for above four test cases
ii. Analyze the algorithms based on no of comparisons and swapping/shifting required
iii. Check for Sort Stability factor
iv. No of passes needed
v. Best /average/ worst case of the each algorithm based on above test case
vi. Memory space required to sort

Searching :

- Find the max and minimum comparison required to search element
- Calculate how many comparisons are required for unsuccessful search

Application:

Useful in managing large amount of data.

FAQ:

- What is the need of sorting
- What is searching?
- How to get output after each pass?
- What do you mean bypass?
- What is recursion?
- Where the values are stored in recursion?
- Explain the notation Ω, θ, O for time analysis.
- Explain the time complexity of bubble sort and linear search, binary search.
- What is need of structure?
- What are the differences between a union and a structure?
- Which operators are used to refer structure member with and without pointer?
- List the advantages and disadvantages of Sequential / Linear Search?
- List the advantages and disadvantages of binary Search?
- What you mean by internal & External Sorting?
- Analyze Quick sort with respect to Time complexity
- In selecting the pivot for QuickSort, which is the best choice for optimal partitioning:
  a) The first element of the array
  b) The last element of the array
  c) The middle element of the array
  d) The largest element of the array
  e) The median of the array
  f) Any of the above?
- Explain the importance of sorting and searching in computer applications?
- Analyze bubble sort with respect to time complexity.
- What is Divide and conquer methodology?
- How the pivot is selected and partition is done in quick sort?

- What is the complexity of quick sort?

Assignment 2 -Implementation of Stack

AIM: Implementation of Stack ADT and expression conversion
DETAILED PROBLEM STATEMENT:

Write a program to implement stack as an abstract data type using linked list and use this ADT for conversion of infix expression to postfix, prefix and evaluation of postfix/prefix expression.

OBJECTIVE:

1. To understand the concept and implementation of Stack
2. data structure using SLL.
3. To understand the concept of conversion of expression.
4. To understand the concept of evaluation of expression
5. To compare and analyze time and space complexity

OUTCOME:

1. Code linear data structure using array of structure.
2. Apply different sorting techniques on array of structure (Bubble,
3. Insertion and Quicksort) and display the output for every pass.
4. Apply different searching techniques on array of structure (Linear Search, Binary Search) and display the output for every pass..
5. Calculate time complexity.

THEORY:

What is an abstract datatype?

An Abstract Data type is defined as a mathematical model of the data objects that make up a data type as well as the functions that operate on these objects. There are no standard conventions for defining them. A broad division may be drawn between "imperative" and "functional" definition styles. In general terms, an abstract data type is a *specification* of the values and the operations that has two properties:

- It specifies everything you need to know in order to use the datatype
- It makes absolutely no reference to the manner in which the data type will be implemented.

When we use abstract data types, our programs divide into two pieces:

- The Application: The part that uses the abstract datatype.
- The Implementation: The part that implements the abstract datatype.

Concept of Linear data structure using linked organization

Linked lists provides a way to represent an ordered or linear list, in which each item in the list is a part of structure that also contains a "link" to the structure containing the next item. Each item has only one predecessor and only one successor.

Each structure of the list is called as a 'node' & it consists of two fields,

i) One containing the 'item' &

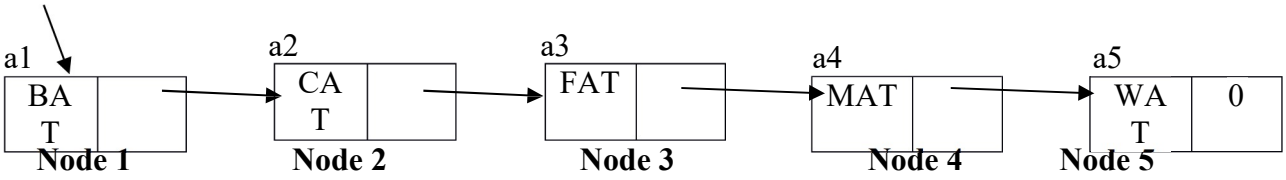
ii) Second containing the 'address' of the next 'node'.

- A linked list therefore can be defined as a collection of structures ordered not by their physical placement in memory (like an array} but by logical links that are stored as part of data in the structure or a node itself. The link is in the form of pointer to another structure or node of the same type.
- Such a structure is represented as follows:

```
struct node
{
        <data type> item;
        struct node *next;
};
```

For example, consider the following ordered list stored as a linked list

L = {BAT, CAT, FAT, MAT, WAT}



Dynamic Memory Management:

1) Allocating a block of memory using new operator:

new operator allocates memory of specified size & returns a pointer of specified type . The format is,

```
node *newp;
newp = new node;
```

where 'newp' is a pointer of type node, pointing to a block of memory of size node and it is NULL if not enough space is available.

2) Releasing the used memory space using delete operator:

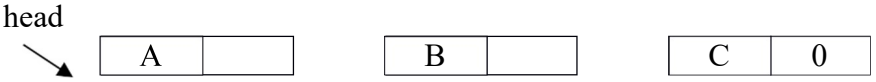
The format is,

```
delete newp;
```

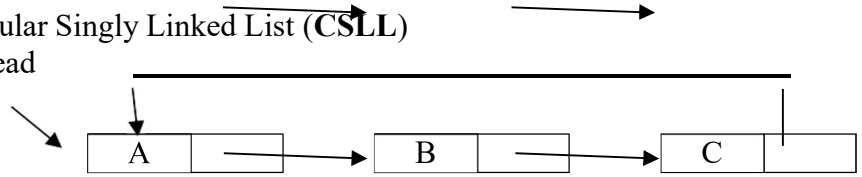
will release a memory block allocated by new operator pointed by pointer 'newp'.

Types of Linked Lists:

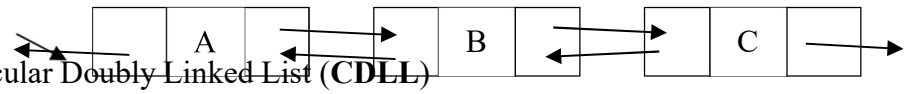
1. Singly Linked List (**SLL**)



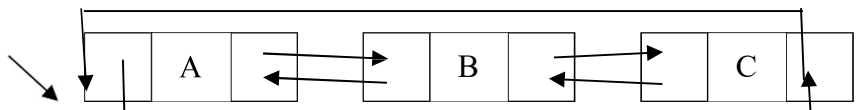
2. Circular Singly Linked List (**CSLL**)



3. Doubly Linked List (**DLL**)



4. Circular Doubly Linked List (**CDLL**)



Example of Linear data structure

Data structure whose elements (objects) are sequential and ordered in a way so that:

- there is only one *first element* and has only one *next element*,
- there is only one *last element* and has only one *previous element*, while all other elements have a *next* and a *previous* element

  Arrays, Strings, Stack, Queue, Lists

1. **Stack :**

   i. **Concept**

   Stacks are more common data objects found in computer algorithms. It is a special case of more general data object, an ordered or linear list. It can be defined as an ordered list, in which all insertions & deletions are made at one end, called as 'top' i.e. Last element inserted is outputted first (**L**ast **I**n **F**irst **O**ut or **LIFO**).

   Stack can be represented mathematically as:

   $S = \{a_1, a_2, \ldots, a_n\}$ where a_1 is the bottommost element & a_n is topmost element. & a_{i+1} is on the top of the element a_i , $1 < i <= n$.

ii. Definition of stack

In computer science, a **stack** is a last in, first out (LIFO) abstract data type and data structure. A stack can have any abstract data type as an element, but is characterized by only two fundamental operations: push and pop. The push operation adds an item to the top of the stack, hiding any items already on the stack, or initializing the stack if it is empty. A pop either reveals previously concealed items, or results in an empty stack. A stack is a restricted data structure, because only a small number of operations are performed on it. The nature of the pop and push operations also mean that stack elements have a natural order. Elements are removed from the stack in the reverse order to the order of their addition: therefore, the lower elements are those that have been on the stack the longest. A collection of items in which only the most recently added item may be removed. The latest added item is at the top. Basic operations are push and pop. Often top and is Empty are available, too. Also known as "last-in, first-out" or LIFO.

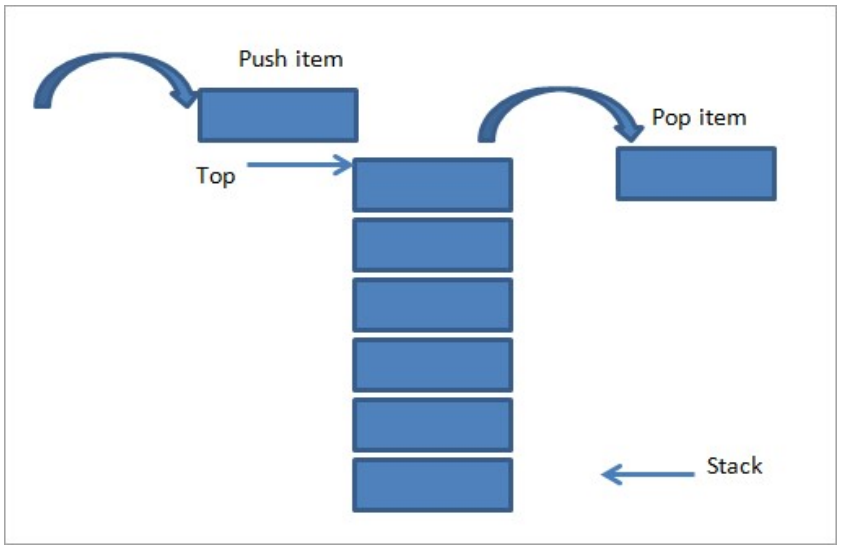
iii. Terminology with stack

An abstract data type (ADT) consists of a data structure and a set of **primitives**

a. **Initialize** creates/initializes the stack
b. **Push** adds a new element
c. **Pop** removes a element
d. **IsEmpty** reports whether the stack is empty
e. **IsFull** reports whether the stack is full
f. **Destroy** deletes the contents of the stack (may be implemented by re-initializing the stack)

iv. Diagram (vertical view)

Given below is a pictorial representation of Stack.



As shown above, there is a pile of plates stacked on top of each other. If we want to add another item to it, then we add it at the top of the stack as shown in the above figure (left-hand side). This operation of adding an item to stack is called "**Push**".

On the right side, we have shown an opposite operation i.e. we remove an item from the stack. This is also done from the same end i.e. the top of the stack. This operation is called "**Pop**".

As shown in the above figure, we see that push and pop are carried out from the same end. This makes

the stack to follow LIFO order. The position or end from which the items are pushed in or popped out to/from the stack is called the "**Top of the stack**".

Initially, when there are no items in the stack, the top of the stack is set to -1. When we add an item to the stack, the top of the stack is incremented by 1 indicating that the item is added. As opposed to this, the top of the stack is decremented by 1 when an item is popped out of the stack

ADT of Stack

Define Structure for stack(Data, Next Pointer)

- **Stack Empty:**

Return True if Stack Empty else False.
Top is a pointer of type structure stack.

Empty(Top)

```
1.  if Top == NULL
2.   return 1
3. else
4.      return 0
```

- **Push Operation:**

Top & Node pointer of structure Stack.
Push(element)

```
1.  Node->data = element; //
2.  Node->Next = Top;
3.  Top = Node
4. Stop.
```

- **Pop Operation:**

Top & Temp pointer of structure Stack.
Pop()

```
1. if Top != NULL
        i. Temp = Top;
        ii.  element=Temp->data;
        iii.   Top = (Top)->Next;
        iv. delete temp;
2. Else Stack is Empty.
3.  return element;
```

Realization of Stack ADT

i. Using Sequential organization (Array)

The three basic stack operations are push, pop, and stack top. Push is used to insert data into the stack. Pop removes data from a stack and returns the data to the calling module. Stack top returns the data at the top of t`he stack without deleting the data from the stack.

Push

Push adds an item at the top of the stack. After the push, the new item becomes the top. The only potential problem with this simple operation is that we must ensure that there is room for the new item. If there is not enough room, the stack is in an overflow state and the item cannot be added.

Following figure shows the push stack operation.

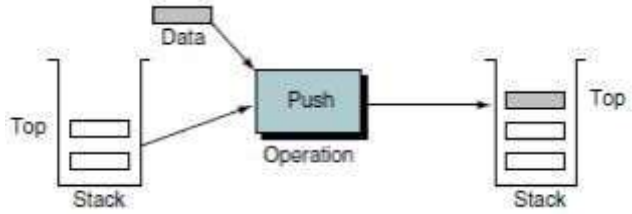


Fig.: Push Stack operation

Pop

When we pop a stack, we remove the item at the top of the stack and return it to the user. Because we have removed the top item, the next older item in the stack becomes the top. When the last item in the stack is deleted, the stack must be set to its empty state. If pop is called when the stack is empty, it is in an underflow state. The pop stack operation is shown in following figure

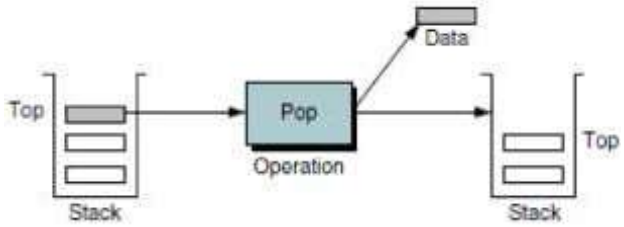


Fig.: Pop Stack operation

Stack top

The third stack operation is stack top. Stack top copies the item at the top of the stack; that is, it returns the data in the top element to the user but does not delete it. You might think of this operation as reading the stack top. Stack top can also result in underflow if the stack is empty. The stack top operation is shown in following figure

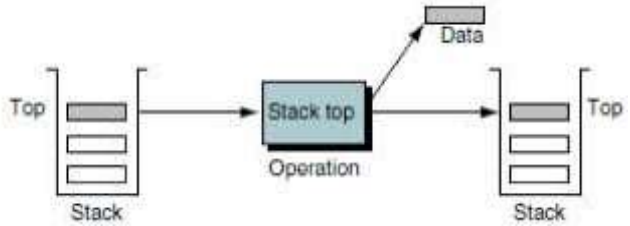


Fig.: Stack top operation

Following figure traces these three operations in an example. We start with an empty stack and push *green* and *blue* into the stack. At this point the stack contains two entries. We then pop *blue* from the top of the stack, leaving *green* as the only entry. After pushing *red*, the stack again contains two entries. At this point we retrieve the top entry, *red*, using stack top. Note that stack top does not remove *red* from the stack; it is still the top element. We then pop *red*, leaving *green* as the only entry. When *green* is popped, the stack is again empty. Note also how this example demonstrates the last in–first out operation of a stack.
Although *green* was pushed first, it is the last to be popped.

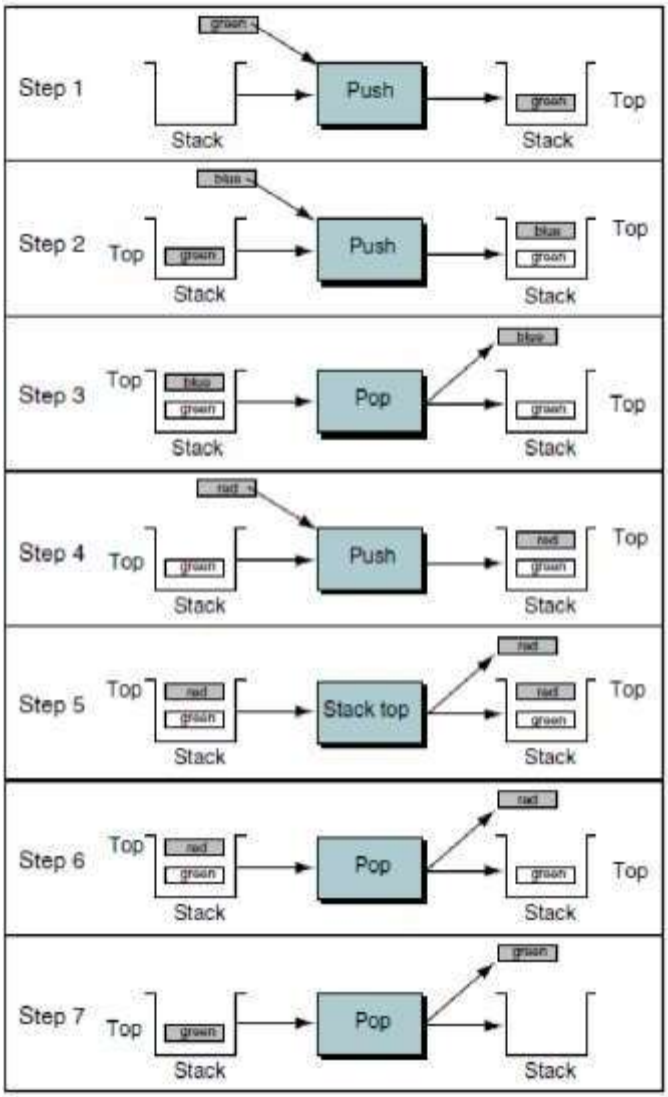


Fig.: Stack Example

ii. Using Linked organization (Linked list)

To implement the linked list stack, we need two different structures, a head and a data node. The head structure contains metadata—that is, data about data—and a pointer to the top of the stack. The data structure contains data and a link pointer to the next node in the stack. The conceptual and physical implementations of the stack are shown in following figure.

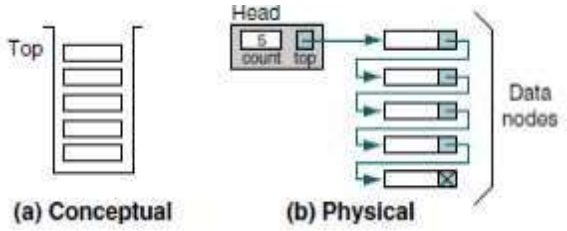


Fig.: Conceptual and Physical Stack Implementations

Stack Head

Generally, the head for a stack requires only two attributes: a top pointer and a count of the number of elements in the stack. These two elements are placed in a structure. Other stack attributes can be placed here also. For example, it is possible to record the time the stack was created and the total number of items that have ever been placed in the stack. These two metadata items allow the user to determine the average number of items processed through the stack in a given period. Of course, we would do this only if such a statistic were required for some reason. A basic head structure is shown in Figure.

Stack Data Node

The rest of the data structure is a typical linked list data node. Although the application determines the data that are stored in the stack, the stack data node looks like any linked list node. In addition to the data, it contains a link pointer to other data nodes, making it a self-referential data structure. In a self-referential structure, each instance of the structure contains a pointer to another instance of the same structure. The stack data node is also shown in following figure

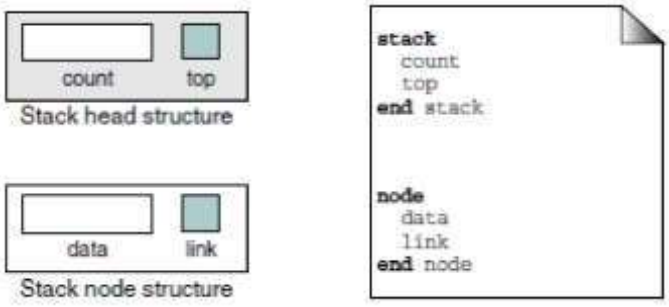


Fig.: Stack head, Stack Data node structure

We use the design shown in following figure, which demonstrates the four most common stack operations: create stack, push stack, pop stack, and destroy stack. Operations such as stacktop are not shown in the figure because they do not change the stack structure.

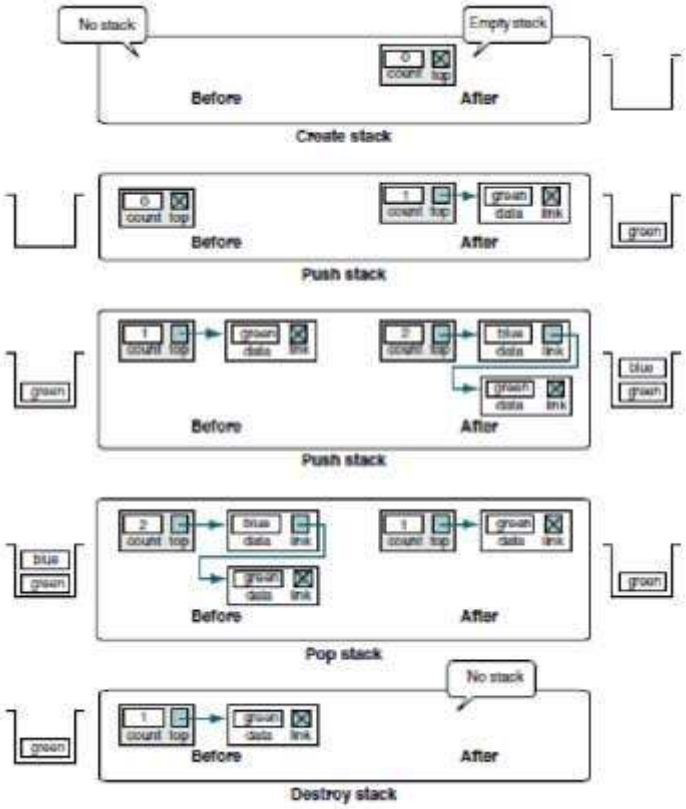


Fig.: Stack Operations

Push Stack

Push stack inserts an element into the stack. The first thing we need to do when we push data into a stack is find memory for the node. We must therefore allocate a node from dynamic memory. Once the memory is allocated, we simply assign the data to the stack node and then set the link pointer to point to the node currently indicated as the stack top. We also need to update the stack top pointer and add 1 to the stack count field. Figure traces a push stack operation in which a new pointer (pNew) is used to identify the data to be inserted into the stack.

To develop the insertion algorithm using a linked list, we need to analyze three different stack conditions: (1) insertion into an empty stack, (2) insertion into a stack with data, and (3) insertion into a stack when the available memory is exhausted. The first two of these situations are shown in Figure 3-8. The third is an error condition.

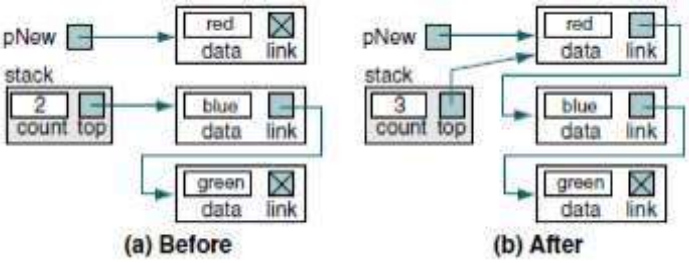


Fig.: Push Stack Example

When we insert into a stack that contains data, the new node's link pointer is set to point to the node currently at the top, and the stack's top pointer is set to point to the new node. When we insert into an empty stack, the

new node's link pointer is set to null and the stack's top pointer is set to point to the new node. However, because the stack's top pointer is null, we can use it to set the new node's link pointer to null. Thus the logic for inserting into a stack with data and the logic for inserting into an empty stack are identical.

Pop Stack

Pop stack sends the data in the node at the top of the stack back to the calling algorithm. It then adjusts the pointers to logically delete the node. After the node has been logically deleted, it is physically deleted by recycling the memory that is, returning it to dynamic memory. After the count is adjusted by subtracting 1, the algorithm returns the status to the caller: if the pop was successful, it returns true; if the stack is empty when pop is called, it returns false. The operations for pop stack are traced in Figure.

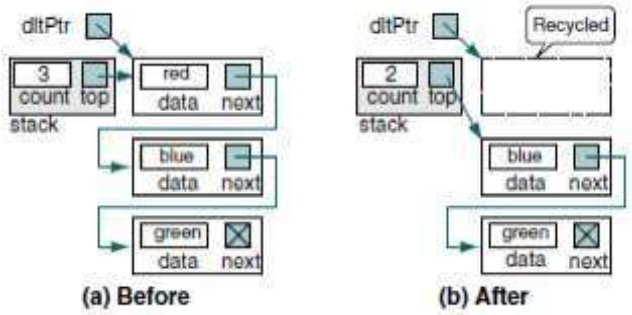


Fig.: Pop Stack Example

Application of Stack

Stack applications can be classified into four broad categories:

I. reversing data (Conversion of decimal number into binary number)
II. parsing data (Parenthesis matching)
III. postponing data usage (Expression conversion and evaluation)
IV. backtracking steps (Goal seeking and the eight queens problem)

Expression conversion and stack

I. Need for expression conversion

One of the disadvantages of the infix notation is that we need to use parentheses to control the evaluation of the operators. We thus have an evaluation method that includes parentheses and two operator priority classes. In the postfix and prefix notations, we do not need parentheses; each provides only one evaluation rule.

Although some high-level languages use infix notation, such expressions cannot be directly evaluated. Rather, they must be analyzed to determine the order in which the expressions are to be evaluated.

II. What is Polish Notation?

Conventionally, we use the operator symbol between its two operands in an arithmetic expression.

A+B C–D*E A*(B+C)

We can use parentheses to change the precedence of the operators. Operator precedence is pre-defined. This notation is called INFIX notation. Parentheses can change the precedence of evaluation. Multiple passes required for evaluation. Named after Polish mathematician Jan Named after Polish mathematician Jan Lukasiewicz

Reverse Polish (POSTFIX) notation refers to the notation in which the operator symbol is placed after its two operands.

AB+ CD* AB*CD+/

Polish PREFIX notation refers to the notation in which the operator symbol is placed before its two operands.

+AB *CD /*AB-CD

III. Advantages Polish notations over infix expression

a. No concept of operator priority.
b. Simplifies the expression evaluation rules.
c. No need of any parenthesis, Hence no ambiguity in the order of evaluation.
d. Evaluation can be carried out using a single scan over the expression string.

IV. Conversion of infix to postfix.

We can also use a stack to convert an expression in standard form (otherwise known as infix) into postfix. We will concentrate on a small version of the general problem by allowing only the operators +, *, (,), and insisting on the usual precedence rules. We will further assume that the expression is legal. Suppose we want to convert the infix expression

a + b * c + (d * e + f) * g

into postfix. A correct answer is a b c * + d e * f + g * +.

When an operand is read, it is immediately placed onto the output. Operators are not immediately output, so they must be saved somewhere. The correct thing to do is to place operators that have been seen, but not placed on the output, onto the stack. We will also stack left parentheses when they are encountered. We start with an initially empty stack.

If we see a right parenthesis, then we pop the stack, writing symbols until we encounter a (corresponding) left parenthesis, which is popped but not output.

If we see any other symbol (+, *, (), then we pop entries from the stack until we find an entry of lower priority. One exception is that we never remove a (from the stack except when processing a). For the purposes of this operation, + has lowest priority and (highest.

When the popping is done, we push the operator onto the stack.

Finally, if we read the end of input, we pop the stack until it is empty, writing symbols onto the output.

The idea of this algorithm is that when an operator is seen, it is placed on the stack. The stack represents pending operators. However, some of the operators on the stack that have high precedence are now known to be completed and should be popped, as they will no longer be pending. Thus prior to placing the operator on the stack, operators that are on the stack, and which are to be completed prior to the current operator, are popped.

This is illustrated in the following table:

Expression	Stack When Third Operator Is Processed	Action
a*b-c+d	-	- is completed; + is pushed

a/b+c*d	+	Nothing is completed; * is pushed
a-b*c/d	- *	*is completed; / is pushed
a-b*c+d	- *	*and - are completed; + is pushed

Parentheses simply add an additional complication. We can view a left parenthesis as a high-precedence operator when it is an input symbol (so that pending operators remain pending) and a low-precedence operator when it is on the stack (so that it is not accidentally removed by an operator). Right parentheses are treated as the special case.

To see how this algorithm performs, we will convert the long infix expression above into its postfix form. First, the symbol a is read, so it is passed through to the output.

Then + is read and pushed onto the stack. Next b is read and passed through to the output.

The state of affairs at this juncture is as follows:

Next, a * is read. The top entry on the operator stack has lower precedence than *, so nothing is output and * is put on the stack. Next, c is read and output. Thus far, we have

The next symbol is a +. Checking the stack, we find that we will pop a * and place it on the output; pop the other +, which is not of lower but equal priority, on the stack; and then push the +.

The next symbol read is a (. Being of highest precedence, this is placed on the stack. Then d is read and output.

We continue by reading a *. Since open parentheses do not get removed except when a closed parenthesis is being processed, there is no output. Next, e is read and output.

The next symbol read is a +. We pop and output * and then push +. Then we read and output f.

Now we read a), so the stack is emptied back to the (. We output a +.

We read a * next; it is pushed onto the stack. Then g is read and output.

The input is now empty, so we pop and output symbols from the stack until it is empty.
As before, this conversion requires only O(N) time and works in one pass through the input. We can add

subtraction and division to this repertoire by assigning subtraction and addition equal priority and multiplication and division equal priority. A subtle point is that the expression a - b - c will be converted to a b - c - and not a b c - -. Our algorithm does the right thing, because these operators associate from left to right. This is not necessarily the case in general, since exponentiation associates right to left: $2^{2^3} = 2^8 = 256$, not $4^3 = 64$. We leave as an exercise the problem of adding exponentiation to the repertoire of operators.

Let's work on one more example before we formally develop the algorithm.

A + B * C – D / E converts to A B C * + D E / -

The transformation of this expression is shown in following figure. Because it uses all of the basic arithmetic operators, it is a complete test.

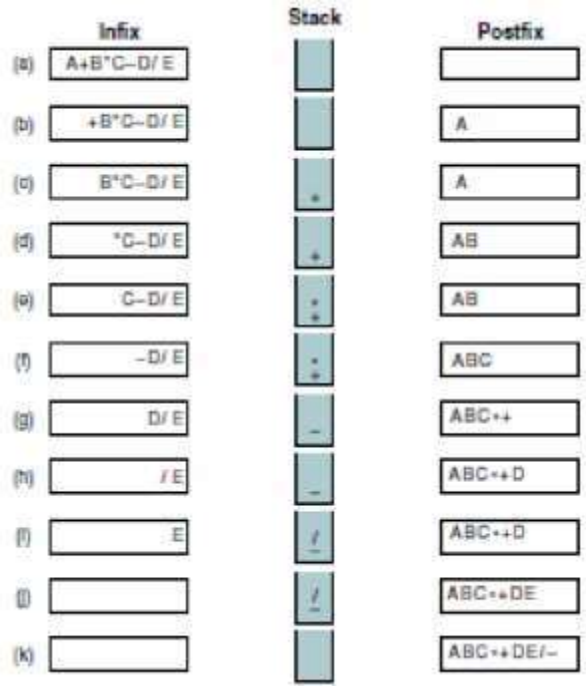


Fig.: Infix Transformation

We begin by copying the first operand, A, to the postfix expression. See Figure (b). The add operator is then pushed into the stack and the second operand is copied to the postfix expression. See Figure (d). At this point we are ready to insert the multiply operator into the stack. As we see in Figure (e), its priority is higher than that of the add operator at the top of the stack, so we simply push it into the stack. After copying the next operand, C, to the postfix expression, we need to push the minus operator into the stack. Because its priority is lower than that of the multiply operator, however, we must first pop the multiply and copy it to the postfix expression. The plus sign is now popped and appended to the postfix expression because the minus and plus have the same priority. The minus is then pushed into the stack. The result is shown in Figure (g). After copying the operand D to the postfix expression, we push the divide operator into the stack because it is of higher priority than the minus at the top of the stack in Figure (i).

After copying E to the postfix expression, we are left with an empty infix expression and two operators in the stack. See Figure (j). All that is left at this point is to pop the stack and copy each operator to the postfix expression. The final expression is shown in Figure (k).

ALGORRITHMS/PESUDOCODE :

We are now ready to develop the algorithm. We assume only the operators shown below. They have been adapted from the standard algebraic notation.

Priority 2: * /

Priority 1: + -

Priority 0: (

- **Convert Infix expression to Postfix expression**

Infix_TO_ Postfix (Input Expression)

```
//Convert infix Expression to postfix.
//Pre Expression is infix notation that has been edited
//to ensure that there are no syntactical errors
//Post postfix Expression has been formatted as a string
Return postfix Expression
Step 1: create Stack (stack)
Step 2: loop (for each character in Expression)
        i. if (character is open parenthesis)
                i. pushStack (stack, character)
        ii. elseif (character is close parenthesis)
                a. popStack (stack, character)
                b. loop (character not open parenthesis)
                        i. concatenate character to postFixExpr
                        ii. popStack (stack, character)
                  c. end loop
        iii. elseif (character is operator) //Test priority of token to token at top of stack
                a. stackTop (stack, topToken)
                b. loop (not emptyStack (stack) AND priority(character) <= priority(topToken))
                        i. popStack (stack, tokenOut)
                        ii. concatenate tokenOut to postFixExpr
                        iii. stackTop (stack, topToken)
                c. end loop
                d. pushStack (stack, token)
        iv. else  // Character is operand
                a. Concatenate token to postFixExpr
        v. end if
Step 3: end loop    //Input Expression empty.
//Pop stack to postFix
Step 4: loop (not emptyStack (stack))
        i. popStack (stack, character)
        ii. concatenate token to postFixEx
Step 5: end loop
Step 6: return postFix
//end inToPostFix
```

- **Evaluation postfix and infix with example**

Now let's see how we can use stack postponement to evaluate the postfix expressions we developed earlier. For example, given the expression shown below,

A B C + *

and assuming that A is 2, B is 4, and C is 6, what is the expression value?

The first thing you should notice is that the operands come before the operators. This means that we will have to postpone the use of the operands this time, not the operators. We therefore put them into the stack. When we find an operator, we pop the two operands at the top of the stack and perform the operation. We then push the value back into the stack to be used later.

Following figure traces the operation of our expression. (Note that we push the operand values into the stack, not the operand names. We therefore use the values in the figure.)

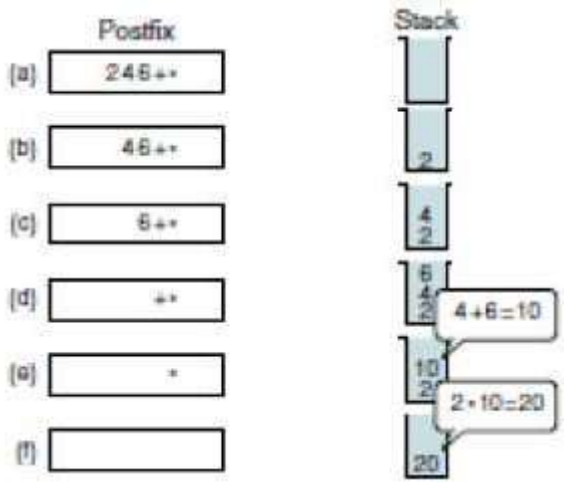


Fig.: Evaluation of Postfix expression

When the expression has been completely evaluated, its value is in the stack.
The algorithm is shown below
Evaluation of Postfix Expressions
postFixEvaluate (input Postfix _expr)
//This algorithm evaluates a postfix expression and returns its value.
//Pre a valid expression
//Post postfix value computed
//Return value of expression
Step 1 :createStack (stack)
Step 2 : loop (for each character)
a. if (character is operand)
i. pushStack (stack, character)
b.2 else
i. popStack (stack, oper2)
ii. popStack (stack, oper1)
iii. operator = character
iv. set value to calculate (oper1, operator, oper2)
v. pushStack (stack, value)
c. end if
Step 3: end loop
Step 4 : popStack (stack, result)
Step 5: return (result)
//end postFixEvaluate

Test cases/validation

Validation

I. If Stack Empty	Display message “Stack Empty”
II. If memory not available	Display message “memory not available”
III Parenthesis matching	Display appropriate message open/close parenthesis missing

Infix to postfix /prefix Test Cases

INPUT:	**POSTFIX OUTPUT:**
(A+B) * (C-D)	AB+CD-*
A$B*C-D+E/F/(G+H)	AB$C*D-EF/GH+/+
((A+B)*C-(D-E))$(F+G)	AB+C*DE—FG+$
A-B/(C*D$E)	ABCDE$*/-
A^B^C	ABC^^
INPUT:	**PREFIX OUTPUT:**
(A+B) * (C-D)	*+AB-CD
A$B*C-D+E/F/(G+H)	+-*$ABCD//EF+GH
((A+B)*C-(D-E))$(F+G)	$-*+ABC-DE+FG
A-B /(C*D$E)	-A/B*C$DE
A^B^C	^A^BC

FAQ:

1. What is data structure?
2. Types of data structure?
3. Examples of linear data structure & Non-linear data structure?
4. What are the operations can implement on stack?
5. Explain recursion using stack.
6. Explain how a string can be reversed using stack.
7. How does a stack similar to list? How it is different?
8. List advantages and disadvantages of postfix and prefix expression over infix expression.

Assignment 3 - Circular Queue

AIM : **Implementation of Circular Queue using Array**

DETAILED PROBLEM STATEMENT:

Implement Circular Queue using Array. Perform following operations on it.
a) Insertion (Enqueue)
b) Deletion (Dequeue)
c) Display
(Note: Handle queue full condition by considering a fixed size of a queue.)

OBJECTIVE:

1. To understand the concept of Circular Queue.
2. Implement Circular queue using sequential organization.
3. Understand and apply the queue full and queue empty conditions.

OUTCOME:

1. Understand the conditions of Circular Queue.
2. Implement different operations like insert and delete on the circular queue.
3. Display contents of queue after every operation.

THEORY:

Queue:

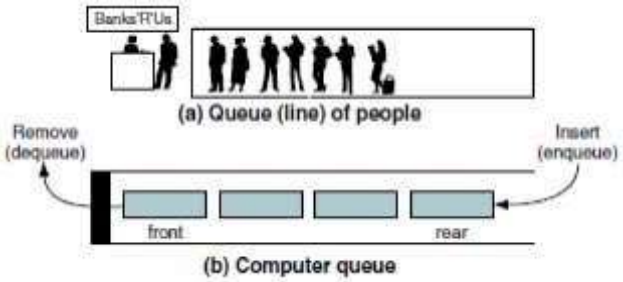
i. Concept

Queues are more common data objects found in computer algorithms. It is a special case of more general data object, an ordered or linear list. It can be defined as an ordered list, in which all insertions are allowed at one end, called as 'rear' & all deletions are made at other end, called as 'front' i.e. First element inserted is outputted first (**F**irst **I**n **F**irst **O**ut or **FIFO**).
Queue can be represented mathematically as :
S = {a1, a2, …,an} where a1 is the first element & an is last element.& ai is ahead of ai+1 , 1=< i < n.

ii. Definition of Queue:

Queue is a linear list in which data can be inserted at one end, called the rear, and deleted from the other end, called the front. It is a first in–first out (FIFO) restricted data structure
Figure 1 shows two representations of a queue: one a queue of people and the other a computer queue. Both people and data enter the queue at the rear and progress through the queue until they arrive at the front. Once they are at the front of the queue, they leave the queue and are served.

Fig.1: Queue Concept



iii. Terminology with Queue

An abstract data type (ADT) consists of a data structure and a set of **primitives**

a. **Initialise** creates/initialises the queue
b. **Enqueue** adds a new element
c. **Dequeue** removes a element
d. **IsEmpty** reports whether the queue is empty
e. **IsFull** reports whether the queue is full
f. **Destroy** delete the contents of the queue

iv. Diagram (horizontal view)

Given below is a pictorial representation of queue.

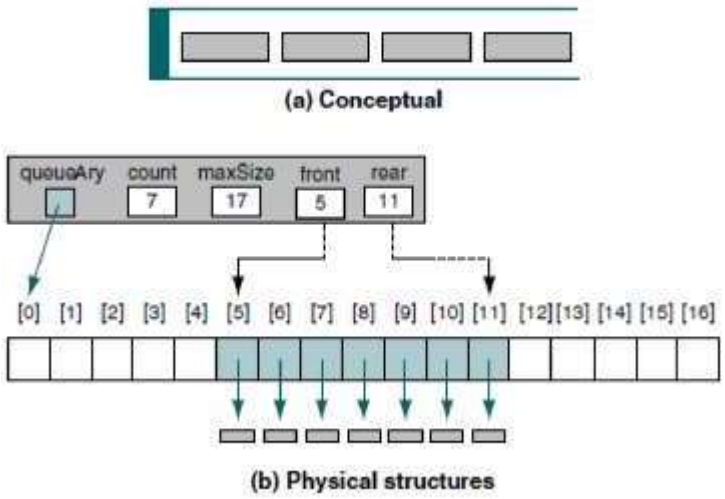


Fig.2: Queue stored in an array

An enqueue to an empty queue is placed in the first element of the array, which becomes both the front and the rear. Subsequent enqueues are placed at the array location following rear; that is, each enqueue stores the queue data in the next element after the current queue rear. Thus, if the last element in the queue is stored at array location 11, the data for the next enqueue is placed in element 12.

Dequeues take place at the front of the queue. As an element is deleted from the queue, the queue front is advanced to the next location; that is, queue front becomes queue front plus one. Figure 2(a) shows a conceptual queue; Figure 2(b) shows a physical queue after it has been in operation for a period of time. At the point shown, the data have migrated from the front of the array to its center.

When we implement a queue in an array, we use indexes rather than pointers. Thus, the front of the queue in figure 3 is 5 and the rear is 16.

The definition of a full queue changes when we implement a queue in an array. A full queue is defined as every location filled. Because arrays have a finite number of elements, we can determine that the queue is full by testing the queue count against the maximum number of elements.
When data arrive faster than the queue service time, the queue tends to advance to the end of the array. This leads to the situation where the last element in the array is occupied but the queue is not full because there are empty elements at the beginning of the array. This situation is shown in Figure F-3.

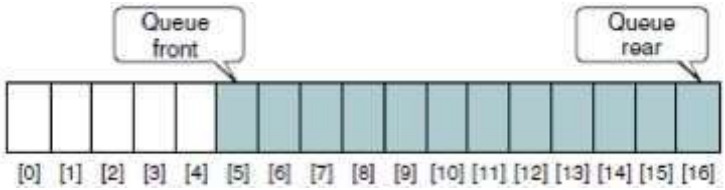


Fig.3: Array Queue filled with Last element filled

When the data are grouped at the end of the array, we need to find a place for the new element when we enqueue data. One solution is to shift all of the elements from the end to the beginning of the array. For example, in Figure 3 element 5 can be shifted to element 0, element 6 to 1, 7 to 2, and so forth until element 16 is shifted to 11.

v. Circular Queue

A more efficient alternative is to use a circular array. In a circular array, the last element is logically followed by the first element. This is done by testing for the last element and, rather than adding one, setting the index to zero. Given our array queue above, the next element after 16 is 0. A circular queue is shown in Figure 4.
With this understanding of a circular queue, we are ready to rewrite the abstract data type. The only difference in the calling sequence between the two implementations is the addition of a maximum queue size in the create queue function.

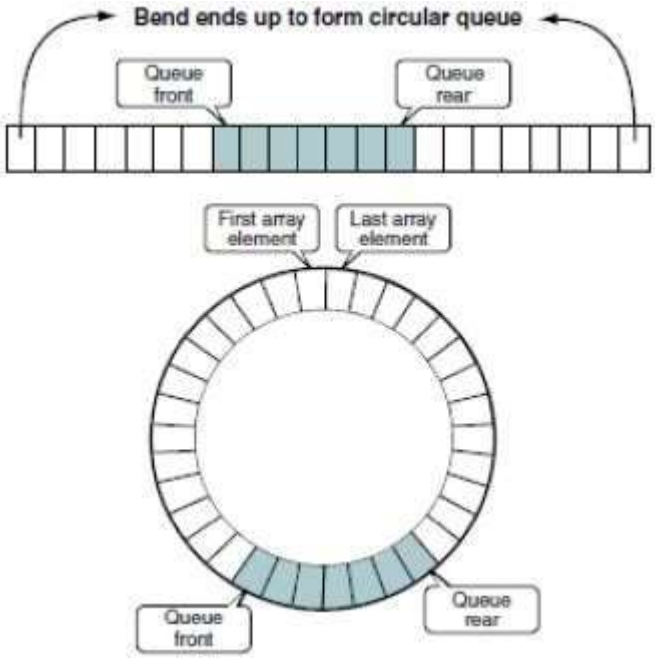


Fig.4: Circular Queue

Figure 5 illustrates the operation of this code on an example. The first figure 5 shows an empty queue with first = rear = 0. The second figure shows the queue after the integer 10 is inserted. The Third figure shows the queue when 20, 30, 40, 50, and 60 have been inserted. The fourth figure shows the queue after 70 is inserted. Notice that, although one slot remains empty, the queue is now full because Queue::Insert will not permit another element to be inserted. If it did permit an insertion at this stage, rear and front would be the same. This is the condition that Queue: Delete checks to determine whether the queue is empty! This would make it impossible to distinguish between the queue being full and being empty. The fifth figure shows the queue after two integers (10 and 20) are deleted. The last figure shows a full queue after the insertion of integers 80 and 90.

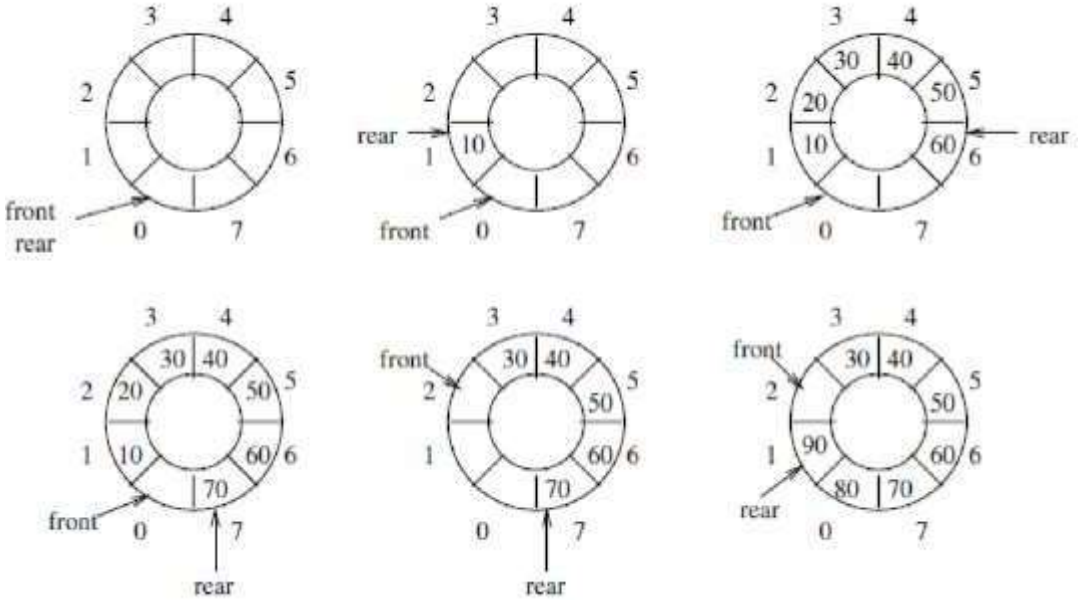


Fig. 5 Demonstration of queue in a circular array

vi. Applications of Queues:

1. Multiuser, multiprogramming environment job scheduling
2. Reversing stack using queue
3. queue Simulation, all types of customer service(like railway reservation) centers are designed using the concept of queues
4. Categorizing data

ALGORITHMS/ PESUDOCODE:

Insertion in Circular queue

There are three scenario of inserting an element in a queue.

1. If (rear + 1)%maxsize = front, the queue is full. In that case, overflow occurs and therefore, insertion can not be performed in the queue.
2. If rear != max - 1, then rear will be incremented to the mod(maxsize) and the new value will be inserted at the rear end of the queue.
3. If front != 0 and rear = max - 1, then it means that queue is not full therefore, set the value of rear to 0 and insert the new element there.

- **Insert_CQ(VAL)**

```
Step 1: if (rear+1) %MAX = front
            Write "OVERFLOW "
            Execute step 4
Step2: [End Of if]
Step 3: if Front = -1 and Rear = -1
          i. set front = rear= 0
          ii. else if rear = MAX - 1 and front ! = 0
                    set rear = 0
         iii. else
                    1.set rear= (rear + 1) % MAX
Step 4: end  if

Step 5: set QUEUE[rear] = VAL
Step 6: Exit
```

- **Deletion in circular queue**

To delete an element from the circular queue, we must check for the three following conditions.
1. If front = -1, then there are no elements in the queue and therefore this will be the case of an underflow condition.
2. If there is only one element in the queue, in this case, the condition rear = front holds and therefore, both are set to -1 and the queue is deleted completely.
3. If front = max -1 then, the value is deleted from the front end the value of front is set to 0.
4. Otherwise, the value of front is incremented by 1 and then delete the element at the front end.

Delete_CQ():

```
Step 1: if  (isempty_Q())
                    Write " UNDERFLOW "
                    Goto Step 4
Step 2: End if
Step 3:  set VAL = QUEUE[front]
Step 4: if front =rear
                    i. set front = rear = -1
Step 5:  else
             i. if  front = MAX -1
                              a.  set front= 0
            ii.  else
                    a.  set  front = front + 1
           iii. end if
Step 6: end if
Step 7: exit
```

Test cases/validation :

Addition of elements in :

Token	Insert More?
A	Y
B	Y
C	Y
D	Y
E	N

********** OUTPUT **********

Display of elements in Queue :

After creation	front	rear	elements
At front :	0	0	queue is empty
At rear :	0	0	queue is empty
All from front :	0	0	queue is empty
All from rear :	0	0	queue is empty

After Additions	front	rear	elements
At front :	0	5	A
At rear :	0	5	E
All from front :	0	5	A B C D E
All from rear :	0	5	E D C B A

****

6. Test Cases:

1. Create an empty circular queue
2. Display front element
3. Display rear element
4. Display all elements of queue starting from front
5. Display all elements of queue starting from rear
6. Add element A in queue
7. Add elements B, C, D, and E in queue
8. Display front element
9. Display rear element
10. Delete an element from queue
11. Display front element
12. Display rear element
13. Display all elements of queue starting from front
14. Display all elements of queue starting from rear

Assignment 4 – Expression Tree and Tree Traversals

AIM : To construct an expression tree and perform recursive and non-recursive traversals.

DETAILED PROBLEM STATEMENT :

To construct an Expression Tree from postfix and prefix expression. Perform recursive and non- recursive in-order, pre-order and post-order traversals.

OBJECTIVE:

1. To understand non-linear data structure: Tree.
2. To implement non-linear data structure: Tree.
3. To understand applications of Tree.

OUTCOME:

1. To implement expression tree.
2. To perform recursive and non-recursive traversals.
3. To understand applications of tree

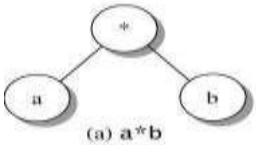
THEORY:

Definition of Expression Tree:
For an Algebraic expressions such as: 3 + (4 * 6) or a + b * c

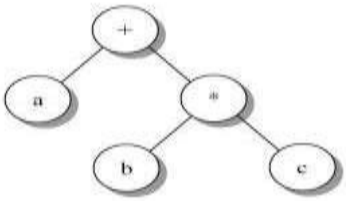
The terminal nodes (leaves) of an expression tree are the variables or constants in the expression (3,4, 6, or a, b, c). The non-terminal nodes of an expression tree are the operators (+, -, *, /).

Notice that the parentheses which appear in equation do not appear in the tree.

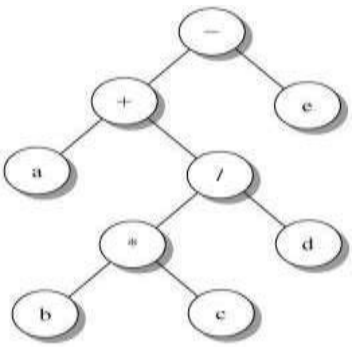
Examples of Expression Tree:



(a) a*b



(b) a + b* c



(c) a + b*c/d − e

Applications of Expression Tree:

1. Expression conversion i.e. to convert infix expression to postfix or prefix and vice a versa.
2. Evaluation of an infix, prefix and postfix expression.

Data Structures to Implement Expression Tree:

An arithmetic expression consists of operands (variable or constant) and operators. To construct an expression binary tree Stack is an appropriate data structure.
In construction of expression tree process two stack as an array are declared one as operand stack to store operand as node and other as operator stack to store operator as a node for an infix or prefix or postfix expression.

Different type of traversals with example:

To traverse a non-empty binary tree in **pre-order**:

1. Visit the root.
2. Traverse the left sub tree.
3. Traverse the right sub tree.

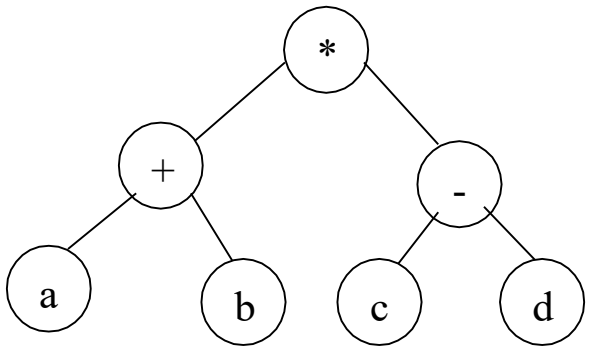
To traverse a non-empty binary tree in **in-order**:

1. Traverse the left sub tree.
2. Visit the root.
3. Traverse the right sub tree.

To traverse a non-empty binary tree in **post-order**,

1. Traverse the left sub tree.
2. Traverse the right sub tree.
3. Visit the root.

Consider an example of an expression tree:



1. Pre-order (Prefix) : * + a b – c d

2. In-order (Infix) : a + b * c - d

3. Post-order (postfix) : a b + c d - *

ADT of Expression Tree:

Structure: Binary_Tree ( ) is

Objects: a finite set of nodes either empty or consisting of a root node, left_Binary_Tree, and right_Binary_Tree.

Functions:
For all bt, bt1, bt2 ∈ Binary_Tree, item ∈ element
Binary_Tree Create( )::= creates an empty binary tree
Boolean IsEmpty(bt)::= if (bt==empty binary tree) return TRUE else return FALSE

Binary_Tree MakeBT(bt1, item, bt2)::= return a binary tree whose left sub tree is bt1, whose right sub tree is bt2, and whose root node contains the data item

Binary_Tree Lchild(bt)::= if (IsEmpty(bt)) return error else return the left sub tree of bt
Element Data(bt)::= if (IsEmpty(bt)) return error else return the data in the root node of bt
Binary_Tree Rchild(bt)::= if (IsEmpty(bt)) return error else return the right sub tree of bt
END

ALGORITHMS/PESUDOCODE :

- **Create Expression Tree:**

For constructing expression tree we use a stack. We loop through input expression and do following for every character.

Create_ET(input expression)

//For constructing expression tree we use a stack.
//We loop through input expression and do following for every character.
Step 1: Read an infix expression in an array and convert it to postfix/prefix
Step 2: While not of postfix expression

i. If character is operand push that into stack

ii. If character is operator
    a. New BNode
    b. BNode-> data = operator assign data part of the node
    c. BNode- >Rchild =pop stack top // assign right child first top symbol of the stack
    d. BNode ->Lchild = pop stack top2 // assign left child as second top symbol of the stack
    e.. push current node again.

Step 3: end of while
Step 4: root of the tree = pop stack
Step 5: End of Create_ET

- **In-order Traversal (Recursive)**

1. Traverse the left sub tree, i.e. call In-order (left – sub-tree)
2. Visit the root.
3. Traverse the right sub tree, i.e. call In-order (right – sub-tree)

- **Pre-order Traversal (Recursive)**

1. Visit the root.
2. Traverse the left sub tree, i.e. call Pre-order (left – sub-tree)
3. Traverse the right sub tree, i.e. call Pre-order (right – sub-tree)

- **Post-order Traversal (Recursive)**

1. Traverse the left sub tree, i.e. call Post-order (left – sub-tree)
2. Traverse the right sub tree, i.e. call Post-order (right – sub-tree)
3. Visit the root.

- **In-order Traversal (Non- recursive)**

1. Create an empty stack S.
2. Initialize current node as root
3. Push the current node to S and set current = current->left until current is NULL
4. If current is NULL and stack is not empty then
    a. Pop the top item from stack.
    b. Print the popped item, set current = poped_item->right
    c. Go to step 3.
5. If current is NULL and stack is empty then stop.

- **Pre-order Traversal (Non – recursive)**

1 Create an empty stack Sand push root node to stack.
2. Do following while S is not empty.

    a. Pop an item from stack and print it.
    b. Push right child of popped item to stack
    c. Push left child of popped item to stack

3. If stack is empty, then stop.

- **Post-order Traversal (Non – recursive)**

1. Create an empty stack S
2. Do following while root is not NULL
    a. Push root's right child and then root to stack.
    b. Set root as root's left child.
3. Pop an item from stack and set it as root.
    a. If the popped item has a right child and the right child is at top of stack, then

remove the right child from stack, push the root back and set root as root's right child.

b. Else print root's data and set root as NULL.

4. Repeat steps 3.a and 3.b while stack is not empty, then stop.

Test Cases / Validation:

Validations:

1. Identification of symbol as operator or operand.
2. Postfix and prefix expressions can be of char or int data type

Test Cases

1. Test for valid prefix or postfix expression
2. no of operator and operand relationship check

Sr. No.	Sample Infix Expression	Postfix	Prefix
1.	A+B*C	ABC*+	+A*CB
2.	A*B-C	AB*C-	-*ACB
3.	A^B-C	AB^C-	-^ABC
4.	A+B*C^E	ABCE^*+	+A*B^CE
5.	A-B*C+A	ABC*-A+	-+A*BCA
6.	(A+B)/(C+D)^E^F-D*F-D	AB+CD+EF^^/DF*-D-	/+AB+CD^EF*DFD
7.	A+B+C	AB+C+	++ABC
8.	A*B/C	AB*C/	/*ACB
9.	A^B^C	ABC^^	^A^BC

FAQS:

1. What is tree? What are properties of trees?
2. What is Binary tree, Binary search tree, Expression tree & General tree?
3. What are the members of structure of tree & what is the size of structure?
4. What are rules to construct binary tree?
5. What is preorder, post-order, in-order traversal?
6. Difference between recursive & Non-recursive traversal?
7. What are rules to construct binary search tree?
8. What are rules to construct expression tree?
9. How binary tree is constructed from its traversals?

Assignment 5 - Binary search tree

AIM : **Implementation of binary search tree**

DETAILED PROBLEM STATEMENT :

Implement binary search tree and Perform following operations:

a) Insert,
b) delete,
c) search
d) display tree (traversal)
e) display – depth of tree
f) display - mirror image
e) create a copy
f) Display all parent nodes with their child nodes
g) display tree level wise
h) display leaf nodes.

(Note: Insertion, Deletion, Search and Traversal are compulsory, from rest of operations, perform Any three)

OBJECTIVE:

1. To understand difference between Binary Tree and Binary Search Tree.
2. To implement Binary Search Tree
3. To understand applications of Binary Search Tree

OUTCOME :

1. To construct binary search tree.
2. To perform different operations on it.
3. To do traversals on a tree.

THEORY:

Definition of binary search tree

A binary tree in which each internal node x stores an element such that the element stored in the left subtree of x are less than or equal to x and elements stored in the right subtree of x are greater than or equal to x. This is called binary-search-tree property.

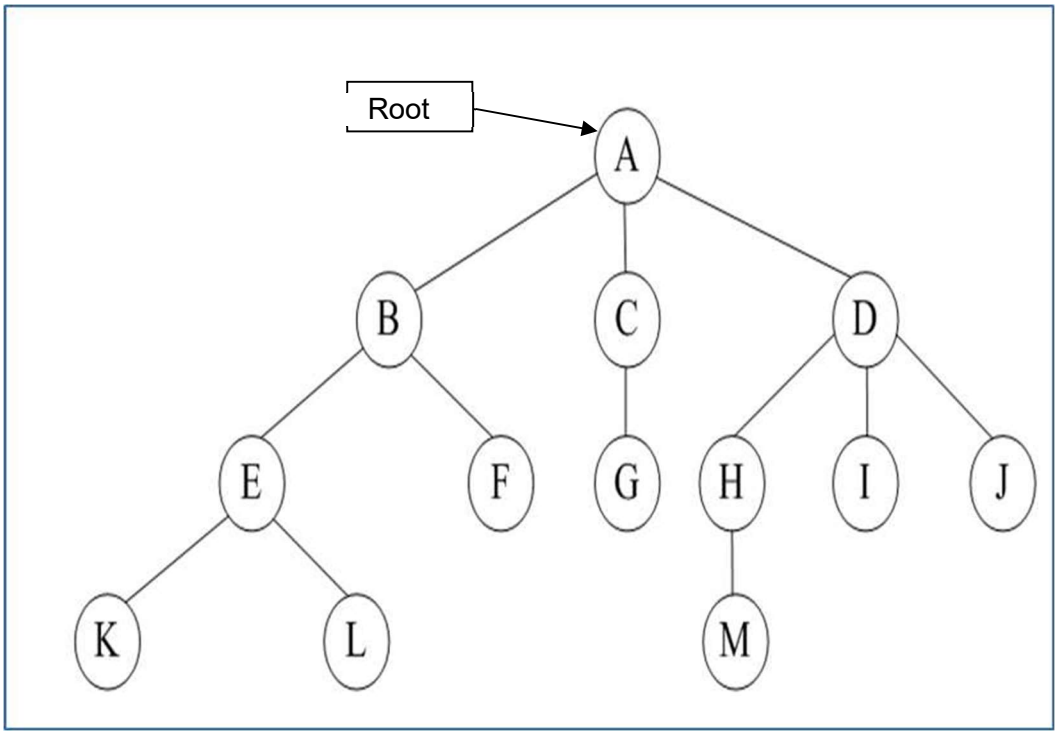


Fig 1. Binary tree

Binary Search Tree :

- **Concept:**

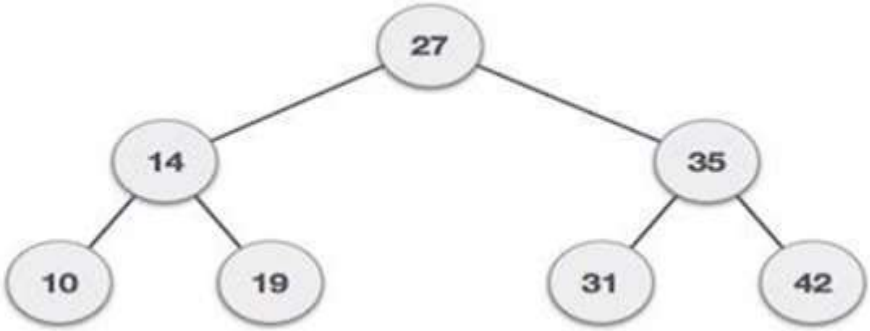
A Binary Search Tree is a type of binary tree data structure in which the nodes are arranged in order, hence also called as "ordered binary tree". It is a node-based data structure, which provides an efficient and fast way of sorting, retrieving, searching data.

- **Definition:**

Binary search tree is a node-based binary tree data structure which has the following properties:

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

- Type of data structure: It is a non-linear type of data structure.



- ADT:

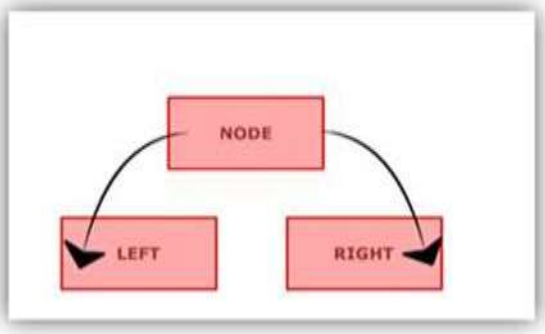
It is a special kind of binary tree which performs following operations:

1. Insert (X)::= depending on the root element X get inserted either to left or to right and new BST

will be formed.

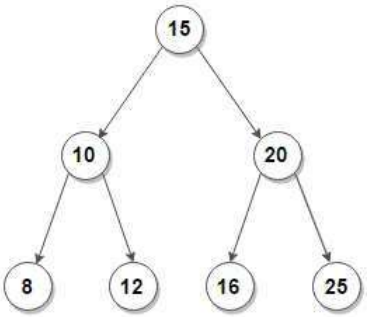
2. Search (X)::= the element is searched either to the left or to the right half of the tree.
3. Delete (X)::= the element is deleted and new BST will be formed.
4. Traversal (root)::= it will return all traversals of BST.

- Realization of ADT:



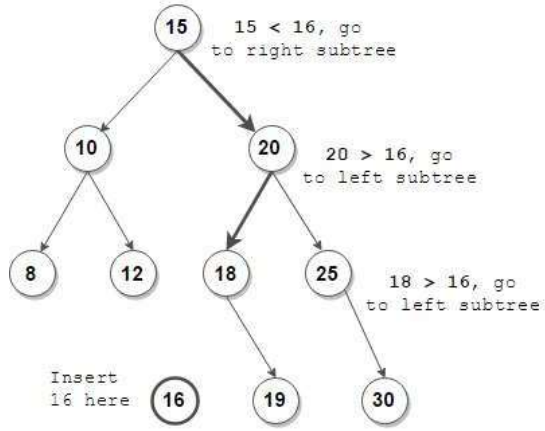
Data that represents value stored in the node.
Left that represents the pointer to the left child.
Right that represents the pointer to the right child.

- Example with basic operations:



Binary Search Tree

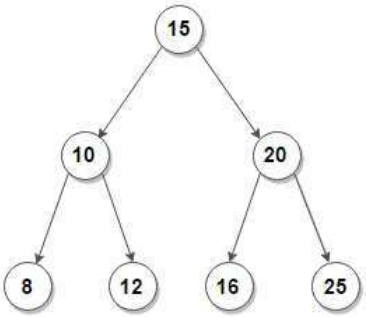
Insert: insert 16 in the above BST



Insert (root, 16)

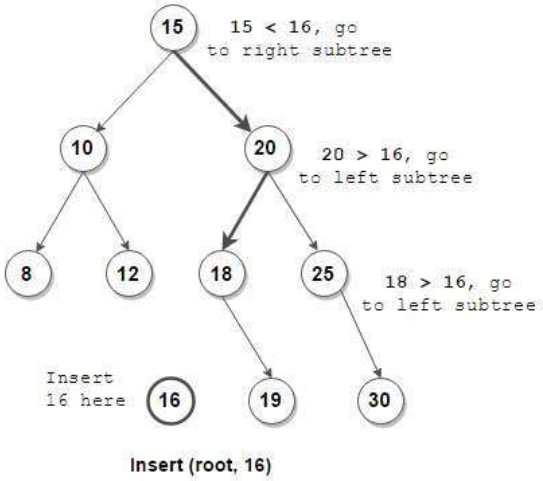
Right that represents the pointer to the right child.

- **Example with basic operations:**



Binary Search Tree

Insert: insert 16 in the above BST



Insert (root, 16)

Traversals:

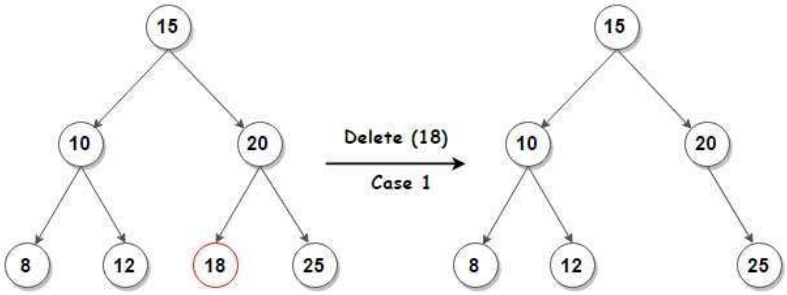
Inorder Traversal: 8 10 12 15 16 18 19 20 25 26

Preorder Traversal: 15 10 8 12 20 18 16 19 25 30

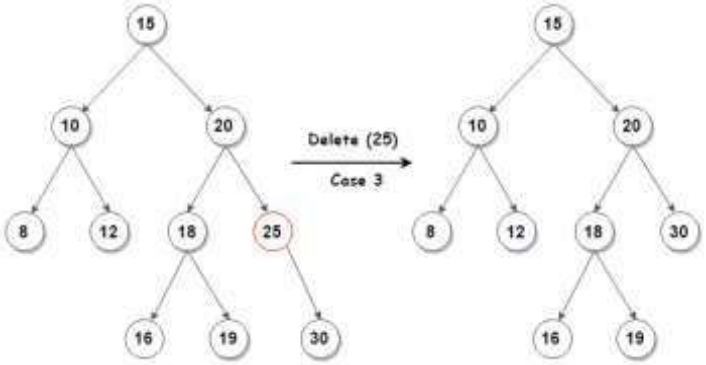
Postorder Traversal: 8 12 10 16 19 18 30 25 20 15

Delete:

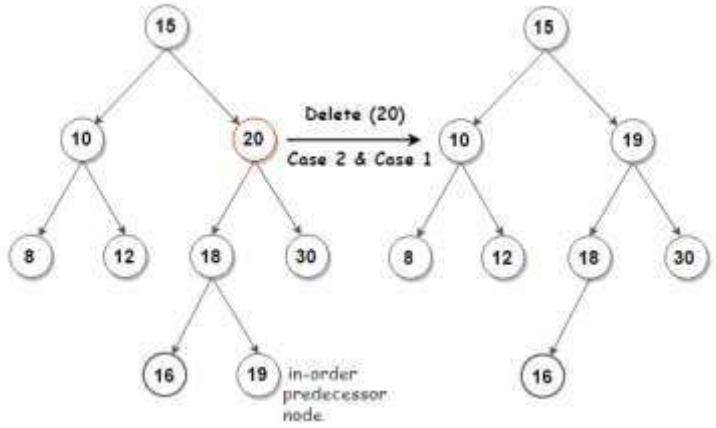
1) **Node to be deleted is leaf:** Simply remove from the tree.



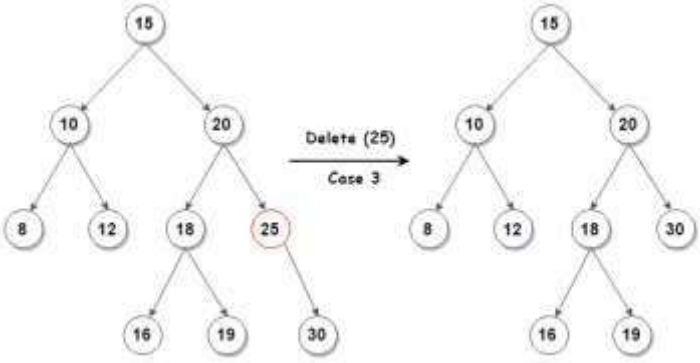
2) **Node to be deleted has only one child:** Copy the child to the node and delete the child



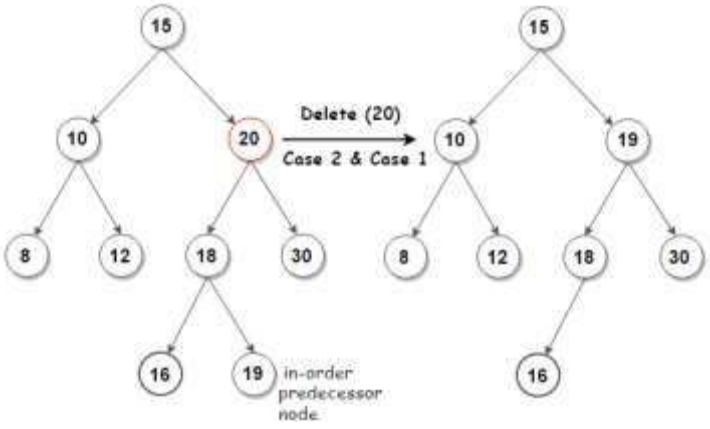
3) **Node to be deleted has two children:** Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.



4) **Node to be deleted has only one child:** Copy the child to the node and delete the child



5) **Node to be deleted has two children:** Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.



- **Insert Node:**

Insert(Root, Node)

Root is a variable of type structure represent Root of the Tree. Node is a variable of type structure represent new Node to insert in a tree.
Step 1: Repeat Steps 2, 3 & 4 Until Node do not insert at appropriate position.
Step 2: If Node Data is Less that Root Data & Root Left Tree is NULL
Then insert Node to Left.
Else Move Root to Left
Step 3 : Else If Node Data is Greater that Equal that Root Data & Root Right Tree is NULL
Then insert Node to Right.
Else Move Root to Right.
Step 4: Stop.

- **Search Node:**

Search (Root, Item)
Root is a variable of type structure represent Root of the Tree. Item is the element to search. This function search an element in a Tree.
Step 1: Repeat Steps 2,3 & 4 Until element Not find && Root != NULL
Step 2: If item Equal to Root Data
Then print message item present.
Step 3 : Else If item Greater that Equal that Root Data
Then Move Root to Right.
Step 4 : Else Move Root to Left.
Step 5: Stop.

- **Delete Node:**

Dsearch(Root, Item)

Root is a variable of type structure represent Root of the Tree. Item is the element to delete. Stack is a pointer array of type structure. PTree(Parent of Searched Node),Tree(Node to be deleted), RTree(Pointg to Right Tree),Temp are pointer variable of type structure;

Step 1: Search an Item in a Binary Search Tree
Step 2: If Root = = NULL Then Tree is NULL
Step 3: Else //Delete Leaf Node
If Tree->Left = = NULL && Tree->Right = = NULL
Then a) If Root = = Tree Then Root = NULL;
b) If Tree is a Right Child PTree->Right=NULL;
Else PTree->Left=NULL;
Step 4: Else // delete Node with Left and Right children
If Tree->Left != NULL && Tree->Right != NULL
Then a) RTree=Temp=Tree->Right;
b) Do steps i && ii while Temp->Left !=NULL
i) RTree=Temp;
ii) Temp=Temp->Left;
c)RTree->Left=Temp->Right;
d) If Root == Tree//Delete Root Node
Root=Temp;
e)If Tree is a Right Child PTree->Right=Temp;
Else PTree->Left=Temp;
f)Temp->Left=Tree->Left;
g)If RTree!=Temp
Then Temp->Right = Tree->Right;
Step 5: Else //with Right child
If Tree->Right!= NULL
Then a) If Root==Tree Root = Root->Right;
b) If Tree is a Right Child PTree->Right=Tree->Right;
Else PTree->Left=Tree->Left;
Step 6: Else //with Left child
If Tree->Left != NULL
Then a) If Root==Tree Root = Root->Left;
b) If Tree is a Right Child PTree->Right=Tree->Left;
Else PTree->Left=Tree->Left;
Step 7: Stop.

- **Inorder Traversal Recursive :**

Tree is pointer of type structure.

InorderR(Tree)
Step 1: If Tree != NULL
Step 2: InorderR(Tree->Left);
Step 3: Print Tree->Data
Step 4: InorderR(Tree->Right);

- **Postorder Traversal Recursive:**

Tree is pointer of type structure.

PostorderR(Tree)
Step 1: If Tree != NULL
Step 2: PostorderR(Tree->Left);
Step 3: PostorderR(Tree->Right);
Step 4: Print Tree->Data;

- **Preorder Traversal Recursive:**

Tree is pointer of type structure.

PreorderR(Tree)
Step 1: If Tree != NULL
Step 2: Print Tree->Data;
Step 3: PreorderR(Tree->Left);
Step 4: PreorderR(Tree->Right);

- **Mirror of thetree Recursive**

ptr Mirror_BST(BST root)

Step 1: If treet != NULL
Step 2: temp =tree->Right
tree-> Right= tree->Left
tree->Left=temp
Step 3: Mirror_BST(tree->Left)
Step 4: Mirror_BST(tree->Right)

- **Height of the tree recursive :**

int height (root)
Step1 : If (root=NULL)
Display " tree is empty"
stop
Step 2: Else
Step 3: return 1 + max(height(root->left),height(root->right))

- **Level Order Traversal of a tree :**

levelorder(root)

```
//  q is QUEUE of size(1:n)
Step 1: If(T=NULL ) //empty tree
            then  write('Tree is Emty)
             return
Step 2: else          //create empty Queue
Step 3:         q = empty queue
Step 4:         enqueue(T)      //Intially root

Step 5:    while(!(isempty(q))
Step 6:    node = dequeue(q)  //front
Step 7:    write /visit (node)
Step  8:              if (node->left != NULL )
           //insert left node next level
Step 9:                enqueue(node->left)
           //insert right node of next levle
Step 10:              if (node->right != NULL)
              enqueue(node->right)
Step 11:nd Level Order
```

Validations :

1. No duplicate key is allowed in binary search tree while insertion.

Test Cases

1. Try to insert duplicate key: Message should be display not allowed.
2. In mirror image, inorder traversal should be in descending order.
3. Try to delete leaf node: it should return proper binary search tree.
4. Try to delete a node with one child: it should return proper binary search tree.
5. Try to delete a node with two child: it should return proper binary search tree.

FAQS:

1. What is Binary search tree?
2. What are the members of structure of tree & what is the size of structure?
3. What are rules to construct binary search tree?
4. How general tree is converted into binary tree?
5. What is binary threaded tree?
6. What is use of thread in traversal?

Assignment 6- Threaded Binary Tree

AIM : Implementation In-order Threaded Binary Tree (TBT)

DETAILED PROBLEM STATEMENT :

To implement In-order TBT and to perform In-order, and Pre-order traversals.

OBJECTIVE :

1. To understand construction of TBT.
2. To implement In-order TBT.
3. To understand In-order, and Pre-order traversals of TBT.
4. To understand pros/cons of TBT over Binary Tress.
5.

OUTECOME :

1. To implement In-order TBT
2. To Perform In-Order and Preorder Traversal of TBT

THEORY :

Issues with regular Binary Tree Traversals:

1. The storage space required for stack and queue is large.
2. The majority of pointers in any binary tree are NULL. For example, a binary tree with n nodes has n+1 NULL pointers and these were wasted.
3. It is difficult to find successor node (Pre-order, In-order and Post-order successors) for a given node.

Motivation for Threaded Binary Trees:

To solve these problems, one idea is to store some useful information in NULL pointers. In the traversals of regular binary tree, stack/queue is required to record the current position in order to move to right subtree after processing the left subtree. If the null links are replaced with useful information, then storing current position in order to move to right subtree after processing the left subtree on the stack / queue is not required. The binary trees which store such information in NULL pointers are called Threaded Binary Trees.

What to store in the null links?

The common convention is put predecessor/successor information. That means, if TBT to be constructed is an In-order-TBT then for a given node left pointer will contain in-order predecessor information and right pointer will contain In-order successor information. These special Pointers are called Threads.

Types of Threaded Binary Trees:

Based on above forms we get three representations for threaded binary trees .

Pre-order Threaded Binary Trees: NULL left pointer will contain Pre-order predecessor information and NULL right pointer will contain Pre-order

1. successor information.

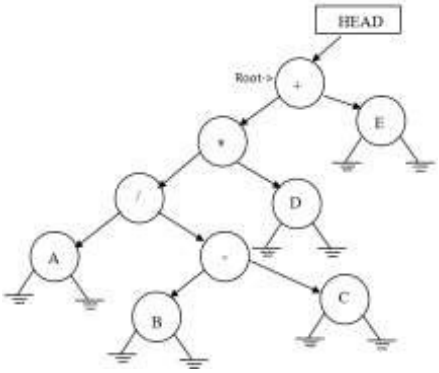
2. **In-order Threaded Binary Trees:** NULL left pointer will contain In-order predecessor information and NULL right pointer will contain In-order successor information.
3. **Post-order Threaded Binary Trees:** NULL left pointer will contain Post-order predecessor information and NULL right pointer will contain Post-order successor information.

In-Order Threaded Binary Tree

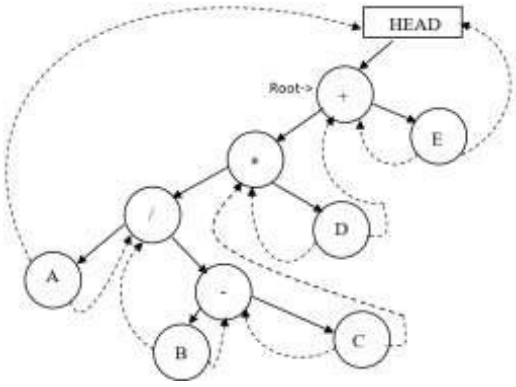
Using above concept, if right link of node p (i.e. RCHILD (p)) is null, then it can be replaced with a pointer (thread) to a node which would immediately succeed the node p (i.e. successor of p) while traversing the binary tree in In-order manner. Similarly, if left link of node p (i.e. LCHILD(p)) is null, then it can be replaced with a pointer (thread) to a node which would immediately precede the node p (i.e. predecessor of p)while traversing the binary tree in In-order manner.

For example, (Ref. figure below), let us consider a binary tree for the prefix arithmetic expression +*/A-BCDE

(Note: It is not mandatory that TBT should be constructed only by using either prefix / postfix expression.)



The tree has n=9 nodes, n-1 (i.e. 8) normal (constructional) pointers (**represented by solid directional lines**), and n+1 (i.e. 10) null links. In TBT, the n+1 null links are replaced by Threads (**represented by dotted lines**) (Ref. figure below).



If the above tree is traversed in the In-order manner, then the output would be

A / B – C * D + E----------(1)

For example, node D's left link is replaced with a pointer (Thread) to a node '*' which is an immediate predecessor of node D (Ref. In-order Traversal in 1). Node D's right link is replaced with a pointer (Thread) to a node '+' which is an immediate successor of node D (Ref. In-order Traversal in 1).

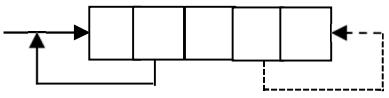
To distinguish between normal (constructional) pointers and a thread two extra one bit fields i.e. LBIT and RBIT are required in the node structure. Each node has five fields namely LBIT, LCHILD, DATA, RCHILD, RBIT and as shown below

LBIT	LCHILD	DAT A	RCHILD	RBIT

LBIT: '0' if LCHILD is a thread i.e. pointer to predecessor; '1' if LCHILD is pointer to root of LEFT binary tree
LCHILD: A pointer (address) to root node of LEFT binary tree or left thread
DATA: Atom or data item or information at node
RCHILD: A pointer (address) to root node of RIGHT binary tree or right thread
RBIT: '0' if RCHILD is a thread i.e. pointer to successor; '1' if RCHILD is pointer to root of RIGHT binary tree

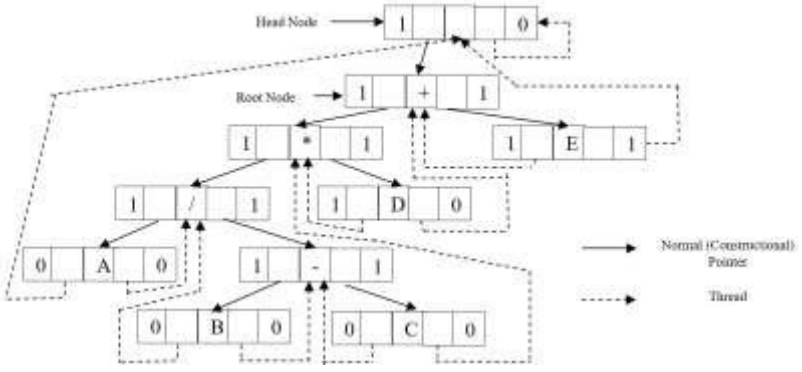
In the above figure node 'A' s left pointer and node 'E' s right pointer have been left dangling as node 'A' doesn't have any In-order predecessor and node 'E' doesn't have any In-order successor (Ref. 1). In order to have no loose Threads, all dangling pointers are connected to a node called as Head node in all the TBT's.

The empty in order threaded tree binary tree will have only one HEAD node as shown in the following diagram



Representation of the TBT in Computer's Memory by Using the Node Structure

The complete memory representation for the tree is as shown below



ALGORITHMS/ PESUDOCODE:

- **Creation of TBT**

Create_root()

```
///1.  Creat a Head node
  Step 1 :  head= getnode('$')
  Step 2 :  head->lthread=head-rthread=1
  Step 3 :  head->lptr=head->rptr= head
  //2. Create root node :
  Step 4 :  root =getnode()
  Step 5 :  root->lthread =rthread=1
  Step 6 : root->rptr=lptr= head
  Step 7 :  head->lptr=root
  Step 8 :  head->lthread= 0
//End of root create()
```

- **Insert node**

Insert(head , X)

```
Step 1: if(head= Null)
        i.   THEN  error message ('CREATE ROOT FIRST')
        ii.   (create_ root(head))
        iii.  return head
Step 2:  Parent =head ->lptr
Step 3:  Repeat through  step  4  till the successful insertion is not happens
Step 4:  Write ('ROOT IS' , parent->data)
    //take user choice were to insert
     '1.INSERT AT LEFT , 2.INSERT AT RIGHT'
    //based on the choice insert at left or right.
Step 5:  if choice is 1
          //insert  as a left chiled
         I. if (parent->lthread=1)//if no lchild
                i.    new = getnode (X)
                ii.   new->lptr= parent->lptr
                iii.   new->rptr= parent
                iv.    new->lthread=new->rthread= 1
                v.    parent-lptr=new
                vi.   parent->lthread=0
         II  Else
              i. parent= parent->lpter
 Step 7:  if  choice is 2
            //insert at right side
           I. if (parent->rthread=1)//if no right child
```

i. new = getnode (X)
ii. new-rptr= parent->rptr
iii. new->lptr= parent
iv. new->rthread=new->lthread= 1
v. parent->rpr=new
vi. parent->rthread=0

II. Else

i.parent= parent->rpter

tep 5 : end of insert

- **Non Recursive Preorder Traversal**

Non_preorder(head)

Step 1: current = head->lptr
Step 2: if current =head
i. THEN WRITE('EMPTY TREE')
ii. Exit
Step 3: while current ! =head
i. display(current->data) //process data in preorder
ii. if(current->lthread =0) //if having leftchiled process left subtree a. current=current->lptr
iii. else
//if no left subtree , go to right subtree
a. while (current->rthread=1)
i. current=current->rptr
b. current=current->rptr// if left subtree process it first
c. end while
Step 4: end while
Step 5: end of preorder

- **Non Recursive Inorder Traversal**

Non_Inroder(head)

Step 1: current =head->lptr
Step 2: if (current->lptr=head)
i. THEN WRITE('EMPTY TREE')
ii. Exit
Step 3: while (current ->lthread=0) //go to leftmost chiled of the left subtree
i. current=current->lptr
ii. while(current !=head)
a. display(current ->data)
. b. If current->rthread= 1
 current =current->rptr
c. else
i. current= current->rptr

ii. repeat while (current->lthread =0)
a. current =current->lptr
iii. end while
d. end if
iii. end while
Step 4: end While
Step 5: end of inorder

Analysis of Algorithm:

1. Time Complexity:

Time complexity for normal binary tree traversal whether recursive or non-recursive is also O(n), where 'n' is equal to number of nodes in binary tree, since every node is visited only once in both. For the TBT, it is still O(n), but it is constant times less than recursive and non-recursive using stack.

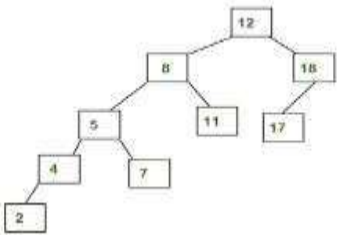
2. Space Complexity:
In recursive and non-recursive traversals of normal binary tree. We need a stack of depth 'k' which is equal to height or depth of a binary tree. But in TBT stack is not required.

Test cases / validation:

Test Cases
Test your program for following cases
For each test cases
1. tree with two nodes
2. tree with 10 nodes

INPUT :

Enter node in the following order



OUTPUT :
Show non recursive preorder and inorder traversal
Preorder : 12,8,5,4,2,7,11,18,17
Inorder : 2,4,5,7,8,11,12,17,18

Assignment 7- Minimum Spanning tree

AIM : Implementation of Minimum Spanning tree using Prims and Kruskals Algorithm

DETAILED PROBLEM STATEMENT:

Represent a graph of your college campus using adjacency list /adjacency matrix. Nodes should represent the various departments/institutes and links should represent the distance between them. Find minimum spanning tree

a) Using Kruskal's algorithm.
b) Using Prim's algorithm.

OBJECTIVE

1. To study Graph theory.
2. To study different graph traversal methods.
3. To understand the real time applications of graph theory.
4. To Implement a MST using Prims and Kruskals algorithm

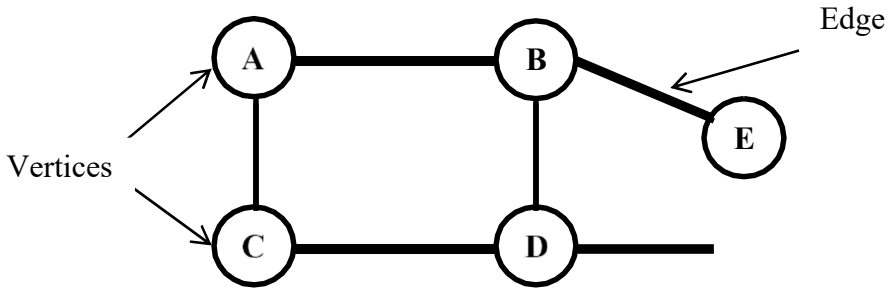
OUTCOME :

1. Understand Non-linear data structure - graph.
2. Represent a graph using adjacency list / adjacency matrix.
3. Implement Prim's and Kruskal's Algorithm
4. Identify applications of Minimum Spanning Trees.
5. Analyze the Time and Space complexity.

THEORY:

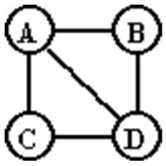
Introduction Graph :

- **Definition :** A graph is a triple G = (V,E, φ)where • V is a finite set, called the vertices of G, E is a finite set, called the edges of G, and,φ is a function with domain E and codomain P2(V).
- **Loops:** A loop is an edge that connects a vertex to itself.
- **Degrees of vertices**: Let G = (V,E, φ) be a graph and v ∈ V a vertex. Define the degree of v, d(v) to be the number of e ∈ E such that v ∈ φ(e); i.e., e is Incident on v.
- **Directed graph:** A directed graph (or digraph) is a triple D = (V,E, φ) where V and E are finite sets and φ is a function with domain E and codomain V × V . We call E the set of edges of the digraph D and call V the set of vertices of D.
- **Path:** Let G = (V,E, φ) be a graph.
  Let e1, e2, . . . , en−1 be a sequence of elements of E (edges of G) for which there is a sequence a1, a2, . . . , an of distinct elements of V (vertices of G) such that φ(ei) = {ai, ai+1} for i = 1, 2, . . . , n − 1. The sequence of edges e1, e2, . . . , en−1 is called a path in G. The sequence of vertices a1, a2, . . . , an is called the vertex sequence of the path.
- **Circuit and Cycle:** Let G = (V, E, φ) be a graph and let e1, . . . , en be a trail with vertex sequence a1, . . . , an, a1. (It returns to its starting point.) The subgraph G′of Ginduced by the set of edges{e1,...,en}is called a circuit of
  G. The length of the circuit is n.
  e.g.:
  This graph G can be defined as G = (V, E) Where V= {A,B,C,D,E} and
  E={(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.



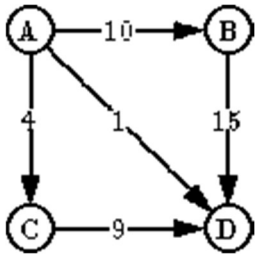
Different representations of graph:

1. **Adjacency matrix:** Graphs $G = (V, E)$ can be represented by adjacency matrices $G[v_1..v_{|V|}, v_1..v_{|V|}]$, where the rows and columns are indexed by the nodes, and the entries $G[v_i, v_j]$ represent the edges. In the case of unlabeled graphs, the entries are just Boolean values.



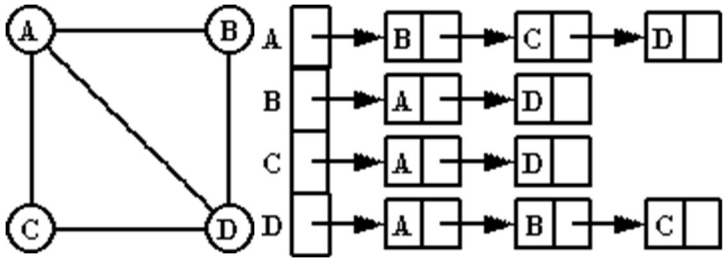
	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	1
D	1	1	1	0

In case of labeled graphs, the labels themselves may be introduced into the entries.



	A	B	C	D
A		10	4	1
B	∞	∞	∞	15
C	∞	∞	∞	9
D	∞	∞	∞	∞

2. **Adjacency List:** A representation of the graph consisting of a list of nodes, with each node containing a list of its neighboring nodes.



- **Spanning Tree:**

A Spanning Tree of a graph G = (V, E) is a sub graph of G having all vertices of G and no cycles in it.

Minimal Spanning Tree: The cost of a graph is the sum of the costs of the edges in the weighted graph. A spanning tree of a graph G= (V, E) is called minimal cost spanning tree or simply the minimal spanning tree of G if its cost is minimum.

- When a graph G is connected, depth first or breadth first search starting at any vertex visits all the vertices in G.
- The edges of G are partitioned into two sets i.e. T for the tree edges & B for back edges. T is the set of tree edges and B for back edges. T is the set of edges used or traversed during the search & B is the setof remaining edges.
- The edges of G in T form a tree which includes all the vertices of graph G and this tree is called as spanning tree.

Definition: Any tree, which consists solely of edges in graph G and includes all the vertices in G, is called as spanning tree. Thus for a given connected graph there are multiple spanning trees possible. For maximal connected graph having _n' vertices the number of different possible spanning trees is equal to n.
Cycle: If any edge from set B of graph G is introduced into the corresponding spanning tree T of graph G then cycle is formed. This cycle consists of edge (v, w) from the set B and all edges on the path from w to v in T.

There are many approaches to computing a minimum spanning tree. We could try to detect cycles and remove edges, but the two algorithms we will study build them from the bottom-up in a greedy fashion.

Prim's Algorithm – starts with a single vertex and then adds the minimum edge to extend the spanning tree.

Kruskal's Algorithm – starts with a forest of single node trees and then adds the edge with the minimum weight to connect two components.

- **Prim's algorithm:** Prim's algorithm is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm was discovered in 1930 by mathematician Vojtech Jarník and later independently by computer scientist Robert C. Prim in 1957 and rediscovered by Edsger Dijkstra in 1959. Therefore it is sometimes called the DJP algorithm, the Jarník algorithm, or the Prim-Jarník algorithm.

Applications of spanning Trees:

- To find independent set of circuit equations for an electrical network. By adding an edge from set B to spanning tree we get a cycle and then Kirch off's second law is used on the resulting cycle to obtain a circuit equation.
- Using the property of spanning trees we can select the spanning tree with (n-1) edges such that total cost is minimum if each edge in a graph represents cost.
- Analysis of project planning
- Identification of chemical compounds

- Statistical mechanics, genetics, cybernetics, linguistics, social sciences

ALGORITHMS/ PESUDOCODE :

- **Prims algorithm :**

Data structure used:

Array: Two dimensional array (adjacency matrix) to store the adjacent vertices & the weights associated edges

One dimensional array to store an indicator for each vertex whether visited or not. #define max 20

int adj_ver[max][max], int edge_wt[max][max], int ind[max]

- **Algorithm to generate spanning tree by Prim's**

Prims(Weight ,Vertex)

//Weight is a two dimensional array having V no of rows and columns. KOWN ,cost , PRIEV are the l vectors.

Step 1: Repeat for I = 1 to V
  i. KNOWN[I] = 0
  ii. PREV[I] = 0
  iii. cost[I] = 32767
Step 2: current= 1 //starting vertex of prims
Step 3: Total_V =0 //total vertex considered till the time
Step 4: KNOWN[current] =1 // Start is known now

Step 5: Repeat thru step 6 while Total_v != Vertex
  i. mincost= 32767
  ii. Repeat for I =1 to V
  iii. If (weight[current][I] != 0 AND KNOWN[I] =0)
    i. If (cost[I] >= weight[current][I])
      a. cost[I] = weight[current][I]
    ii. end if
  iv. Repeat for I =1 to V //finding min cost edge from current vertices
    i. If (KNOWN[I] = 0 AND cost[I] <= mincost)
      a. mincost =cost[I] //if min is Cost[i]
      b. current = I //next node visited is I
    ii. end if
  v. end if
Step 6: KNOWN[current] = 1
Step 7: Toatal_v= Total_v + 1
Step 8: mincost = 0
Step 9: Repeat for I = 1 to V
  i. WRITE(I , PREV[I]) //display mst edges
  ii. If cost[I] != 32767
    a. mincost =mincost + cost[I]
Step 10 : display final mincost
Step 11: end of prims

Trace of Prim's algorithm for graph G1 starting from vertex 1

Step No.	Set A	Set (V-A)	Min cost Edge (u, v)	Cost	Set B
Initial	{1}	{2,3,4,5,6,7}	--	--	{}
1	{1,2}	{3,4,5,6,7}	(1, 2)	1	{(1, 2)}
2	{1, 2, 3}	{4, 5, 6, 7}	(2, 3)	2	{(1,2},(2,3)}
3	{1,2,3,5}	{4, 6, 7}	(2, 5)	4	{(1,2),(2,3),(2,5)}
4	{1,2,3,5,4}	{6, 7}	(1,4)	4	{(1,2),(2,3),(2,5),(1,4)}
5	{1,2,3,5,4,7}	{6}	(4,7)	4	{(1,2),(2,3),(2,5),(1,4),(4,7)}
6	{1,2,3,5,4,7.6}	{ }	(7,6)	3	{(1,2),(2,3),(2,5),(1,4),(4,7),(7,6)}
		Total Cost		17	

Thus the minimum spanning tree for graph G1 is : A = { 1,2,3,4,5,7,6} B ={(1,2),(2,3),(2,5),(1,4),(4,7),(7,6)}, total Weight: 1+2+4+3+4+3=17

- **Kruskal's Algorithm:**

- **Prerequisite for Kruskals**

```
struct edge
{
    Number v1,v2,wt
} edge
```

- **Create edge Matrix**

AdjToEdges(int weight[][MAX], int n, edge E[])

```
Step 1: for i=0 to n do
Step 2:  for j=i+1 to n do
            i.i f(weight[i][j])
                a. edge_matrix[k]. start=i
                b. edge_matrix[k].end =j
                c. edge_matrix[k++].Value=G[i][j]
            ii. end if
Step 3:end for
Step 4: end for return k;
Step 5: end AdjToEdges
```

- **Function to sort the edges according to weights Algorithm**

SortEdges(edge edge_matrix[], no_edge)

```
Step 1:for i=0 to no_edge
Step 2: for j=i+1 to no_edge
            i. if(edge_matrix[i].value> edge_matrix[j].value)
                    a. t=edge_matrix[i]
                    b. edge_matrix[i]= edge_matrix[j]
                    c.  edge_matrix[j]=t
            ii. end if
Step 3 : end for j
Step 4: end for i
```

Step 5: end sort edge

Kruskals (G, N)

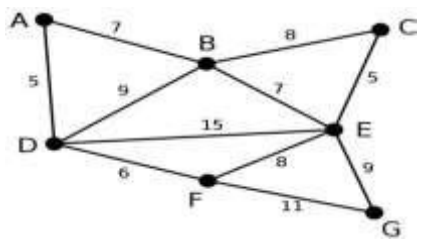
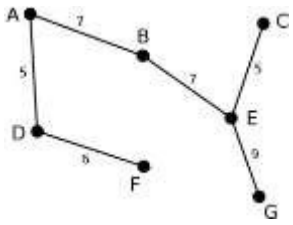
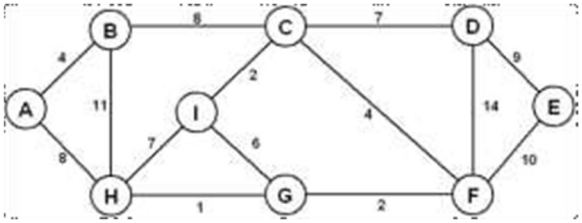
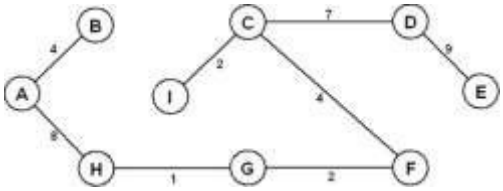
//G is a pointer to head of the adjacency List. N is max. number of vertices.
// L and K =0
Step 1: mincost = 0

Step 2: EARRAY(G)

Step 3:Repeat for I=1 to N
- i. set[I] = I

Step 4: Repeat Thru step 6 while L < Vertex-1 //select the min cost edge tillno of edges =Vertex-1
- i. T = Edge_mat[K] // select min cost edge
- ii. K = K+1 // to select next edge
- iii. Repeat for I = 1 to N
- iv. PV1=FIND(set,T.V1) //check the set Membership of V1
- v.PV2=FIND(set,T.V2) // check the Set membership of V2
- vi. if (PV1 != PV2) //if both v1 and v2 are belongs to different set
  //they are not forming cycle and can be added to fina
  - a.WRITE(V1(T),V2(T) , D(T))
  - b. mincost = micost + Dist[T]
  - c. L=L+1
  - d. for J=1 to N
    1. If (C[J] = PV2)
       i. C[J]=PV1
    2. end if
  - e. end for
- vii. end if

Step 5: display mincost
Step 6: end Kruskals

INPUT	OUTPUT	Remark
		Cost of MST -39
		Cost of MST -37

Testcases :

a) Display the total number of comparisons required to construct the graph in computer memory.

b) Display the results as given in the sample o/p above.

c) Finally conclude on time & time space complexity for the construction of the graph and for generation of minimum spanning tree using Prim‘s algorithm.

FAQS:

1. Explain the PRIM‘s algorithm for minimum spanning tree.
2. What are the traversal techniques?
3. What are the graph representation techniques?

Assignment 8 – Shortest Path

AIM: Implementation of Dijikstra's Algorithm
DETAILED PROBLEM STATEMENT :
Represent a given graph using adjacency matrix /adjacency list and find the shortest path using Dijkstra's algorithm (single source all destination).

OBJECTIVE:

1. To understand the application of Dijkstra's algorithm
2. To know various applications of Dijikstra's Algorithm

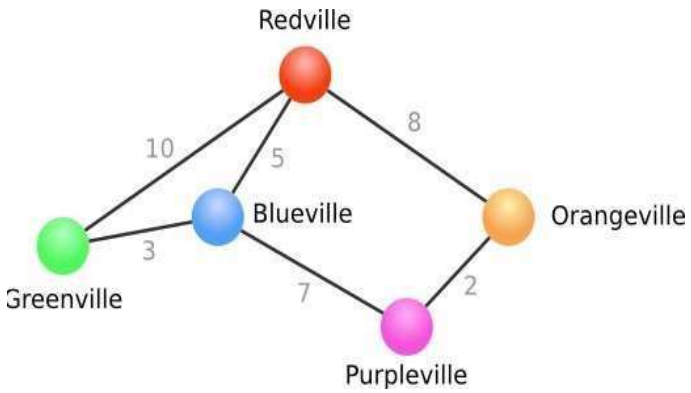
OUTCOME :

1. Implement Dijkstra's Algorithm.
2. Identify applications where Dijkstra's algorithm can be used

THEORY:
Definition of Dijkstra's Shortest Path

a) To find the shortest path between points, the weight or length of a path is calculated as the sum of the weights of the edges in the path.
b) A path is a shortest if there is no path from x to y with lower weight.
c) Dijkstra's algorithm finds the shortest path from x to y in order of increasing distance from x. That is, it chooses the first minimum edge, stores this value and adds the next minimum value from the next edge it selects.
d) It starts out at one vertex and branches out by selecting certain edges that lead to new vertices.
e) It is similar to the minimum spanning tree algorithm, in that it is "greedy", always choosing the closest edge in hopes of an optimal solution.

Example:
It is easiest to think of the geographical distances, with the vertices being places, such as cities.



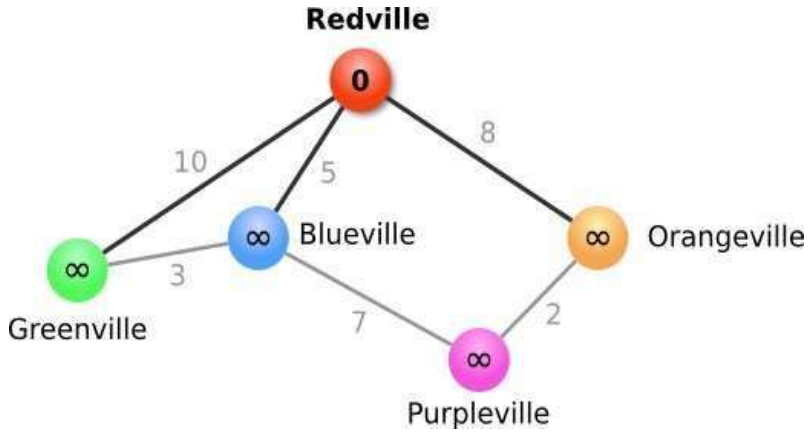
Imagine you live in Redville, and would like to know the shortest way to get to the surrounding towns: Greenville, Blueville, Orangeville, and Purpleville. You would be confronted with problems like: Is it faster to go through Orangeville or Blueville to get to Purpleville? Is it faster to take a direct route to Greenville, or to take the route that goes through Blueville? As long as you knew the distances of roads going directly from one city

to another, Dijkstra's algorithm would be able to tell you what the best route for each of the nearby towns would be.

- Begin with the source node (city), and call this the current node. Set its value to 0. Set the value of all other nodes to infinity. Mark all nodes as unvisited.
- For each unvisited node that is adjacent to the current node (i.e. a city there is a direct route to from the present city), do the following. If the value of the current node plus the value of the edge is less than the value of the adjacent node, change the value of the adjacent node to this value. Otherwise leave the value as is.
- Set the current node to visited. If there are still some unvisited nodes, set the unvisited node with the smallest value as the new current node, and go to step 2. If there are no unvisited nodes, then we are done.

In other words, we start by figuring out the distance from our hometown to all of the towns we have a direct route to. Then we go through each town, and see if there is a quicker route through it to any of the towns it has a direct route to. If so, we remember this as our current best route.

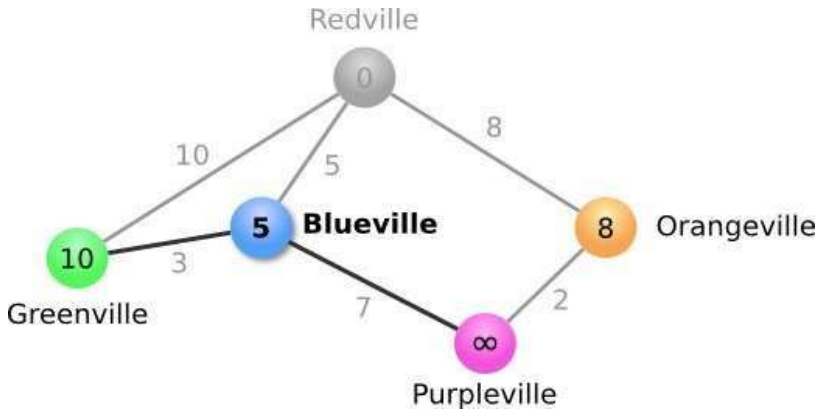
StepI:



We set Redville as our current node. We give it a value of 0, since it doesn't cost anything to get to it from our starting point. We assign everything else a value of infinity, since we don't yet know of a way to get to them.

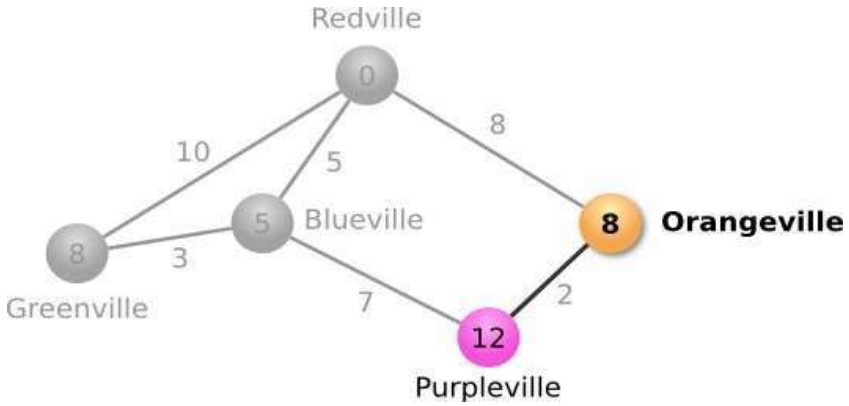
Step II:

Next, we look at the unvisited cities our current node is adjacent to. This means Greenville, Blueville and Orangeville. We check whether the value of the connecting edge, plus the value of our current node, is less than the value of the adjacent node, and if so we change the value. In this case, for all three of the adjacent nodes we should be changing the value, since all of the adjacent nodes have the value infinity. We change the value to the value of the current node (zero) plus the value of the connecting edge (10 for Greenville, 5 for Blueville, 8 for Orangeville). We now mark Redville as visited, and set Blueville as our current node since it has the lowest value of all unvisited nodes.



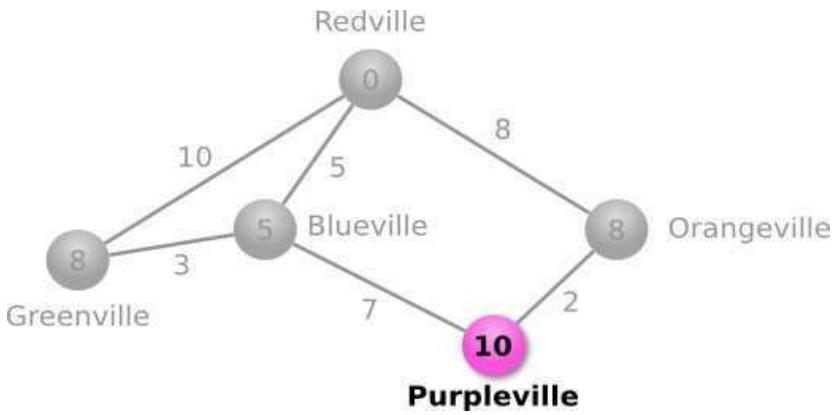
Step III:

The unvisited nodes adjacent to Blueville, our current node, are Purpleville and Greenville. So we want to see if the value of either of those cities is less than the value of Blueville plus the value of the connecting edge. The value of Blueville plus the value of the road to Greenvile is 5 + 3 = 8. This is less than the current value of Greenville (10), so it is shorter to go through Blueville to get to Greenville. We change the value of Greenville to 8, showing we can get there with a cost of 8. For Purpleville, 5 + 7 = 12, which is less than Purpleville‘s current value of infinity, so we change its value as well? We mark Blueville as visited. There are now two unvisited nodes with the lowest value (both Orangeville and Greenville have value 8). We can arbitrarily choose Greenville to be our next current node. However, there are no unvisited nodes adjacent to Greenville! We can mark it as visited without making any other changes, and make Orangeville our next current node.



Step IV:

There is only one unvisited node adjacent to Orangeville. If we check the values, Orangeville plus the connecting road is 8 + 2 = 10, Purpleville's value is 12, and so we change Purpleville's value to 10. We mark Orangeville as visited, and Purpleville is our last unvisited node, so we make it our current node. There are no unvisited nodes adjacent to Purpleville, so we're done!



All above steps can be simply put in a tabular form like this:

Current	Visited	Red	Green	Blue	Orange	Purple	Description
Red		0	Infinity	Infinity	Infinity	Infinity	Initialize Red as current, set initial values
Red		0	10	5	8	Infinity	Change values for Green, Blue, Orange
Blue	Red	0	10	5	8	Infinity	Set Red as visited, Blue as current
Blue	Red	0	8	5	8	12	Change value for Purple
Green	Red, Blue	0	8	5	8	12	Set Blue as visited, Green as current
Orange	Red, Blue, Green	0	8	5	8	12	Set Green as visited, Orange as current
Orange	Red, Blue, Green	0	8	5	8	10	Change value for Purple
Purple	Red, Blue, Green, Orange	0	8	5	8	10	Set Orange as visited, Purple as current
	Red, Blue, Green, Orange, Purple	0	8	5	8	10	Set Purple as visited

ALGORITHM/PESUDOCODE:

College Area represented by Graph.A graph G with N nodes is maintained by its adjacency matrix Cost.
Dijkstra's algorithm find shortest path matrix D of GraphG.
Staring Node is 1.

[I] Dijikstras(v,cost,dist,n)

// Using distance and visited
//dist[j],1<=j<=n, is set to the length of the shortest path from vertex v to vertex j in a diagraph
//G with n vertices. dist[v] is then set to zero.

//G is represented by its cost adjacency matrix cost[1:n,1:n]

Step 1: For i=1 to n do
// Initialize S
i. S[i]=false
ii. dist[i]=cost[v,i]

Step 2: S[v]=true
Step 3 : Dist[v]=0 //put v in S
Step 4 : for num=2 to n-1 do
//determine n-1 paths from v
// Choose u from among those vertices not in S such that dist[u] is minimum
i. S[u]=true //put u in S
ii. for (each w adjacent to u with S[w]=false)do
//update distance
→ if (dist[w]>dist[u]+cost[u,w])) then
→ dist[w]=dist[u]+cost[u,w]
➔ End if
iii. end for
Step 5: end of Dijikistras

II. Dijkstra(Graph , source) //using Queu data structure

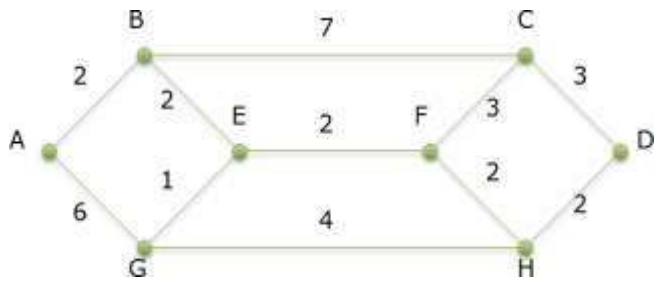
Step 1: for each vertex v in Graph : // Initializations
i. dist[v] := infinity // Unknown distance function from source to v
ii. previous[v] := undefined // Previous node in optimal path from source
end for
Step 2: dist[source] := 0 // Distance from source to source
Step 3 : Q := the set of all nodes in Graph // All nodes in the graph are unoptimized - thus are in Q
Step 4 : while Q is not empty: // The main loop
i. u := vertex in Q with smallest dist[]
ii. if dist[u] = infinity
a. break // all remaining vertices are inaccessible from source
iii. remove u from Q
iv. for each neighbor v of u : // where v has not yet been removed from Q
a. alt := dist[u] + dist_between(u , v)
b. if alt < dist[v]: // Relax (u,v,a)
i. dist[v] := alt
ii. previous[v] := u

Step 5: for 1 to n do
Display (previous[i], I , dist[i]) // display shortest path

INPUT:
The graph in the form of edges, nodes and corresponding weights, the source node and destination node.

OUTPUT:
The shortest path and length of the shortest path

SAMPLE INPUT



SAMPLE OUTPUT

Shortest Path is
A -B-E-F-H-D

Cost - 10

Remark

Consider Source Vertex as node A and Destination Vertex as node D

FAQs:

1. What at is shortest path?
2. What are the graph representation techniques?
3. What is adjacency Matrix?
4. What is adjacency list?
5. What is adjacency Multi-list?

Assignment 9 - Heap sort

AIM: **Implementation of Heap sort**
DETAILED PROBLEM STATEMENT :
Implement Heap sort to sort given set of values using max or min heap.

OBJECTIVE:

1. To know the concept of a Heap data structure
2. To learn heap sort
3. To analyze heap sort time and space complexity

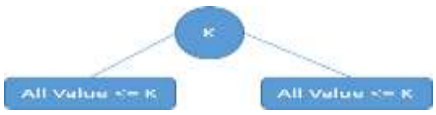
OUTCOME:

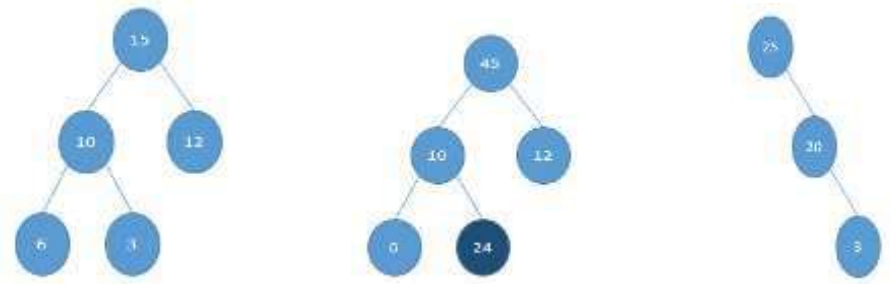
1. Understand the properties of Heap data structure.
2. Implement Heap sort.
3. Analysis of heap sort

THEORY:

Heap Data Structure :

i) Heap Tree:
Heap is a binary tree structure with the following properties:
1. The tree is complete or almost complete
2. The key value of each node is greater than or equal to the key value of its descendants.





(A) Valid Heap Tree

(B) Invalid Heap Tree due to sub-tree 10

(C) Invalid Heap Tree not nearly complete

ii) Max-Heap
If the key value of each node is greater than or equal to the key value of its descendants, this heap structure is called Max-Heap.

iii) Min-Heap
If the key value of each node is less than or equal to the key value of its descendants, this heap structure is called Min-Heap.

b)Maintenance Operations on Heap
To perform insert and delete a node in heap structure, it needs two basic algorithms :
1.**Reheap up**
This operation reorders a "broken" heap by floating the last element up the tree until it is at its correct position in the heap.

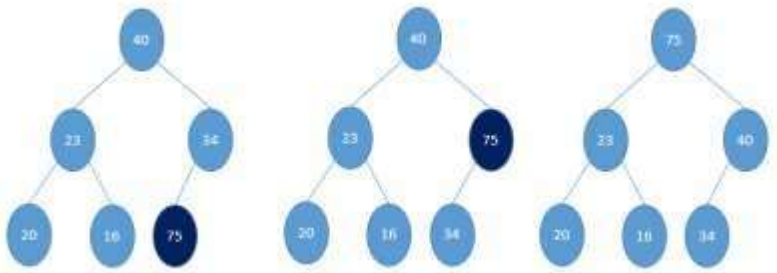


Fig. a
Not a heap tree

Fig. b
node 75 moved up
at correct position,

Fig. c
node 75 moved
it is heap tree

2.Reheap down
This operation reorders a "broken" heap by pushing the root down the tree until it is at its correct position in the heap.

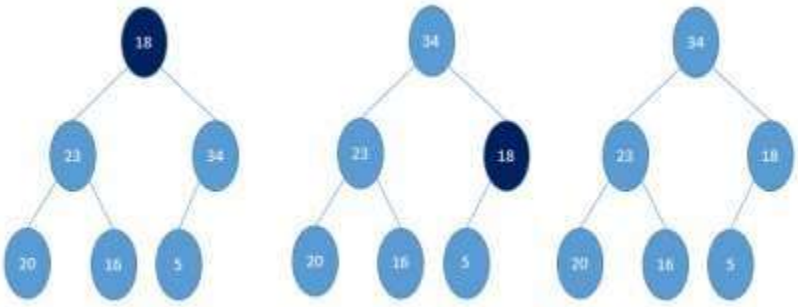


Fig. a
Not a heap tree

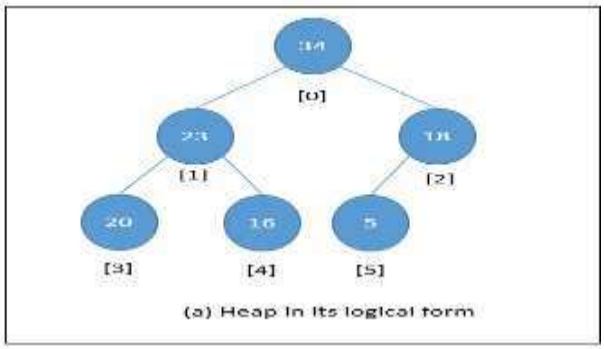
Fig. b
Moved down node
18 at its correct place

Fig. c
It is heap tree
(Max heap)

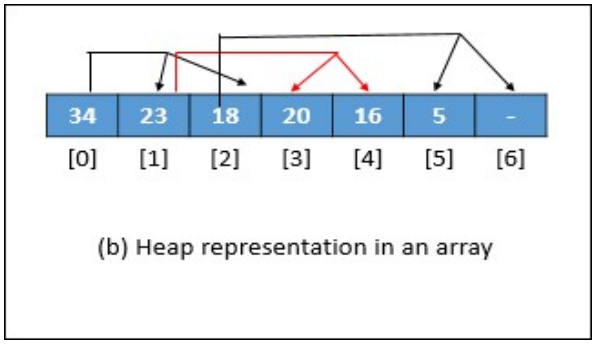
Heap Implementation

Heap implementation is possible using an array because it must be a complete or almost complete binary tree, which allows a fixed relationship between each node and its children and it can be calculated as :

i. For node located at ith index, its children are :
   Left child : 2i + 1
   Right child : 2i + 2
ii. Parent of node located at ith index : (i-1)/2
iii. Given the index for a left child, j,its right sibling at : j+1
iv. Given the index for a right child, k, its left sibling at :k-1



(a) Heap in its logical form



(b) Heap representation in an array

Applications of Heap

i. Selection algorithm
ii. Priority Queue
iii. Sorting (Heap Sort)

Heap Sort Complexity

The Heap sort efficiency is O(n log n)

ALGORITHMS/PESUDOCODE:

Heapify_asc(data,i)

Step 1 : Set Lt = Left(i)
Step 2 : Set Rt = Right(i)
Step 3 : if Lt <= heap_size[data] - 1 and data[Lt] > data[i]
i. then Set Max = Lt;
else
ii. Set Max = i
Step 4 : if Rt <= heap_size[data] - 1 and data[Rt] > data[Max]
i. then Set Max = Rt
Step 5 : if Max != i
i. then Swap(data[i], data[Max])
Step 6 : Heapify_asc(data, Max)
Step 7: end of Heapify_asc

BuildHeap(data)

Step 1 : Set heap_size[data] = length[data]
Step 2 : for i= (length[data]-1)/2 to 0
i. Heapify(data,i)
Step 3: End of BuildHEap

Heapsort(data)

Step 1 : BuildHeap(data)
Step 2 : for i = length[data] - 1 to 1
i. Swap(data[0],data[i])
ii. Set heap_size[data] = heap_size[data] - 1
iii. Heapify(data,0)
Step 3: End of heapsort

INPUT:

1. 45, 78, 24, 36, 12
2. 10, 28, 56, 68, 75
3. 89, 70, 64, 52,

OUTPUT:

Enter how many elements you want to sort? :45 5
Enter elements :45 78 36 24 12
1. Ascending order
2. Descending order
3. Exit
Enter your choice :1

size = 5 78 45 36 24 12
After building the heap…
Sorted Elements are: 12 24 36 45 78
1. Ascending order
2. Descending order
3. Exit
Enter your choice :2

size = 5 12 24 36 45 78
After building the heap...
Sorted Elements are: 78 45 36 24 12
1. Ascending order
2. Descending order
3. Exit
Enter your choice :3

FAQS:

1. How do you sort an array using heap sort?
2. What are the heap properties ?
3. What is the space complexity of heap Sort?
4. Heap sort time complexity ?
5. Compare heap sort with quick and merge sort?

Assignment 10 - Sequential file Organization

AIM: To implement program for Sequential Access File

DETAILED PROBLEM STATEMENT:

Department maintains student's database. The file contains roll number, name, division and address. Write a program to

i. create a sequential file to store and maintain student data.
ii. It should allow the user to add, delete information of student.
iii. Display information of particular student.
-> If record of student does not exist an appropriate message is displayed.
->If student record is found it should display the student details.

OBJECTIVE:

1. Understand the concept of Permeant Data structure
2. To learn Sequential file organization.
3. Create a sequential file and various operations on file .
4. Design the application using file data structure for database management .

OUTCOME:

1. Understand file as a permanent data structure.
2. Implement a file and preform various operations on it.
3. To implement the Database application using file data structure

THEORY :

File :

A file is a collection on information, usually stored on a computer‘s disk. Information can be saved to files and then later reused.

Type of File

- **Binary File**

- The binary file consists of binary data
- It can store text, graphics, sound data in binary format
- The binary files cannot be read directly o Numbers stored efficiently

- **Text File**

- The text file contains the plain ASCII characters
- It contains text data which is marked by _end of line‘ at the end of each record
- This end of record marks help easily to perform operations such as read and write o Text file cannot store graphical data.

- **File Organization :**

The proper arrangement of records within a file is called as file organization. The factors that affect file organization are mainly the following:

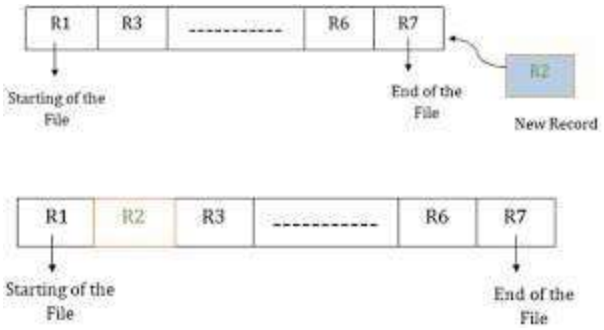
- Storage device
- Type of query
- Number of keys
- Mode of retrieval/update of record

- **Different types of File Organizations are as :**

- Sequential file
- Direct or random access file
- Indexed sequential file
- Multi-Indexed file

- **Sequential file :**

In sequential file, records are stored in the sequential order of their entry.
This is the simplest kind of data organization. The order of the records is fixed. Within each block, the records are in sequence . A sequential file stores records in the order they are entered. New records always appear at the end of the file.



Features of Sequential files :

- Records stored in pre-defined order.
- Sequential access to successive records.
- Suited to magnetic tape.
- To maintain the sequential order updating becomes a more complicated and difficult task**.** Records will usually need to be moved by one place in order to add (slot in) a record in the proper sequential order. Deleting records will usually require that records be shifted back one place to avoid gaps in the sequence.
- Very useful for transaction processing where the hit rate is very high e.g. payroll systems, when the whole file is processed as this is quick and efficient.
- Access times are still too slow (no better on average than serial) to be useful in online applications.

Drawbacks of Sequential File Organization

- Insertion and deletion of records in in-between positions huge data
- Movement Accessing any record requires a pass through all the preceding records,

which is time consuming. Therefore, searching a record also takes more time.
- Needs reorganization of file from time to time.
- If too many records are deleted logically, then the file must be reorganized to free the space occupied by unwanted records

Primitive Operations on Sequential files

- **Open—**This opens the file and sets the file pointer to immediately before the first record
- **Read-next—**This returns the next record to the user. If no record is present, then EOF condition will be set.
- **Close—**This closes the file and terminates the access to the file
- **Write-next—**File pointers are set to next of last record and write the record to the file
- **EOF—**If EOF condition occurs, it returns true, otherwise it returns false
- **Search—**Search for the record with a given key
- **Update—**Current record is written at the same position with updated values

➢ **Direct or random access file :**

Files that have been designed to make direct record retrieval as easy and efficiently as possible is known as directly organized files. Though we search records using key, we still need to know the address of the record to retrieve it directly. The file organization that supports Files such access is called as direct or random file organization. Direct access files are of great use for immediate access to large amounts of information. They are often used in accessing large databases.

Advantages of Direct Access Files :

- Rapid access to records in a direct fashion.
- It doesn't make use of large index tables and dictionaries and therefore response times are very fast.

➢ **Indexed sequential file :**

- Records are stored sequentially but the index file is prepared for accessing the record directly. An index file contains records ordered by a record key. The record key uniquely identifies the record and determines the sequence in which it is accessed with respect to other records.

- A file that is loaded in key sequence but can be accessed directly by use of one or more indices is known as an indexed sequential file. A sequential data file that is indexed is called as indexed sequential file. A solution to improve speed of retrieving target is index sequential file. An indexed file contains records ordered by a record key. Each record contains a field that contains the record key.

- This system organizes the file into sequential order, usually based on a key field, similar in principle to the sequential access file. However, it is also possible to directly access records by using a separate index file. An indexed file system consists of a pair of files: one holding the data and one storing an index to that data.The index file will store the addresses of the records stored on the main file. There may be more than one index created for a data file e.g. a library may have itsbooks stored on computer with indices on author, subject and class mark.

Characteristics of Indexed Sequential File :

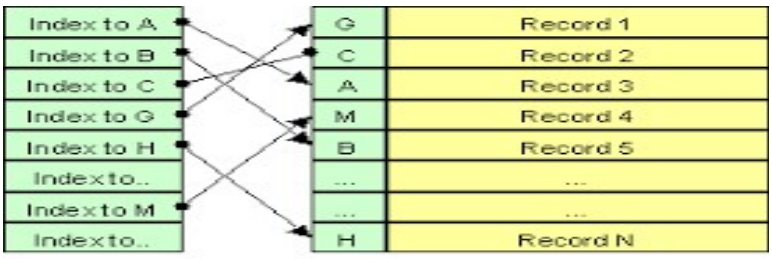
- Records are stored sequentially but the index file is prepared for accessing the
- record directly
- Records can be accessed randomly
- File has records and also the index
- Magnetic tape is not suitable for index sequential storage
- Index is the address of physical storage of a record
- When randomly very few are required/accessed, then index sequential is better
- Faster access method
- Addition overhead is to maintain index
- Index sequential files are popularly used in many applications like digital library

Primitive operations on Index Sequential files (IS):

- **Write (add, store) :** User provides a new key and record, IS file inserts the new
- record and key.
- **Sequential Access (read next) :** IS file returns the next record (in key order)
- **Random access (random read, fetch) :** User provides key, IS file returns the record or "not there"
- **Rewrite (replace) :** User provides an existing key and a new record, IS file replaces existing record with new.
- **Delete :** User provides an existing key, IS file deletes existing record

Indexed Sequential Files :

This is basically a mixture of sequential and indexed file organization techniques. Records are held in sequential order and can be accessed randomly through an index. Thus, these files share the merits of both systems enabling sequential or direct access to the data. The index to these files operates by storing the highest record key in given cylinders and tracks. Note how this organization gives the index a tree structure. Obviously this type of file organization will require a direct access device, such as a hard disk. Indexed sequential file organization is very useful where records are often retrieved randomly and are also processed in (sequential) key order. Banks may use this organization for their auto-bank machines i.e. customers randomly access their accounts throughout the day and at the end of the day the banks can update the whole file sequentially.



Indexed File Organisation

Advantages of Indexed Sequential Files:

1. Allows records to be accessed directly or sequentially.
2. Direct access ability provides vastly superior (average) access times.

Disadvantages of Indexed Sequential Files:

1. The fact that several tables must be stored for the index makes for a considerable storage overhead.
2. As the items are stored in a sequential fashion this adds complexity to the addition/deletion of records. Because frequent updating can be very inefficient, especially for large files, batch updates are often performed.

Application of files :
Database applications :

- ticket reservation system
- hotel management system
- online examinations
- student admission process etc.

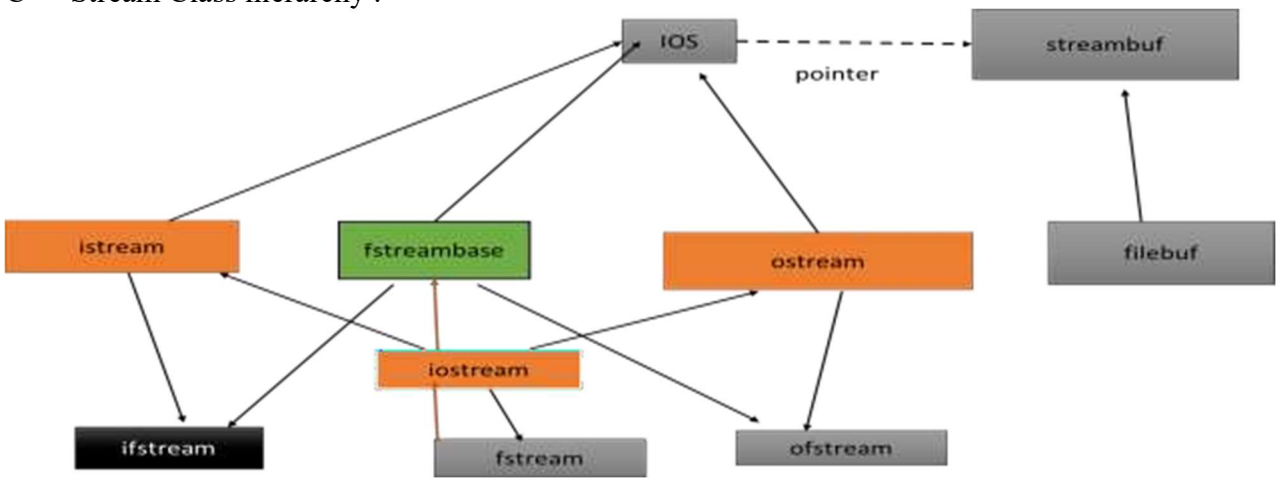
Software System applications

- operating system,
- language processors
- graphics systems etc.

To Build a vital software system of the future,

- We need to be able to structure and manipulate information effectively and efficiently
- To fulfill this we need a information management system .
- File organization is one of its component**.**

C++ Stream Class hierarchy :



Class	Functions	Meaning
Ifstream	open() get() getline() read()	Open the file in default input mode(read from file) ios::in Read one character from the file Read one complete line from the file Read any type of data from the file

	tellg(), seekg()	Direct access file functions(not required for this assignment)
Ofstream	open() put() write() tellp(), seekp()	Open the file in default output mode(write into file)ios::out Write one character into the file Write any type of data into the file Direct access file functions(not required for this assignment)
Fstream	Inherits all above functions through iostream	Provides support for simultaneous input and output operations. Ios::in|ios::out

Outline of implementation

1.

```
struct student
{
Int stud_rollno;
char stud_name[30];
char stud_division;
char stud_address[30];
DoB stud_DoB;
float stud_percent;
char stud_grade;
}
```

2.

```
struct DoB
{
        int stud_day;
 int stud_month;
int stud_year;
        }
```

3.

```
class seqfile
{
        student stud_rec;
  ostream outfile;
   istream infile;

        public:
    void create();
    void display(key);
    void add();
void search(key);
        Void modify(key);
        void delete(key);
  }
```

ALGORITHMS :

Note : StudentData is file a name

- **Create a file**

CreateAFile()

Step 1 : Open StudentData for output
Step 2 : If(file opened) then
Step 3: scan noofReords
Step 4: For I =1 to noofRecords
    i. Display "Please enter the information of student: "
    ii. Get rollno, name, divison, address, date of birth, percentage, grade
    iii. Write rollno, name, divison, address, date of birth, percentage, grade into StudentData
Step 5: end for
Step 6 : Close file
Step 7: END CreateAFile

- **Display a file**

DisplayFileContents()

Step 1: Open StudentData for input
Step 2: If FileNotPresent
    i. Display error message
    ii. Exit
Step 3: end if
Step 4: while Not EndofFile StudentData
    i. Read Student Information from StudentData
    ii. Display Student Information
Step 5: end while
Step 6: Close StudentData
Step 7 : endDisplayFileContents

- **Add a record**

AddNewRecords()

Step 1: Open StudentData for append // file pointer will automatically moved to the end of file
Step2 : If FileNotPresent
    i. Display error message
    ii. Exit
Step 3: end if
Step 4: Display "Please enter the information of student: "
    i. Get rollno, name, divison, address, date of birth, percentage, grade
    ii. Write rollno, name, divison, address, date of birth, percentage, grade into StudentData

Step 5: Close StudentData
Step 6: end AddNewRecords

- **Search a record**

SearchRecord(key)

Step 1: Open StudentData for input
Step 2: if FileNotPresent
    i. Display error message
    ii. Exit
Step 3: end if
Step 4: while Not EndofFile StudentData
    i. Read Student Information from StudentData
    ii. if StudentRecord contains key //key can be unique roll no or if 'name' there can be multiple records displayed
        a. Display Student Information
    iii. ENDIF
Step 5 : end while
Step 6: Close StudentData
Step 7: end SearchRecord

- **Modify a Record**

ModifyRecord(key)
Step 1: Open StudentData for input
Step 2: if FileNotPresent
    i. Display error message
    ii. Exit
Step 3: end if
Step 4: Open Temporary file for output
Step 5: while Not EndofFile StudentData
    i. Read Student Information from StudentData
    ii. if StudentRecord contains key //key should be unique as 'roll no'
        a. Display Student Information
        b. Get new information for modification
        c. Write information into Temporary file
    iii. else
        a. Write information into Temporary file
    iv. end if
Step 6: end while
Step 7: Delete StudentData
Step 8: Rename Temporary File as StudentData
Step 9: Close StudentData
Step 10:end ModifyRecord

- **Delete a Record**

Delete Record(key)
Step 1: Open StudentData for input
Step 2: if FileNotPresent
    i. Display error message
    ii. Exit
Step 3: end if
Step 4: Open Temporary file for output
Step 5: while Not EndofFile StudentData
    i. Read Student Information from StudentData
    ii. if StudentRecord contains key //key should be unique as 'roll no'
        a. Continue // read next record
    iii. else
        a. Write information into Temporary file
    iv. endif
Step 6: end whil
Step 7: Delete StudentData
Step 8: Rename Temporary File as StudentData
Step 9: Close StudentData
Step 10: end ModifyRecord

Test Cases:

1. Open a file in reading mode which does not exist.
2. Open a file in writing mode which is already exist.
3. Search /Modify/Delete record with no key present.
4. Not able to open a new file.
5. We may have invalid file name.

FAQs:

1. Which header file is required to use file I/O operations?
2. Which class is used to create an output stream?
3. Which class is used to create a stream that performs both input and output operations?
4. By default, all the files in C++ are opened in_____ mode.
5. What is the use of ios::trunc mode?
6. What is the return type of open() method?
7. Which is the default mode of the opening using the fstream class?
8. Which is the default mode of the opening using the ifstream class