GH raisoni COLLEGE
Engineering and Management
Pune

G H Raisoni College of Engineering & Management
An Empowered Autonomous Institute, Affiliated to Savitribai Phule Pune University
Approved by AICTE, New Delhi and Recognized by Govt. of Maharashtra, NAAC Accredited "A+" Grade
Navin Gat No.1200, Domkhel Road, Wagholi, Pune – 412 207 (India)
T: +91 - 9604787185/186 | E: ghrcem.pune@raisoni.net | W: www.ghrcemp.raisoni.net

Department of Information Technology

LAB MANUAL

Data Structures and Algorithms Lab (23UITP2305)

SY B.Tech. (IT) Semester- III

Pattern 2023 (As Per NEP 2020)

Academic Year: 2024-25

List of Laboratory Experiments

Sr. No.	List of Experiments	Relevance to CO
1.	Write a program to perform various operations on Arrays Data Structures.	CO1
2.	Write a program to demonstrate Bubble Sort and Insertion sort techniques using arrays. Consider a student database of SY IT class (at least 10 records). Database contains different fields of every student like Roll No, Name and SGPA. Design a roll call list, arrange list of students according to roll numbers in ascending order. Arrange list of students alphabetically.	CO1
3	Write a program to demonstrate **singly and double linked list**: To maintain club member's information using singly linked list for following: Department of Information Technology has student's club named 'COMIT'. Students of Second, third and final year of department can be granted membership on request. Similarly one may cancel the membership of club. First node is reserved for president of club and last node is reserved for secretary of club. Store student MIS registration no. and Name. Write functions to a) Add and delete the members as well as president or even secretary. b) Compute total number of members of club c) Display members d) Display list in reverse order using recursion e) Two linked lists exists for two divisions. Concatenate two lists.	CO2
4	Write a program to search a particular student according to SGPA using binary search, and Hashing.	CO2
5.	Write a program to perform the **Stack operations**: And Write a program to convert an Infix expression to Postfix expression. **OR t**o check the well form-ness of parenthesis, whether a given expression is balanced or unbalanced**.**	CO2
6	Write a program to simulate the **Queues** for following: Create a queue for jobs of an operating system, write functions to add job and delete job from queue. If the operating system does not use priorities, then the jobs are processed in the order they enter the system. **OR**	CO2

	Write a program to simulate double-ended queue (deque) with functions to add and delete elements from either end of the deque and obtain a data representation mapping a deque into a one-dimensional array.	
7.	Write a program to demonstrate **tree** for following: Construct the Binary tree using linked list and perform the inorder, preorder, post order traversals (recursive and non-recursive). **OR** A book consists of chapters, the chapters consist of sections and sections consist of subsections. Construct a tree and print the nodes and find the time and space requirements of the method.	CO3
8	Write a program to construct **Binary search tree** by inserting the values in the order given. Begin with an empty binary search tree and after constructing the tree i. Insert new node ii. Find number of nodes in longest path iii. Minimum data value found in the tree iv. Change a tree so that roles of the left and right pointers are swapped at every node v. Search a value	CO3
9.	Write a program to perform the **Graph** Traversal techniques **DFS** and **BFS** : Perform the Depth First Search and Breadth First Search using adjacency list or matrix, use the map of the historical places around Pune city as the graph, Identify the famous land marks as nodes and perform DFS and BFS.	CO4
10.	Write a program to demonstrate the minimum spanning tree using **Prim's** and **Kruskal's** algorithm for following: There is a business with several offices that need to lease phone lines to connect them up with each other; and the phone company charges different amounts of money to connect different pairs of cities. You want a set of lines that connects all your offices with a minimum total cost.	CO4
Content Beyond Syllabus		
1.	Write a program to Insert and Search Operations in Trie (Prefix tree) Data Structure.	
2.	Write a program to Insert, Delete, and Search Operations in Skip List Data Structure.	

Experiment No. 1

Aim: - Write a program to demonstrate **various operations on arrays**.

Objective: - Perform various operations on arrays like traversal, insertion, deletion, search, update, sorting, and display.

Mapping with CO1: Describe basics of data structures and analyze complexity of various algorithms

Flowchart:- draw flowchart on left side blank (without line rolling page).

Theory:- Array is a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

Declaring Arrays:-

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows −

type arrayName [arraySize];

This is called a single-dimensional array. The arraySize must be an integer constant greater than zero and type can be any valid C data type. For example, to declare a 10-element array called balance of type double, use this statement −

float balance[10];

Here balance is a variable array which is sufficient to hold up to 10 double numbers.

Initializing Arrays:-

You can initialize an array in C either one by one or using a single statement as follows −

```
float  balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

Accessing Array Elements:-

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example −

```
double salary = balance[9];
```

Basic operations supported by an array:-

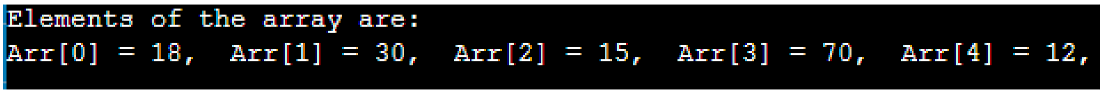
- Traverse − print all the array elements one by one.
- Insertion − Adds an element at the given index.
- Deletion − Deletes an element at the given index.
- Search − Searches an element using the given index or by the value.
- Update − Updates an element at the given index.
- Sorting − rearrange of elements in a specific order (Ascending/descending)
- Display − Displays the contents of the array.

Traversal operation

This operation is performed to traverse through the array elements. It prints all array elements one after another.

```
#include <stdio.h>
int main() {
  int Arr[5] = {18, 30, 15, 70, 12};
  int i;
  printf("Elements of the array are:\n");
  for(i = 0; i<5; i++)
    printf("Arr[%d] = %d,  ", i, Arr[i]);
return 0;
}
```

Output



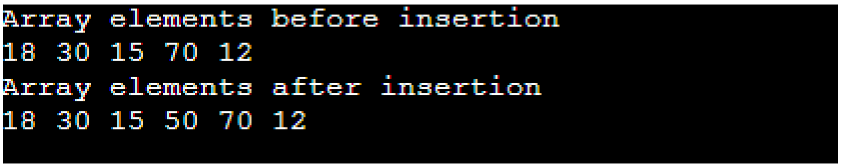
```
Elements of the array are:
Arr[0] = 18,  Arr[1] = 30,  Arr[2] = 15,  Arr[3] = 70,  Arr[4] = 12,
```

Insertion operation

This operation is performed to insert one or more elements into the array. As per the requirements, an element can be added at the beginning, end, or at any index of the array.

```
#include <stdio.h>
int main() {
int arr[20] = { 18, 30, 15, 70, 12 };
int i, x, pos, n = 5;
printf("Array elements before insertion\n");
for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
printf("\n");
x = 50; // element to be inserted
pos = 4;
n++;   //n =6  // 6-1=5   5>=4
for (i = n-1; i >= pos; i--)  //n-1 = 6-1 =5
        arr[i] = arr[i - 1];  // arr[5] =arr[5-1] =arr[4]
arr[pos - 1] = x;       // arr[4-1]=arr[3]=50
printf("Array elements after insertion\n");
for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
printf("\n");
return 0;
}
```

Output



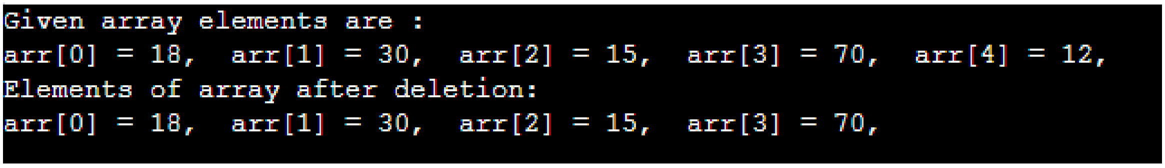
```
Array elements before insertion
18 30 15 70 12
Array elements after insertion
18 30 15 50 70 12
```

Deletion operation

As the name implies, this operation removes an element from the array and then reorganizes all of the array elements.

```
#include <stdio.h>
int main() {
   int arr[] = {18, 30, 15, 70, 12};
   int k = 30, n = 5;
   int i, j;
   printf("Given array elements are :\n");
   for(i = 0; i<n; i++) {
      printf("arr[%d] = %d,  ", i, arr[i]);
   }
   j = k;
   while( j < n) {
      arr[j-1] = arr[j];
      j = j + 1;
   }
   n = n -1;
   printf("\nElements of array after deletion:\n");
   for(i = 0; i<n; i++) {
      printf("arr[%d] = %d,  ", i, arr[i]);
   }
}
```

Output



```
Given array elements are :
arr[0] = 18,  arr[1] = 30,  arr[2] = 15,  arr[3] = 70,  arr[4] = 12,
Elements of array after deletion:
arr[0] = 18,  arr[1] = 30,  arr[2] = 15,  arr[3] = 70,
```

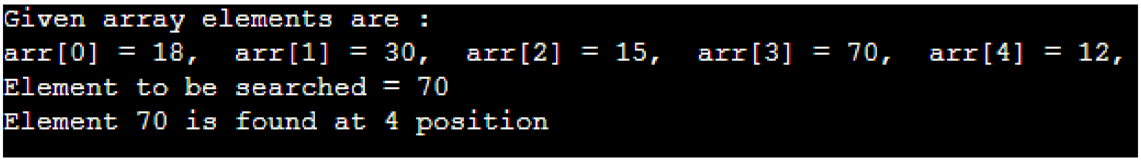
Search operation

Linear search (sequential search) is a method for searching for an element in a collection of elements. In linear search, each element of the collection is visited one by one in a sequential fashion to find the desired element.

This operation is performed to search an element in the array based on the value or index.

```
#include <stdio.h>
int  main() {
   int arr[5] = {18, 30, 15, 70, 12};
   int item = 70, i, j=0 ;
   printf("Given array elements are :\n");
   for(i = 0; i<5; i++) {
      printf("arr[%d] = %d,  ", i, arr[i]);
   }
    printf("\nElement to be searched = %d", item);
    while( j < 5){                 // j=0   j<5  0<5
      if( arr[j] == item ) {    // arr[0]==70  18==70  arr[1]=30  30==70  15==70   arr[3]=70  70==70
         break;
      }
      j = j + 1;
   }
   printf("\nElement %d is found at %d position", item, j+1);
}
```

Output



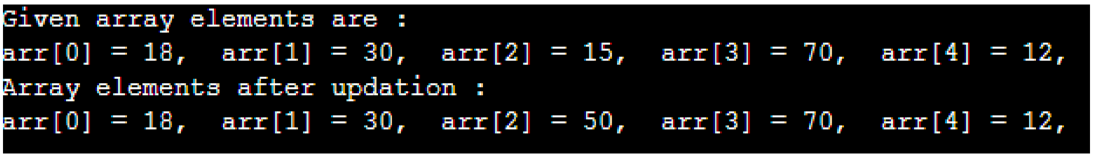
```
Given array elements are :
arr[0] = 18,  arr[1] = 30,  arr[2] = 15,  arr[3] = 70,  arr[4] = 12,
Element to be searched = 70
Element 70 is found at 4 position
```

Update operation

This operation is performed to update an existing array element located at the given index.

```
#include <stdio.h>
int main() {
  int arr[5] = {18, 30, 15, 70, 12};
  int item = 50, i, pos = 3;
  printf("Given array elements are :\n");
  for(i = 0; i<5; i++) {
    printf("arr[%d] = %d,  ", i, arr[i]);
  }
  arr[pos-1] = item;
  printf("\nArray elements after updation :\n");
  for(i = 0; i<5; i++) {
    printf("arr[%d] = %d,  ", i, arr[i]);
  }
return 0;
}
```

Output



```
Given array elements are :
arr[0] = 18,  arr[1] = 30,  arr[2] = 15,  arr[3] = 70,  arr[4] = 12,
Array elements after updation :
arr[0] = 18,  arr[1] = 30,  arr[2] = 50,  arr[3] = 70,  arr[4] = 12,
```

Question Bank:

1. What is an Array? How to create an Array? Give its advantages and disadvantages.
2. Can the size of an array change at run time?
3. How to sort an Array?
4. How to get largest and smallest number in an array?

5. What is the logic to reverse the array?
6. Given a sorted array **arr[]** of size N and a number **X**, find the number of occurrences of **X** in given array.
7. List and explain different operations performed on the array.
8. How to compare Two Arrays?
9. How to declare multidimensional array?

Experiment No. 2

Aim: - Write a program to demonstrate Bubble Sorting and Insertion Sorting techniques using arrays.

Objective: - Consider a student database of SY IT class (at least 10 records). It contains different fields of every student like Roll No, Name and SGPA.

Prepare a roll call list of students, Arrange according to roll numbers in ascending order, Arrange list of students alphabetically.

Mapping with CO1: Describe basics of data structures and analyze complexity of various algorithms

Theory: - Bubble Sort is an algorithm which is used to sort N elements that are given in a memory for eg: an Array with N number of elements. Bubble Sort compares the entire elements one by one and sort them based on their values.

It is called Bubble sort, because with each iteration the smaller element in the list bubbles up towards the first place, just like a water bubble rises up to the water surface.

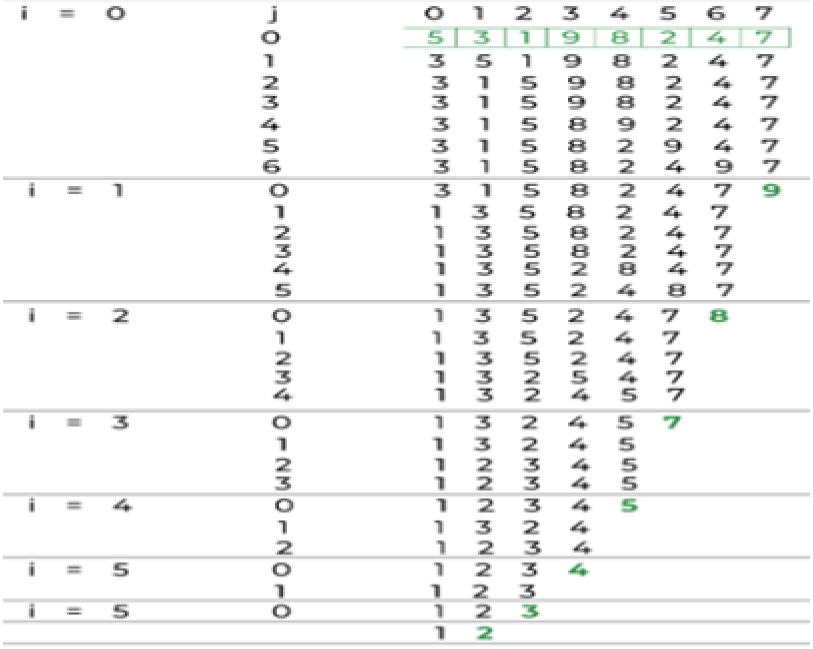
Sorting takes place by stepping through all the data items one-by-one in pairs and comparing adjacent data items and swapping each pair that is out of order.

Algorithm:

```
procedure bubbleSort( A : list of sortable items )
Start
```

```
    n = length(A)
    repeat
        swapped = false
        for i = 1 to n-1 inclusive do
            /* if this pair is out of order */
            if A[i-1] > A[i] then
                /* swap them and remember something changed */
                swap( A[i-1], A[i] )
                swapped = true
            end if
        end for
    until not swapped
end procedure
Stop
```

Example:



i = 0	j	0	1	2	3	4	5	6	7
	0	5	3	1	9	8	2	4	7
	1	3	5	1	9	8	2	4	7
	2	3	1	5	9	8	2	4	7
	3	3	1	5	9	8	2	4	7
	4	3	1	5	8	9	2	4	7
	5	3	1	5	8	2	9	4	7
	6	3	1	5	8	2	4	9	7
i = 1	0	3	1	5	8	2	4	7	9
	1	1	3	5	8	2	4	7	
	2	1	3	5	8	2	4	7	
	3	1	3	5	8	2	4	7	
	4	1	3	5	2	8	4	7	
	5	1	3	5	2	4	8	7	
i = 2	0	1	3	5	2	4	7	8	
	1	1	3	5	2	4	7		
	2	1	3	5	2	4	7		
	3	1	3	2	5	4	7		
	4	1	3	2	4	5	7		
i = 3	0	1	3	2	4	5	7		
	1	1	3	2	4	5			
	2	1	2	3	4	5			
	3	1	2	3	4	5			
i = 4	0	1	2	3	4	5			
	1	1	3	2	4				
	2	1	2	3	4				
i = 5	0	1	2	3	4				
	1	1	2	3					
i = 5	0	1	2	3					
		1	2						

Optimizing bubble sort:

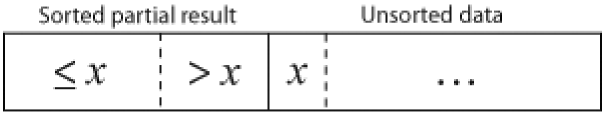
The bubble sort algorithm can be easily optimized by observing that the n-th pass finds the n-th largest element and puts it into its final place. So, the inner loop can avoid looking at the last n−1 items when running for the n-th time:

```
procedure bubbleSort( A : list of sortable items )
Start
  n = length(A)
  repeat
    swapped = false
    for i = 1 to n-1 inclusive do
      /* if this pair is out of order */
      if A[i-1] > A[i] then
        /* swap them and remember something changed */
        swap( A[i-1], A[i] )
        swapped = true
      end if
    end for
    n = n -1
  until not swapped
end procedure
Stop
```

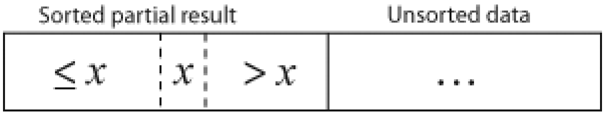
Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list. In the each iteration insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

Sorting is typically done in-place, by iterating up the array, growing the sorted list behind it. At each array-position, it checks the value there against the largest value in the sorted list (which happens to be next to it, in the previous array-position checked). If larger, it leaves the element in place and moves to the next. If smaller, it finds the correct position within the sorted list, shifts all the larger values up to make a space, and inserts into that correct position.

The resulting array after k iterations has the property where the first k + 1 entries are sorted ("+1" because the first entry is skipped). In each iteration the first remaining entry of the input is removed, and inserted into the result at the correct position, thus extending the result:

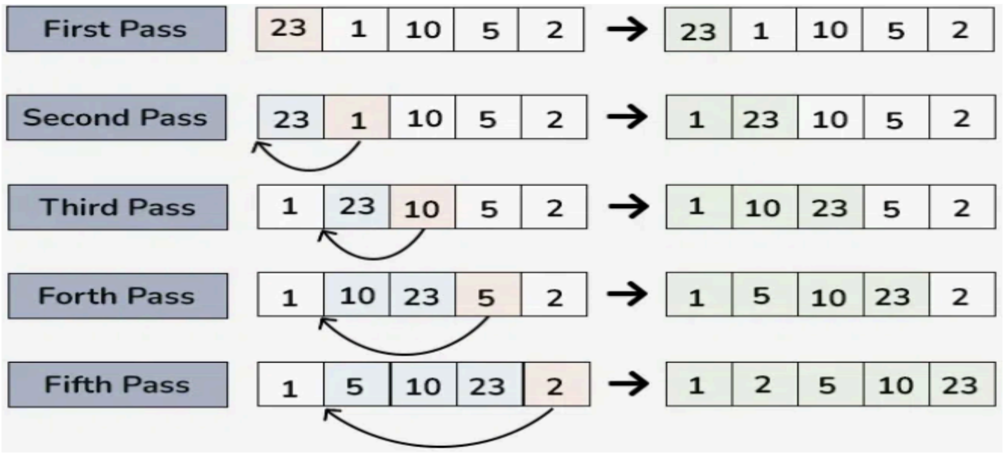


Becomes



with each element greater than *x* copied to the right as it is compared against *x*.

Consider an array having elements: {23, 1, 10, 5, 2}



First Pass	23	1	10	5	2	23	1	10	5	2
Second Pass	23	1	10	5	2	1	23	10	5	2
Third Pass	1	23	10	5	2	1	10	23	5	2
Forth Pass	1	10	23	5	2	1	5	10	23	2
Fifth Pass	1	5	10	23	2	1	2	5	10	23

Most common variant of insertion sort, which operates on arrays, can be described as follows:

- Suppose there exists a function called *Insert* designed to insert a value into a sorted sequence at the beginning of an array. It operates by beginning at the end of the sequence and shifting each element one place to the right until a suitable position is found for the new element. The function has side effect of overwriting the value stored immediately after the sorted sequence in the array.
- To perform an insertion sort, begin at the left-most element of the array and invoke *Insert* to insert each element encountered into its correct position. The ordered sequence into which the element is inserted is stored at the beginning of the array in the set of indices already examined. Each insertion overwrites a single value: the value being inserted.

Algorithm:

Step 1 − If it is the first element, it is already sorted. return 1;

Step 2 − Pick next element

Step 3 − Compare with all elements in the sorted sub-list

Step 4 − Shift all the elements in the sorted sub-list that is greater than the value to be sorted

Step 5 − Insert the value

Step 6 − Repeat until list is sorted

Pseudocode:

```
Start
procedure insertionSort( A : array of items )
  int holePosition
  int valueToInsert
  for i = 1 to length(A) inclusive do:
    /* select value to be inserted */
    valueToInsert = A[i]
    holePosition = i
    /*locate hole position for the element to be inserted */
    while holePosition > 0 and A[holePosition-1] > valueToInsert do:
      A[holePosition] = A[holePosition-1]
      holePosition = holePosition -1
    end while
    /* insert the number at hole position */
    A[holePosition] = valueToInsert
  end for
end procedure
Stop
```

Conclusion:

In Bubble Sort, n-1 comparisons will be done in 1st pass, n-2 in 2nd pass, n-3 in 3rd pass and so on. Therefore, time complexity becomes O (n^2).This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where **n** is the number of items.

In insertion sort, the best case input is an array that is already sorted, a linear running time (i.e. O(n)). During the each iteration, the first remaining element of the input is only compared with the right-most element of the sorted subsection of the array. The worst case input is an array sorted in reverse order. In these cases the every iteration of the inner loop will scan and shift the entire sorted subsection of the array before inserting the next element. This gives insertion sort a quadratic running time (i.e., O(n^2)).The average case is also quadratic(i.e., O(n^2)), which makes insertion sort impractical for sorting large arrays. However, insertion sort is one of the fastest algorithms for sorting very small arrays.

Question Bank:

1. Explain the sorting?
2. What are the different types of sorts in data structures?
3. How many passes are required in the bubble sort and insertion sort?
4. What is the time complexity of bubble sort and insertion sort?

Experiment No. 3

Aim: - Write a program to demonstrate singly linked list and double linked list.

Objective: - Implement singly linked list and double linked list for following

- Maintain club member's information: Department of Information Technology has student's club named 'COMIT'. Students of Second, third and final year of department can be granted membership on request. Similarly one may cancel the membership of club. First node is reserved for president of club and last node is reserved for secretary of club. Store the student MIS registration no. and Name. Write functions to a) Add and delete the members as well as president or even secretary. b) Compute total number of members of club c) Display members d) Display list in reverse order using recursion e) Two linked lists exists for two divisions. Concatenate two lists.
- Store the appointment schedule for day. Appointments are booked randomly using linked list. Set start and end time and min and max duration for visit slot. Write functions for
  a) Display free slots
  b) Book appointment
  c) Cancel appointment (check validity, time bounds, availability etc)

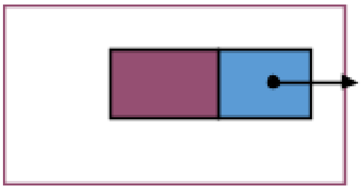
d) Sort list based on time

e) Sort list based on time using pointer

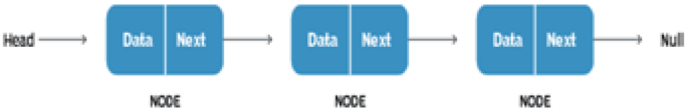
Mapping with CO2: Create Linked list, Stack, Queue and use hashing & Collision resolution Techniques to solve complex problems from different domains

Theory:-

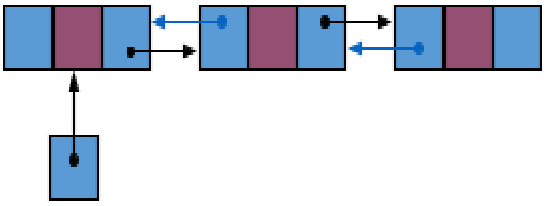
Singly Linked List is a linear and unidirectional data structure, in which elements are not stored at contiguous memory locations, where data is saved on the nodes, and each node is connected via a link to its next node. Each node contains a data field and a link to the next node. Singly Linked Lists can be traversed in only one direction, whereas Doubly Linked List can be traversed in both directions.



Node



Singly Linked List



Doubly Linked List

Unknown number of elements: When you don't know how many elements you need to store during the program runtime, you can use the Linked List. Memory is allocated dynamically as you add elements to the linked lists.

Random access: In a scenario, where you don't need to use the random access from the elements, you can use the Linked List.

Insertion in the middle: Insertion in the middle of an array is a complex task , because you need to push other elements to the right. However, a linked List allows you to add an element to any position you want.

Operations of Singly Linked List:

Singly Linked List is good for dynamically allocating memory. It provides all the basic operations of the linked list

- IsEmpty: determine whether or not the list is empty
- InsertNode: insert a new node at a particular position (at head/tail, after/before a node)
- FindNode: Search a node with a given value
- DeleteNode: delete a node with a given value (at head/tail)
- DisplayList: print all the nodes in the list
- Traversing the Linked List, Updating Linked List, Merging two linked lists

Pseudo-codes:

1) Singly Linked List

Insertion at the head of a Singly Linked List

```
function insertAtHead( head, value ):
    newNode = Node(value)
    if head is NULL:
        head = newNode
        return head
    else:
        newNode.next = head
        return newNode
```

Insertion at the end of a Singly Linked List

```
function insertAtEnd( head, value ):
    newNode = Node(value)
    if head is NULL:
        head = newNode
        return head
    while head.next is not NULL:
        then head = head.next
    head.next = newNode
    newNode.next = NULL
```

Insertion after a node in a Singly Linked List

```
function insertAfter( head, value, searchItem ):
    newNode = Node(value)
    while head.value equals searchItem:
        then head = head.next
    newNode.next = head.next.next
    head.next = newNode
```

Insertion before a node in a Singly Linked List

```
function insertBefore( head, value, searchItem ):
    newNode = Node(value)
    while head.next.value equals searchItem:
        then head = head.next
    newNode.next = head.next.next
    head.next = newNode
```

Delete the head of the singly linked list

```
function deleteHead( head ):
    temp = head
    head = head.next
    free( temp )
    return head
```

Delete the tail of the singly linked list

```
function deleteTail( head ):
    while head.next.next is not NULL:
        head = head.next
    free( head.next )
    head.next = NULL
```

Search and delete a node from a singly linked list

```
function searchAndDelete( head, searchItem ):
        while head.next.next is not NULL  and head.next.value is not equals searchItem :
                head = head.next
        head.next = head.next.next
        delete(head.next)
```

Traverse a singly linked list

```
function traverse( head ):
                while head.next is not NULL:
                        print the value of the head
                        head = head.next
```

2) Doubly linked list

Prev: Each node is linked to its previous node. It is used as a pointer or link.

Next: Each node is linked to its next node. It is used as a pointer or link.

Data: This is used to store data in a node. "Data" can hold other Data Structures inside it. For example, string, dictionary, set, hashmap, etc can be stored in the "Data".

Operations of Doubly Linked List

- Insertion in front
- Insertion in the tail or last node
- Insertion after a node
- Insertion before a node
- Deletion from front
- Deletion from tail
- Search and delete a node
- Traverse head to tail
- Traverse tail to head

Insertion in front of Doubly Linked List

```
function insertAtFront (ListHead, value):
  newNode = Node()
  newNode.value = value
  ListHead.prev = NewNode
  NewNode.next = ListHead
  newNode.prev = NULL
  return ListHead
```

Insertion at the end of Doubly Linked List

```
function insertAtTail(ListHead, value):
  newNode = Node()
  newNode.value = value
  newNode.next = NULL
  while ListHead.next is not NULL:
          then ListHead = ListHead.next
  newNode.prev = ListHead
  ListHead.next = newNode
  return ListHead
```

Insertion after a node

```
function insertAfter(ListHead, searchItem, value):
  List = ListHead
  NewNode = Node()
  NewNode.value = value
  while List.value is not equal searchItem
          then List = ListHead.next
  List = List.next
  NewNode.next = List.next
  NewNode.prev = List
  List.next = NewNode
```

Insertion before a node

```
function insertBefore(ListHead, searchItem, value):
  List = ListHead
  NewNode = Node()
  NewNode.value = value
  while List.next.value is not equal searchItem
        then List = ListHead.next
  NewNode.next = List.next
  NewNode.prev = List
  List.next = NewNode
```

Delete the head of the doubly linked list

```
function deleteHead(ListHead):
  PrevHead = ListHead
  ListHead = ListHead.next
  ListHead.prev = NULL
  PrevHead.next = NULL
  free memory(PrevHead)
  return ListHead
```

Delete the tail of the doubly linked list.

```
function DeleteTail( ListHead ):
  head = ListHead
  while ListHead.next is not NULL:
        ListHead = ListHead.next
  Tail = ListHead.next
  ListHead.next = NULL
  free memory( Tail )
  return head
```

Search and Delete Operation

```
function SearchAndDelete(ListHead, searchItem):
  head = ListHead
  while ListHead.next.value not equals searchItem:
        head = head.next
  deleteNode = head.next
  head.next = head.next.next
  head.next.prev = head
  deleteNode.next, deleteNode.next = NULL
  free memory(deleteNode)
  return ListHead
```

Traverse a doubly linked list from forward

```
function traverseFromFront(ListHead):
  head = ListHead
  while head not equals to NULL:
        print head.data
        head = head.next
  return ListHead
```

Traverse a doubly linked list from the backward

```
function traverseFromBack(ListHead):
  head = ListHead
  while head not equals NULL:
        head = head.next
  tail = head
  while tail not equal to NULL:
        print tail.value
        tail = tail.prev
  return ListHead
```

Conclusion:

The best case time complexity of Singly Linked List, where Insertion in the head can be done in O(1), as traverse inside of the linked list is not needed. Search and delete operation can be done in O(1) if the search element is present in the head node.

The average case time complexity, where Insertion inside a linked list will take O(n) if "n" elements are present in the Singly Linked List. Search and delete can take O(n) too, as the search element can be present in the tail node. In that case, the whole list should be traversed. The worst and average case time complexity is the same for the Singly Linked List

Space Complexity of Singly Linked List: Singly Linked List dynamically allocates memory. If we want to store n elements, it will allocate n memory unit. So, the space complexity is O(n).

The best case time complexity of doubly Linked List, Insertion in head or tail will cost O(1) because we don't need to traverse inside the linked list. The head and the tail pointer can give us access to the head and tail node with O(1) time complexity. Deletion at the head or tail will cost O(1). Searching a node will have the time complexity of O(1), because the target node can be the head node.

The average case time complexity, Insertion at the head or tail will have the time complexity of cost O(1). Deletion at the head or tail will have the time complexity of cost O(1). Searching a node will have the time complexity of O(n), because the search item can reside anywhere between the linked list. Here, "n" is the total node present in the linked list.

The worst-case time complexity of the doubly linked list will be the same as the average case.

Memory complexity is O(n), where "n" is the total number of nodes. While implementing the linked list we must free the memory that we used. Otherwise, for a larger linked list, it will cause memory leakages

Question Bank:

1. What is a Linked list?
2. Can you represent a Linked list graphically?
3. How many pointers are required to implement a simple Linked list?

4. How many types of Linked lists are there?
5. How to represent a linked list node?
6. Describe the steps to insert data at the starting of a singly linked list.
7. How to insert a node at the end of Linked list?
8. How to delete a node from linked list?
9. How to reverse a singly linked list?
10. What is the difference between singly and doubly linked lists?
11. What are the applications that use Linked lists?
12. What will you prefer to use a singly or a doubly linked lists for traversing through a list of elements?

Experiment No. 4

Aim: - Write a program to search a particular student according to SGPA using Binary Search, and Hashing.

Objective: - Write a program to demonstrate sorting techniques using arrays for following:

Consider a student database of SY IT class (at least 10 records). Database contains different fields of every student like Name and SGPA.

Prepare a roll call list of students, arrange according to roll numbers in ascending order and Search a particular student according to student's name, SGPA using Hashing.

Mapping with CO2: Create Linked list, Stack, Queue and use hashing & Collision resolution techniques to solve complex problems from different domains

Theory:-

Binary Search Algorithm is used to find the position of a target value within a sorted array. It works by repeatedly dividing the search interval in half until the target value is found or the interval is empty. The search interval is halved by comparing the target element with the middle value of the search space.

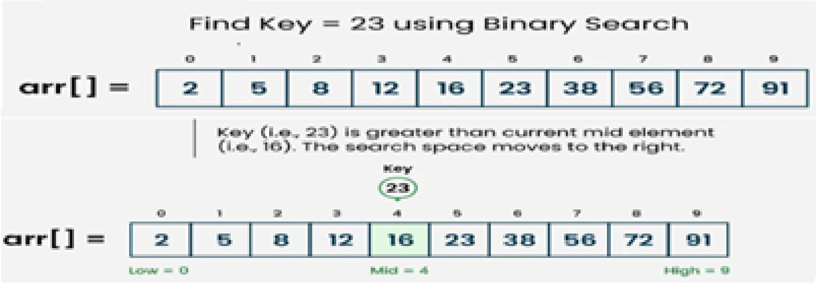
Below is the step-by-step algorithm for Binary Search:

- Divide the search space into two halves by finding the middle index "mid".
- Compare the middle element of the search space with the key.
- If the key is found at middle element, the process is terminated.
- If the key is not found at middle element, choose which half will be used as the next search space.
    - o If the key is smaller than the middle element, then the left side is used for next search.
    - o If the key is larger than the middle element, then the right side is used for next search.
- This process is continued until the key is found or the total search space is exhausted.

The Binary Search Algorithm can be implemented in the following two ways

- Iterative Binary Search Algorithm
- Recursive Binary Search Algorithm

Example: - Consider array **arr[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91}**, & **target = 23**.

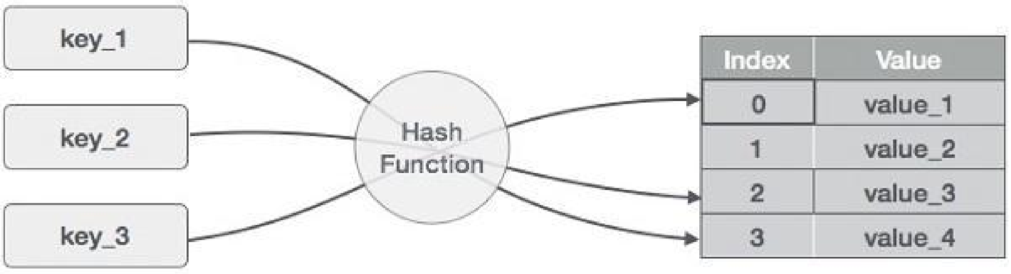

Find Key = 23 using Binary Search
arr[] =
0 1 2 3 4 5 6 7 8 9
2 5 8 12 16 23 38 56 72 91
Key (i.e., 23) is greater than current mid element (i.e., 16). The search space moves to the right.
Key
23
arr[] =
0 1 2 3 4 5 6 7 8 9
2 5 8 12 16 23 38 56 72 91
Low = 0
Mid = 4
High = 9

Algorithm:

```
binary_search(a, low, high, key)
  mid = (low + high) / 2;
  if (low <= high) {
    if (a[mid] == key)
      display   Element found at index =  mid
    else if(key < a[mid])
```

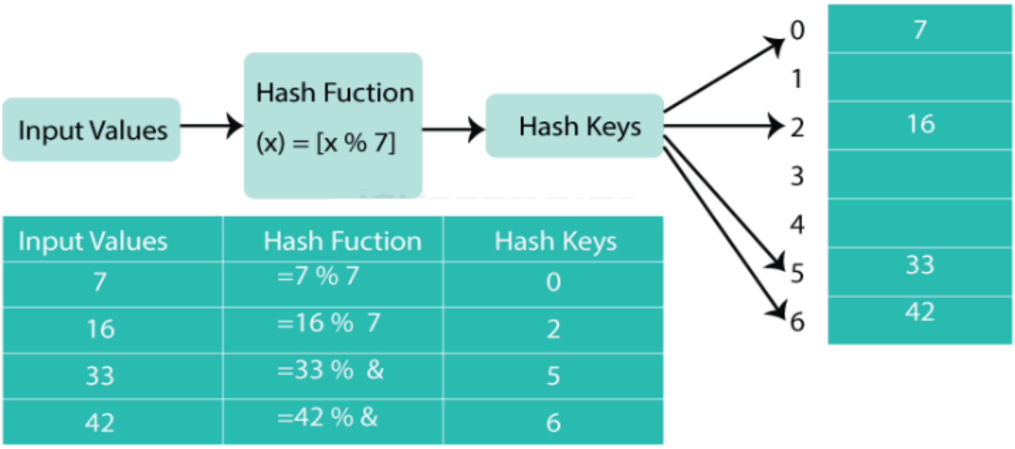
```
      binary_search(a, low, mid-1, key);
    else if (a[mid] < key)
      binary_search(a, mid+1, high, key);
  } else if (low > high)
    display   Unsuccessful Search
```

Hashing is a technique that involves converting a large amount of data into a fixed-size value or a smaller value known as a hash. The hash is generated through a hash function, used to generate the hash value maps the key (or the data to be stored) to an index within the hash table. This index is then used to store the data in the corresponding location within the array. The resulting hash value can then be used to efficiently search, retrieve, and compare data within large data sets. Hashing is useful for several reasons. Firstly, it can reduce the amount of memory required to store large data sets by converting the data into a smaller value. Secondly, it can improve the performance of algorithms by allowing for faster searching and retrieval of data. Finally, it can help to ensure data integrity by detecting duplicate data and preventing collisions (when two different keys map to the same index). This algorithm should be designed to distribute the data evenly across the hash table to reduce the likelihood of collisions. A good hash function should also be fast, simple, and deterministic (i.e. it should always produce the same output for the same input). Hashing is implemented using several methods division method, multiplication method, folding method. The division method involves taking the remainder of the key divided by the size of the hash table to determine the index. The multiplication method involves multiplying the key by a constant value and then taking the fractional part of the result to determine the index. The folding method involves breaking the key into several parts, adding them together, and then using the result to determine the index.



Index	Value
0	value_1
1	value_2
2	value_3
3	value_4

Example:



Input Values	Hash Fuction	Hash Keys
7	=7 % 7	0
16	=16 % 7	2
33	=33 % &	5
42	=42 % &	6

Pseudocodes: Hashing

```
insert(key, value):
  i = 0
  while i ≠ m:
    index = hashFn(key, i)
    if array[index] == null:
      array[index] = (key, value)
      return index
    else:
      i = i + 1
  error "hash table overflow"

find(key, value):
  i = 0
  do:
    index = hashFn(key, i)
    if array[index].key == key:
      return array[index].value
    else:
      i = i + 1
  while array[index] ≠ null or i ≠ m:
  error "hash table overflow"

remove(key, value):
  i = 0
  while i ≠ m:
    index = hashFn(key, i)
    if array[index].key == key:
      array[index] = deleted
      return
    else:
      i = i + 1

insert(key, value):  // insert operation to support deleted elements
  i = 0
  while i ≠ m:
    index = hashFn(key, i)
    if array[index] == null or deleted:
      array[index] = (key, value)
      return index
    else:
      i = i + 1
  error "hash table overflow"
```

Conclusion:

In binary search method at each step, we reduce the length of the table to be searched by half and the table is divided into two equal parts. It can be accomplished in logarithmic time in the worst case i.e. $T(n) = \theta(\log n)$. This version of the binary search takes logarithmic time in the best case.

Hashing is a powerful technique that allows for efficient searching, retrieval, and comparison of data within large data sets. It involves creating a hash function that maps input data to a fixed-size hash value, which is then used as an index within a hash table to store the data. By using hashing, programmers can improve the performance of algorithms and reduce the amount of memory required to store large data sets. Storing elements in Array takes O(1) time, and searching in it takes at least O(log n) time. This time appears to be small, but for a large data set, it can cause a lot of problems and this, in turn, makes the Array data structure inefficient.

Hashing data structure store the data and search it in constant O(1) time

Question Bank:

1. What is binary search? Discuss about its time complexity using sample dataset.
2. What are Conditions while applying Binary Search algorithm
3. Describe Iterative and Recursive Binary Search Algorithm
4. Explain applications, advantages, and disadvantages of Binary Search Algorithm.
5. What is Hashing, hash tables? explain the Components of Hashing
6. What is a Hash function? Perfect Hash function
7. What are Characteristics of good hash function
8. Types/Methods Hashing Function
9. Basic Operations and Issues in Hash function
10. Explain applications, advantages, and disadvantages of Hashing?
11. What is Hash table overflow
12. Explain open addressing and chaining Collision resolution strategies.

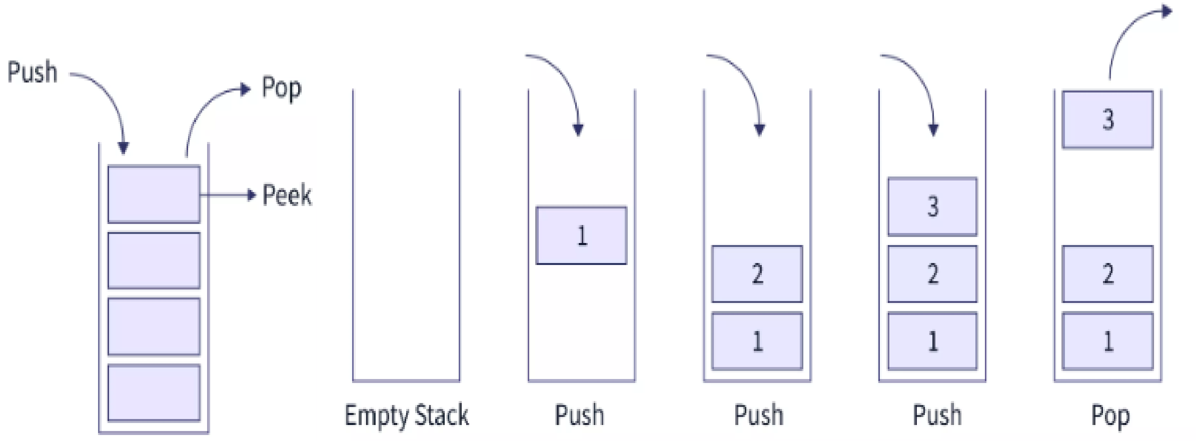
Experiment No. 5

Aim: - Write a program to demonstrate different operations performed on Stack data structure.

Objective: - Perform different ooperations like push, pop, display, peek, stack_empty, stack_full operations on stack data structure.

Mapping with CO2: Create Linked list, Stack, Queue and use hashing & Collision resolution techniques to solve complex problems from different domains.

Theory:- The stack is a type of data structure in which any data inserted is considered to be added on top of first entered data and any data deleted or removed from the data layer structure is deleted from the top only; thus this data structure works on the principle of LIFO (Last In First Out).



Basic features of Stack:

1. Stack is an ordered list of similar data type.

2. Stack is a LIFO (Last in First out) structure or we can say FILO (First in Last out).

3. push() function is used to insert new elements into the Stack and pop() function is used to remove an element from the stack. Both insertion and removal are allowed at only one end of Stack called Top.

4. Stack is said to be in Overflow state when it is completely full and is said to be in Underflow state if it is completely empty.

Basic Operations on Stack Data Structure:

In order to make manipulations in a stack, there are certain operations provided to us.

- push() to insert an element into the stack
- pop() to remove an element from the stack
- top() Returns the top element of the stack.
- isEmpty() returns true if stack is empty else false.
- isFull() returns true if the stack is full else false

Algorithms:

PUSH operation:-

1. Check if the stack is full or not. If the stack is full, then print error of overflow and exit the program by going to step 4

2. If the stack is not full, then

Do Top = Top + 1

3. Set Stack [Top] = VALUE.

4. End

POP operation:-

1. Check if the stack is empty or not. If the stack is empty, then print error of underflow and exit the program by going to step 4.

2. If the stack is not empty, then Print VALUE = STACK [Top]

3. Do Top = Top - 1

4. End

Top or Peek Operation: Returns the top element of the stack

1. Before returning the top element from the stack, we check if the stack is empty.

2. If the stack is empty (top == -1), we simply print "Stack is empty".

3. Otherwise, we return the element stored at index = top **.**

isEmpty Operation: Returns true if the stack is empty, else false

1. Check for the value of top in stack.

2. If (top == -1) , then the stack is empty so return true .

3. Otherwise, the stack is not empty so return false.

isFull Operation: Returns true if the stack is full, else false

1. Check for the value of top in stack.

2. If (top == capacity-1)**,** then the stack is full so return true.

3. Otherwise, the stack is not full so return false

Conclusion*:*

Stacks follow the LIFO principle, ensuring that the last element added to the stack is the first one removed. This behavior is useful in many scenarios, such as function calls and expression evaluation. Push and pop operations on a stack can be performed in constant time (O(1)) , providing efficient access to data It only need to store the elements that have been pushed onto them, making them memory-efficient compared to other data structures.

Question Bank:

1. What is Stack?
2. Which are the different operations that can be performed on stack?
3. Explain PUSH, POP operations on stack
4. What are the applications of stack?

Experiment No. 5b

Aim: - Write a program to a) convert an Infix expression to Postfix expression and b) check whether a given expression is parenthesized or not using stack.

Objective: -

a) Convert an Infix expression to Postfix expression and its evaluation using stack based on given conditions

i. Operands and operator, both must be single character.

ii. Input Postfix expression must be in a desired format.

iii. Only '+', '-', '*' and '/ ' operators are expected.

b) Check the well form-ness of parenthesis using stack, whether a given expression is parenthesized (balanced) or unbalanced. In any language program mostly syntax error occurs due to unbalancing delimiter such as (), {},[].

Mapping with CO2: Create Linked list, Stack, Queue and use hashing & Collision resolution techniques to solve complex problems from different domains.

Theory:- Stack is a simple data structure that allows adding and removing elements in a particular order. Every time an element is added, it goes on the top of the stack and the only element that can be removed is the element that is at the top of the stack, just like a pile of objects. In this, a stack store datatype char. Then, the user is made to enter a string, and then it iterates by the length of string and whenever it approaches an opening bracket, that bracket is inserted (pushed i.e. push function)to the top of the stack. Whenever it comes across a closing bracket, the top element of the stack is deleted (or popped i.e. pop function) with a corresponding opening bracket(as one opening and one corresponding closing bracket give empty Stack). While iterating through the length of the string, if it encounters any character other than the brackets (either opening or closing), then it won't affect our stack in any way.
In the end, the whole string has been iterated by its length times, then for the correct case our stack must be empty if it is not, then the brackets in the string entered are invalid.

Algorithm for Conversion from Infix to Postfix expression:-

1. Given an expression in the Infix form
2. Create and initialize a stack to hold operators.
3. While expression has more token (operator and operands) Do.
4. If the next token is an operand, add it to postfix string.
5. Else if the next token is end parenthesis ')', then
    a. Do until the start parenthesis '(' is opened
        i. Pop operator from the stack
        ii. Add to postfix string

```
    b. End Do
    c. Discard the Popped start parenthesis '('.
6. End if.
7. If the next token is start parenthesis '(', push it on the stack.
8. Else If the next token is an operator, Then
    a. While (Stack is not empty) AND (precedence of operator on top  of the stack is more then the
       current operator) Do
        i. Pop operator from the stack
        ii. Add to postfix string
    b. End While
    c. Push operator to stack
9. End if
10. End While
11. While stack is not empty
    a. Pop operator from the stack
    b. Add to postfix string
12. End While
13. Postfix is Created
```

Example :- Infix expression A * (B + C * D) + E becomes Postfix expression A B C D * + * E +

Follow steps:

Print operands as they arrive.

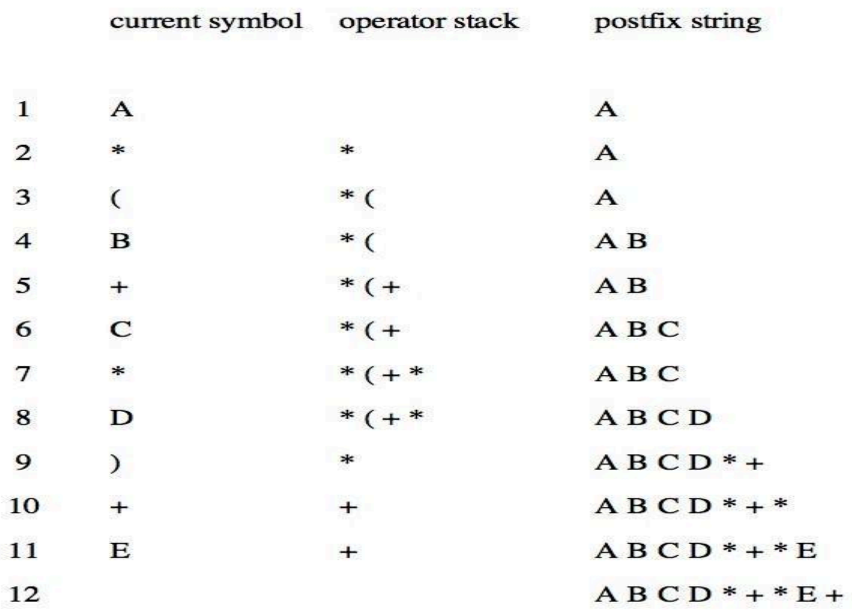
If the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.

If the incoming symbol is a left parenthesis, push it on the stack.

If the incoming symbol is a right parenthesis, pop the stack and print the operators until you see a left parenthesis. Discard the pair of parentheses.

If the incoming symbol has higher precedence than the top of the stack, push it on the stack.

If the incoming symbol has equal precedence with the top of the stack, use association. If the association is left to right, pop and print the top of the stack and then push the incoming operator. If the association is right to left, push the incoming operator.



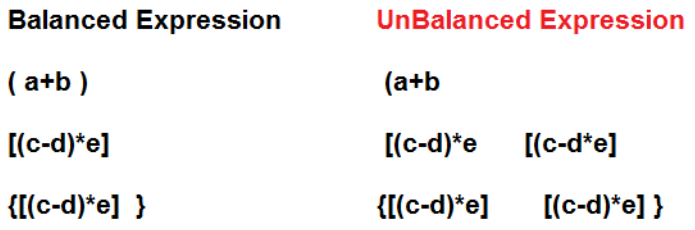
	current symbol	operator stack	postfix string
1	A		A
2	*	*	A
3	(	* (	A
4	B	* (	A B
5	+	* (+	A B
6	C	* (+	A B C
7	*	* (+ *	A B C
8	D	* (+ *	A B C D
9	)	*	A B C D * +
10	+	+	A B C D * + *
11	E	+	A B C D * + * E
12			A B C D * + * E +

Evaluate PostFix Expression 2 3 4 + * 5 *

Algorithm for Parenthesis checker:-

1. Input the expression and put it in a character stack.
2. Scan the characters from the expression one by one.
3. If the scanned character is a starting bracket (' (' or ' { ' or ' ['), then push it to the stack.
4. If the scanned character is a closing bracket (') ' or ' } ' or '] '), then pop from the stack and if the popped character is the equivalent starting bracket, then proceed. Else, the expression is unbalanced.
5. After scanning all the characters from the expression, if there is any parenthesis found in the stack or if the stack is not empty, then the expression is unbalanced.
6. Now, let us see a program to check balanced parentheses in the given expression.

Example:



Balanced Expression	UnBalanced Expression	
(a+b)	(a+b	
[(c-d)*e]	[(c-d)*e	[(c-d*e]
{[(c-d)*e] }	{[(c-d)*e]	[(c-d)*e] }

Conclusion:

By this way, we can perform expression conversion as infix to postfix and its evaluation using stack and check the given expression is well parenthesized (balanced) or unbalanced.

Question Bank:

1. What is Stack?
2. Which are the different operations that can be performed on stack?
3. Explain PUSH, POP operations on stack
4. What are the applications of stack?
5. What is infix, postfix and prefix expression?
6. Conversion – infix to postfix, infix to prefix , etc.
7. Evaluation of infix, postfix and prefix expression

Experiment No. 6

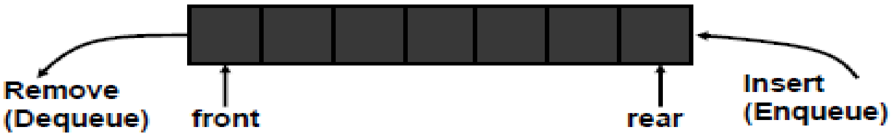
Aim: - Write a program to simulate the Queues.

Objective: -Create a queue for jobs of an operating system, write functions to add job and delete job from queue. If the operating system does not use priorities, then the jobs are processed in the order they enter the system.

Mapping with CO2: Create Linked list, Stack, Queue and use hashing & Collision resolution techniques to solve complex problems from different domains.

Theory:

A Queue is a Data Structure follows the principle of "First in, First out" (FIFO)**,** where the first element added to the queue is the first one to be removed. Queues are commonly used in various algorithms and applications for their simplicity and efficiency in managing data flow.



Basic Operations of Queue Data Structure

- Enqueue (Insert): Adds an element to the rear of the queue.
- Dequeue (Delete): Removes and returns the element from the front of the queue.
- Peek: Returns the element at the front of the queue without removing it.
- Empty: Checks if the queue is empty.
- Full: Checks if the queue is full.

Types of Queue Data Structure:

- Simple Queue or Linear Queue
- Circular Queue
- Priority Queue
- Double Ended Queue (or Deque)

Applications of Queue

- Task scheduling in operating systems
- Data transfer in network communication
- Priority queues for event processing queues for event processing

- To maintain the play list in media players in order to add and remove the songs from the play-list.
- Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.

Algorithms:

Step 1: Include all the header files which are used in the program and define a constant 'SIZE' with specific value.

Step 2: Declare all the user defined functions which are used in queue implementation.

Step 3: Create a one dimensional array with above defined SIZE (int queue[SIZE])

Step 4: Define two integer variables 'front' and 'rear' and initialize both with '-1'. (int front = -1, rear = -1)

Step 5: Then implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on queue.

enQueue(value) - Inserting value into the queue:

In a queue data structure, enQueue() is a function used to insert a new element into the queue. In a queue, the new element is always inserted at rear position. The enQueue() function takes one integer value as parameter and inserts that value into the queue. We can use the following steps to insert an element into the queue...

Step 1: Check whether queue is FULL. (rear == SIZE-1)

Step 2: If it is FULL, then display "Queue is FULL!!! Insertion is not possible!!!" and terminate the function.

Step 3: If it is NOT FULL, then increment rear value by one (rear++) and set queue[rear] = value.

deQueue() - Deleting a value from the Queue:

In a queue data structure, deQueue() is a function used to delete an element from the queue. In a queue, the element is always deleted from front position. The deQueue() function does not take any value as parameter. We can use the following steps to delete an element from the queue...

Step 1: Check whether queue is EMPTY. (front == rear)

Step 2: If it is EMPTY, then display "Queue is EMPTY!!! Deletion is not possible!!!" and terminate the function.

Step 3: If it is NOT EMPTY, then increment the front value by one (front ++). Then display queue[front] as deleted element. Then check whether both front and rear are equal (front == rear), if it TRUE, then set both front and rear to '-1' (front = rear = -1).

display() - Displays the elements of a Queue:

We can use the following steps to display the elements of a queue

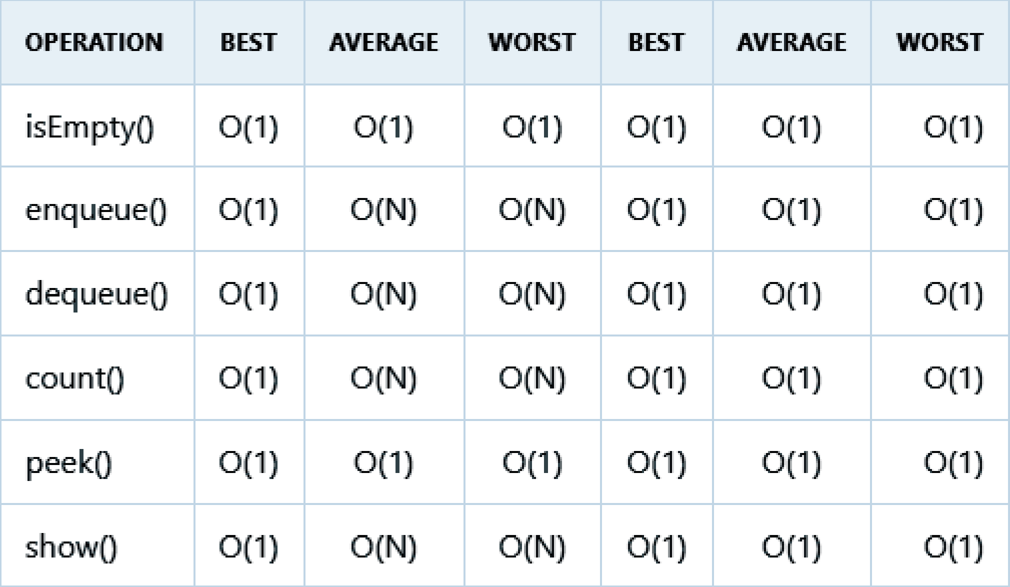
Step 1: Check whether queue is EMPTY. (front == rear)

Step 2: If it is EMPTY, then display "Queue is EMPTY!!!" and terminate the function.

Step 3: If it is NOT EMPTY, then define an integer variable 'i' and set 'i = front+1'.

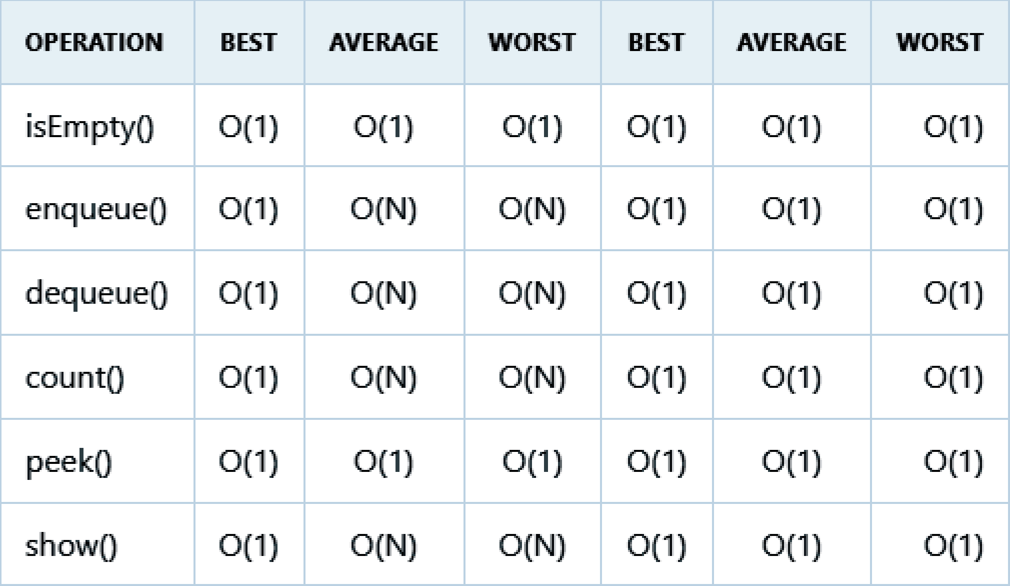
Step 3: Display 'queue[i]' value and increment 'i' value by one (i++). Repeat the same until 'i' value is equal to rear (i <= rear)

Time and space complexity using Array



OPERATION	BEST	AVERAGE	WORST	BEST	AVERAGE	WORST
isEmpty()	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
enqueue()	O(1)	O(N)	O(N)	O(1)	O(1)	O(1)
dequeue()	O(1)	O(N)	O(N)	O(1)	O(1)	O(1)
count()	O(1)	O(N)	O(N)	O(1)	O(1)	O(1)
peek()	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
show()	O(1)	O(N)	O(N)	O(1)	O(1)	O(1)

Time and space complexity using Linked List



OPERATION	BEST	AVERAGE	WORST	BEST	AVERAGE	WORST
isEmpty()	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
enqueue()	O(1)	O(N)	O(N)	O(1)	O(1)	O(1)
dequeue()	O(1)	O(N)	O(N)	O(1)	O(1)	O(1)
count()	O(1)	O(N)	O(N)	O(1)	O(1)	O(1)
peek()	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
show()	O(1)	O(N)	O(N)	O(1)	O(1)	O(1)

Conclusion*:*

The queue is a linear type of data structure used to store the elements in the FIFO technique. A queue used an array or linked list during its implementation**.**

Question Bank:

1. What is Queue?
2. What are the different operations that can be performed on queue?
3. Explain all the operations on queue
4. Which are different types of queues? Explain.

Experiment No. 6

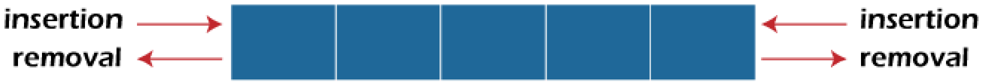
Aim: - Write a program to simulate double-ended queue (deque)

Objective: Double-ended queue (deque) with functions to add and delete elements from either end of the deque and obtain a data representation mapping a deque into a one-dimensional array.

Mapping with CO2: Create Linked list, Stack, Queue and use hashing & Collision resolution techniques to solve complex problems from different domains.

Theory:

A double-ended queue is an abstract data type similar to an simple queue, it allows you to insert and delete from both sides means items can be added or deleted from the front or rear end.

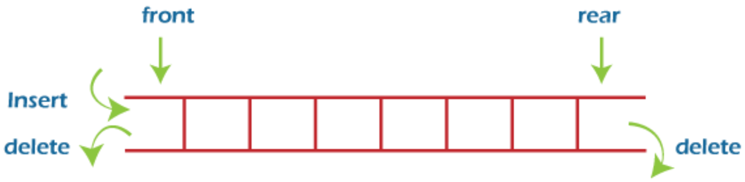


Representation of deque

Variants of a double-ended queue: Input restricted queue and Output restricted queue

Input restricted Queue

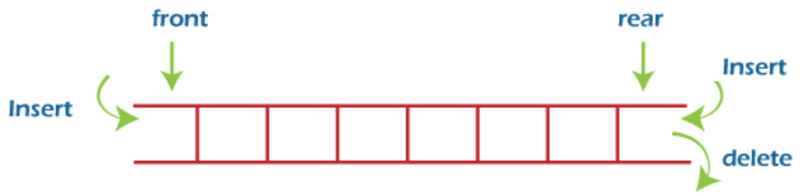
In input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.



input restricted double ended queue

Output restricted Queue

Deletion operation can be performed at only one end, while insertion can be performed from both ends.

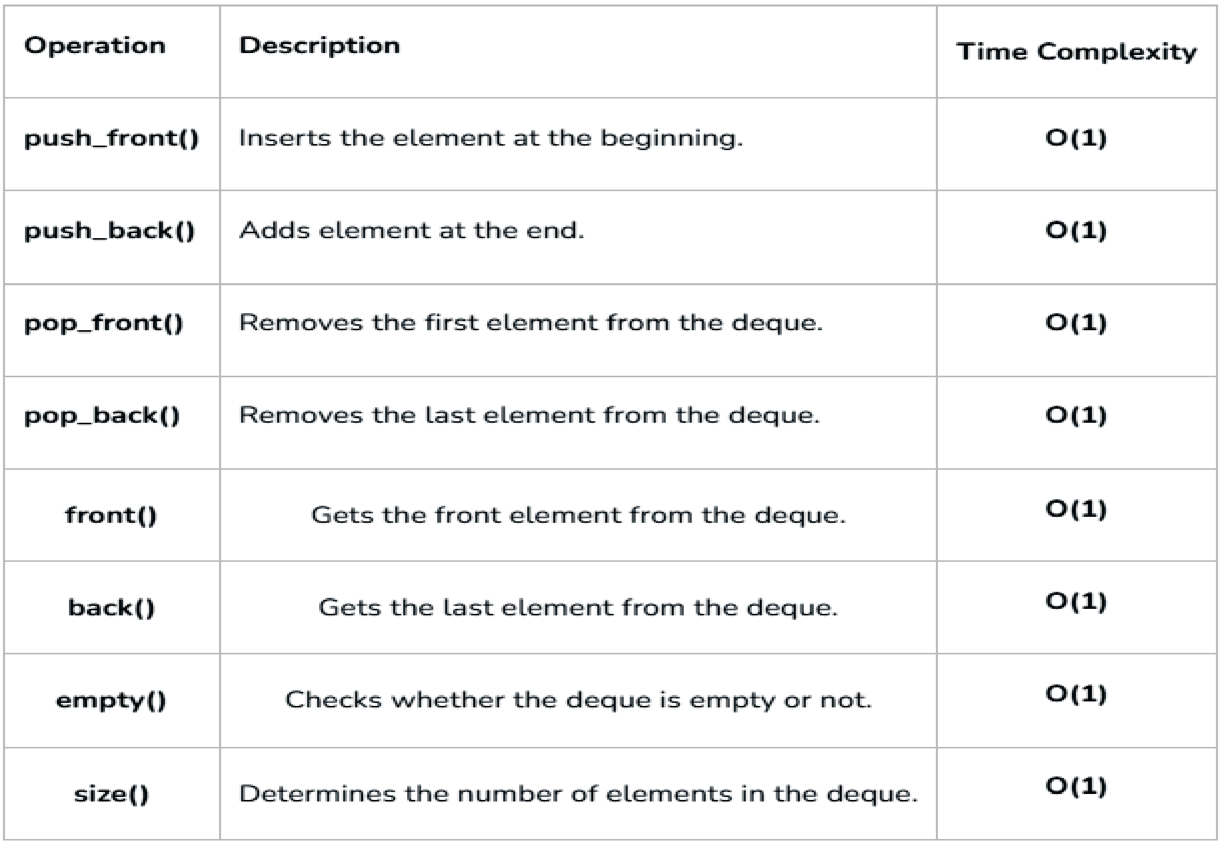


Output restricted double ended queue

Applications of deque

- It can be used as both stack and queue, as it supports both operations.
- It can be used as a palindrome checker means that if we read the string from both ends, the string would be the same.

Time complexity:



Operation	Description	Time Complexity
push_front()	Inserts the element at the beginning.	O(1)
push_back()	Adds element at the end.	O(1)
pop_front()	Removes the first element from the deque.	O(1)
pop_back()	Removes the last element from the deque.	O(1)
front()	Gets the front element from the deque.	O(1)
back()	Gets the last element from the deque.	O(1)
empty()	Checks whether the deque is empty or not.	O(1)
size()	Determines the number of elements in the deque.	O(1)

Algorithm

1. Insertion at rear end

```
Step -1: [Check for overflow] if(rear==MAX) Print("Queue is Overflow”); return;
Step-2: [Insert element] else
rear=rear+1;
q[rear]=no;
[Set rear and front pointer]
if rear=0
rear=1; if front=0
front=1;
Step-3: return
```

2. Insertion at font end

```
Step-1 : [Check for the front position] if(front<=1)
Print (“Cannot add item at front end”); return;
Step-2 : [Insert at front] else
front=front-1; q[front]=no; Step-3 : Return
```

3. Deletion from front end

```
Step-1 [ Check for front pointer] if front=0
print(" Queue is Underflow”); return;
Step-2 [Perform deletion] else
no=q[front];
print(“Deleted element is”,no); [Set front and rear pointer]
if front=rear front=0; rear=0;
else front=front+1; Step-3 : Return
```

4. Deletion from rear end

```
Step-1 : [Check for the rear pointer]
if rear=0
print("Cannot delete value at rear end"); return;
Step-2: [ perform deletion]
else
no=q[rear];
[Check for the front and rear pointer]
 if front= rear
front=0;  rear=0;
else
rear=rear-1;
print("Deleted element is",no);
 Step-3 : Return
```

Implementation of Insertion at rear end

```
void add_item_rear()
{
int num;
printf("\n Enter Item to insert : "); scanf("%d",&num); if(rear==MAX)
{
printf("\n Queue is Overflow"); return;
}
else
{
rear++; q[rear]=num; if(rear==0) rear=1; if(front==0) front=1;
}
}
```

Implementation of Insertion at font end

```
void add_item_front()
{
```

```
int num;
printf("\n Enter item to insert:"); scanf("%d",&num);

if(front<=1)
{
printf("\n Cannot add item at front end"); return;
}
else
{
front--; q[front]=num;
}
}
```

Implementation of Deletion from front end

```
void delete_item_front()
{
int num; if(front==0)
{
printf("\n Queue is Underflow\n"); return;
}
else
{
num=q[front];
printf("\n Deleted item is %d\n",num); if(front==rear)
{
front=0; rear=0;
}
else
{
front++;

}

}
}
```

Implementation of Deletion from rear end

```
void delete_item_rear()
{
int num; if(rear==0)
{
printf("\n Cannot delete item at rear end\n"); return;
}
else
{
num=q[rear]; if(front==rear)
{
front=0; rear=0;
}
else
{
rear--;
printf("\n Deleted item is %d\n",num);
}
}
}
```

Conclusion:

A double-ended queue (deque) allows elements to be added to or removed from either end i.e. items can be added to the back and removed from the front, or vice versa. This makes deques different from linear queues, which have restrictions on where elements can be inserted and deleted.

Experiment No. 7

Aim: - Write a program to demonstrate **tree.**

Objective: - Construct the Binary tree using linked list and perform the inorder, preorder, post order traversals (recursive and non-recursive).

Mapping with CO3: Analyze the operations of a nonlinear-based abstract data type on the various structured data.

Experiment No. 7

Aim: - Write a program to demonstrate **tree.**

Objective: - A book consists of chapters, the chapters consist of sections and sections consist of subsections. Construct a tree and print the nodes and find the time and space requirements of the method.

Mapping with CO3: Analyze the operations of a nonlinear-based abstract data type on the various structured data.

Experiment No. 8

Aim: - Write a program to construct **Binary search tree**

Objective: - Construct Binary search tree by inserting the values in the order given. Begin with an empty binary search tree and after constructing the tree

i. Insert new node

ii. Find number of nodes in longest path

iii. Minimum data value found in the tree

iv. Change a tree so that roles of the left and right pointers are swapped at every node

v. Search a value

Mapping with CO3: Analyze the operations of a nonlinear-based abstract data type on the various structured data.

Theory:

Binary search tree is a prominent data structure used in many systems programming applications for representing and managing dynamic sets. Average case complexity of Search, Insert, and Delete Operations is O(log n), where n is the number of nodes in the tree. A binary tree in which the nodes are labeled with elements of an ordered dynamic set and the following BST property is satisfied: all elements stored in the left subtree of any node x are less than the element stored at x and all elements stored in the right subtree of x are greater than the element at x.

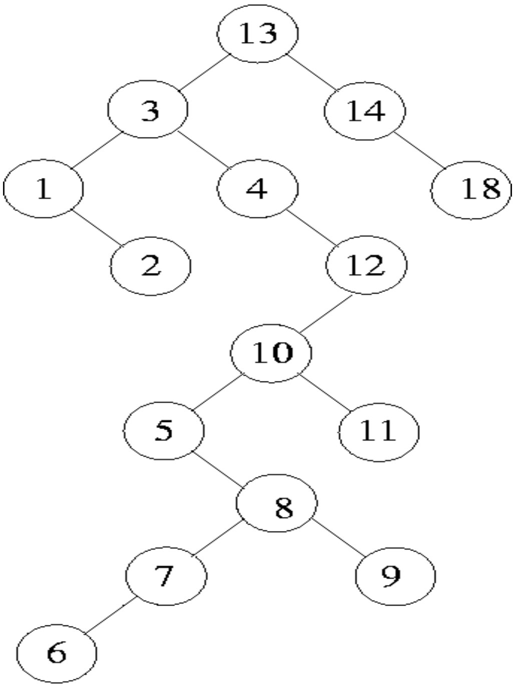
An Example: Figure shows a binary search tree. Notice that this tree is obtained by inserting the values 13, 3, 4, 12, 14, 10, 5, 1, 8, 2, 7, 9, 11, 6, 18 in that order, starting from an empty tree.

Note that inorder traversal of a binary search tree always gives a sorted sequence of the values. This is a direct consequence of the BST property. This provides a way of sorting a given sequence of keys: first, create a BST with these keys and then do an inorder traversal of the BST so created.

Highest valued element in a BST can be found by traversing from the root in the right direction all along until a node with no right link is found (we can call that the rightmost element in the BST). The lowest valued element in a BST can be found by traversing from the root in the left direction all along until a node with no left link is found (we can call that the leftmost element in the BST).

Search is straightforward in a BST. Start with the root and keep moving left or right using the BST property. If the key we are seeking is present, this search procedure will lead us to the key. If the key is not present, we end up in a null link. Insertion in a BST is also a straightforward operation. If we need to insert an element x, we first search for x. If x is present, there is nothing to do. If x is not present, then our search procedure ends in a null link. It is at this position of this null link that x will be included. If we repeatedly insert a sorted sequence of values to form a BST, we obtain a completely skewed BST. The height of such a tree is n - 1 if the tree has n nodes. Thus, the worst case complexity of searching or inserting an element into a BST having n nodes is $O(n)$.

Figure : An example of a binary search tree



Algorithm:

Conclusion:

Experiment No. 9

Aim: - Write a program to perform the **Graph** Traversal techniques **DFS** and **BFS**.

Objective: - Perform the Depth First Search and Breadth First Search using adjacency list or matrix, use the map of the historical places around Pune city as the graph, Identify the famous land marks as nodes and perform DFS and BFS.

Mapping with CO4: Apply graphs, multiway trees search techniques to store and maintain data.

Algorithm:

Conclusion:

Experiment No. 10

Aim: - Write a program to demonstrate the minimum spanning tree using **Prim's** and **Kruskal's** algorithm for following:

Objective: - There is a business with several offices that need to lease phone lines to connect them up with each other; and the phone company charges different amounts of money to connect different pairs of cities. You want a set of lines that connects all your offices with a minimum total cost.

Mapping with CO4: Apply graphs, multiway trees search techniques to store and maintain data.

Theory:

Graphs**:** A graph is a set of *vertices* and *edges* which connect them. We write:

$$\mathbf{G = (V,E)}$$

where **V** is the set of vertices and the set of edges,

$$\mathbf{E = \{ (v_i,v_j) \}}$$

Where $\mathbf{v_i}$ and $\mathbf{v_j}$ are in **V**.

Paths**:** A *path*, p, of length, k, through a graph is a sequence of connected vertices:

$$\mathbf{p = <v_0,v_1,\dots,v_k>}$$

where, for all **i** in (0,**k**-1):

$$\mathbf{(v_i,v_{i+1})} \text{ is in } \mathbf{E}.$$

Cycles**:** A graph contains no *cycles* if there is no path of non-zero length through the graph, p =<v_0, v_1,...,v_k> such that $v_0 = v_k$.

Spanning Trees: A *spanning tree* of a graph, G, is a set of |V|-1 edges that connect all vertices of the graph.

Minimum Spanning Tree:

Given a connected, undirected graph, a spanning tree of that graph is a subgraph which is a tree and connects all the vertices together. A single graph can have many different spanning trees. We can also assign a *weight* to each edge, which is a number representing how unfavorable it is, and use this to assign a weight to a spanning tree by computing the sum of the weights of the edges in that spanning tree. A minimum spanning tree (MST) or minimum weight spanning tree is then a spanning tree with weight less than or equal to the weight of every other spanning tree. More generally, any undirected graph (not necessarily connected) has a minimum spanning forest, which is a union of minimum spanning trees for its connected components.

In general, it is possible to construct multiple spanning trees for a graph, **G**. If a cost, $\mathbf{c_{ij}}$, is associated with each edge, $\mathbf{e_{ij}= (v_i,v_j)}$, then the minimum spanning tree is the set of edges, $\mathbf{E_{span}}$, forming a spanning tree, such that: $\mathbf{C = sum(}\ \mathbf{c_{ij}}$ | all $\mathbf{e_{ij}}$ in $\mathbf{E_{span}}$) is a minimum.

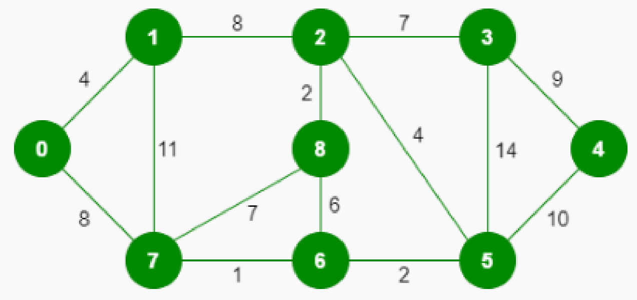


Figure: Graph to find MST

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 continuously increases the size of a tree starting with a single vertex until it spans all the vertices.

1) Input: A connected weighted graph with vertices V and edges E.
2) Initialize: $\mathbf{V_{new}= \{x\}}$, where x is an arbitrary node (starting point) from V, $\mathbf{E_{new}= \{\}}$

3) Repeat until $\mathbf{V_{new}}$**= V:**

4) Choose edge **(u,v)** from E with minimal weight such that u is in $\mathbf{V_{new}}$ and v is not (if there are multiple edges with the same weight, choose arbitrarily)

5) Add v to $\mathbf{V_{new}}$, add (u, v) to $\mathbf{E_{new}}$

6) Output: $\mathbf{V_{new}}$ and $\mathbf{E_{new}}$ describe a minimal spanning tree

Algorithm:

```
    //  Let „adj be the adjacency matrix of graph „G having „v vertices numbered from 1 to v and having
        „e edges.
    //  Let distance, path and visited be arrays of „v elements each.
    //  Array „distance is initialized to ∞, while path array and visited array is initialized to 0
    //  Let current = 1.
    //  Let number of vertices already added to the trace be given as nv and let nv=1.
    //  Repeat steps 7, 8 and 9 while nv ≠ v.
        for i= 1 to v
            if ( adj[current][i] ≠ 0) if (
                visited[i] ≠ 1)
                    if ( distance[i] > adj[current][i])
                        {
                            distance[i] = adj[current][i]
                            path[i]=current
                         }
1.  min = ∞ (in program min=32767)
2.  for i= 1 to v
            if ( visited[i] ≠ 1)
                if ( distance[i] < min)
                    {
```

```
                min = distance[i] current
                = i
            }
3.  visited[current]=1
4.  nv = nv+1
5.  Let c be the minimum cost, initially c=0
6.  for i=2 to v
7.  c = c+ distance[i]
8.  for i = 2 to v
9.  Display vertex i is connected to vertex path[i].
10. Stop
```

Kruskal'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. If the graph is not connected, then it finds a minimum spanning forest (a minimum spanning tree for each connected component). Kruskal's algorithm is an example of a greedy algorithm.

This algorithm is continually to select the edges in the order of smallest weights and accepts an edge if it does not form a cycle. Initially the forest consists of **n** single node trees (and no edges). At each step, we add one (the cheapest one) edge so that it joins two trees together. If it were to form a cycle, it would simply link two nodes that were already part of a single connected tree, so that this edge would not be needed.

Algorithm:

```
1.  Let G be a connected weighted graph having v vertices and e edges.
2.  Let x be a set of all edges of G arranged in increasing order of weights.
3.  Let T be the minimum spanning tree which is initially empty.
4.  while( T contains lesser than v-1 edges AND x is not empty)
        {
5.          Let w be the next edge of set x. i.e. an edge of lowest cost in x.
6.          Remove w from x.
7.          if ( w does not create a cycle in T)
                    Add w to T
            else
                    Discard w
            }
8.  Stop
```

Conclusion:

The time required by Prim's algorithm is $O(|V|^2)$. It will be reduced to $O(|E|\log|V|)$ if heap is used to keep {v: L(v) < infinity}.

A simple implementation using an adjacency matrix graph representation and searching an array of weights to find the minimum weight edge to add requires $O(V^2)$ running time. Using a simple binary heap data structure and an adjacency list representation, Prim's algorithm can be shown to run in time $O(E \log V)$ where E is the number of edges and V is the number of vertices

The time required by Kruskal's algorithm is $O(|E|\log|EV|)$. Using a simple binary heap data structure and an adjacency list representation, Kruskal's algorithm can be shown to run in time $O(E \log V)$ where E is the number of edges and V is the number of vertices.