SSBT's College of Arts, Commerce & Science, Bambhori, Jalgaon

Bachelor Of Computer Application (B.C.A)

Experiment: 01

DOP: **DOC:**

Aim: Write a program to implement Bubble sort

Theory: Bubble sort is a simple sorting algorithm that is easy to understand and implement. This algorithm repeatedly compares the adjacent elements in an array or list and swaps them if they are not in the correct order. This process continues until the entire array is sorted.

Algorithm:

Step 1: Take input of the number of elements and array from the user.
Step 2: Start the outer loop, which will iterate over the array.
Step 3: Inside it, start the inner loop, which will compare adjacent elements and perform swapping.
Step 4: The process will repeat until the array is sorted.

Working of Bubble Sort Algorithm:

To understand the working of the bubble sort technique, let's take an example. Suppose an array of 5 elements [7, 3, 1, 10, 5] is given, and we need to sort. We use two for loops, the outer one will iterate over the array, and the inner loop will compare adjacent elements and perform swapping.

Iteration 1

Compare elements 7 and 3, swap them since 7>3, the new order is 3, 7, 1, 10, 5.

Compare elements 7 and 1, swap them since 7>1, the new order is 3, 1, 7, 10, 5.

Compare elements 7 and 10, not swap them since 7<10, the new order is 3, 1, 7, 10, 5.

Compare elements 10 and 5, swap them since 10>5, the new order is 3, 1, 7, 5, 10.

Iteration 2

Compare elements 3 and 1, swap them since 3>1, the new order is 1, 3, 7, 5, 10.

Compare elements 3 and 7, not swap them since 3<7, the new order is 1, 3, 7, 5, 10.

Compare elements 7 and 5, swap them from 7>5, the new order is 1, 3, 5, 7, 10

Iteration 3

Compare elements 1 and 3, not swap them since 1<3, the new order is 1, 3, 7, 5, 10.

Compare elements 3 and 5, not swap them since 3<5, the new order is 1, 3, 7, 5, 10.

Program:

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

int main()
{
   int n, i, arr[50], j, temp;

   cout<<"Enter the Size(max. 50): ";
   cin>>n;
   cout<<"Enter "<<n<<" Numbers: ";

   for(i=0; i<n; i++)
   {
     cin>>arr[i];
     cout<<"\nSorting the Array using Bubble Sort Technique..\n";
     for(i=0; i<(n-1); i++)
     {
      for(j=0; j<(n-i-1); j++)
      {
         if(arr[j]>arr[j+1])
         {
            temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp;
         }
      }
     }

   }

   for(i=0; i<n; i++)
   {
      cout<<arr[i]<<" ";
      cout<<endl;
      return 0;
   }
```

Output:

Conclusion:_____
_____
_____

Submitted By :
Name:
Sign:
Roll No :

Checked By :
Name: Ms. Pratiksha S. Patil.
Sign:

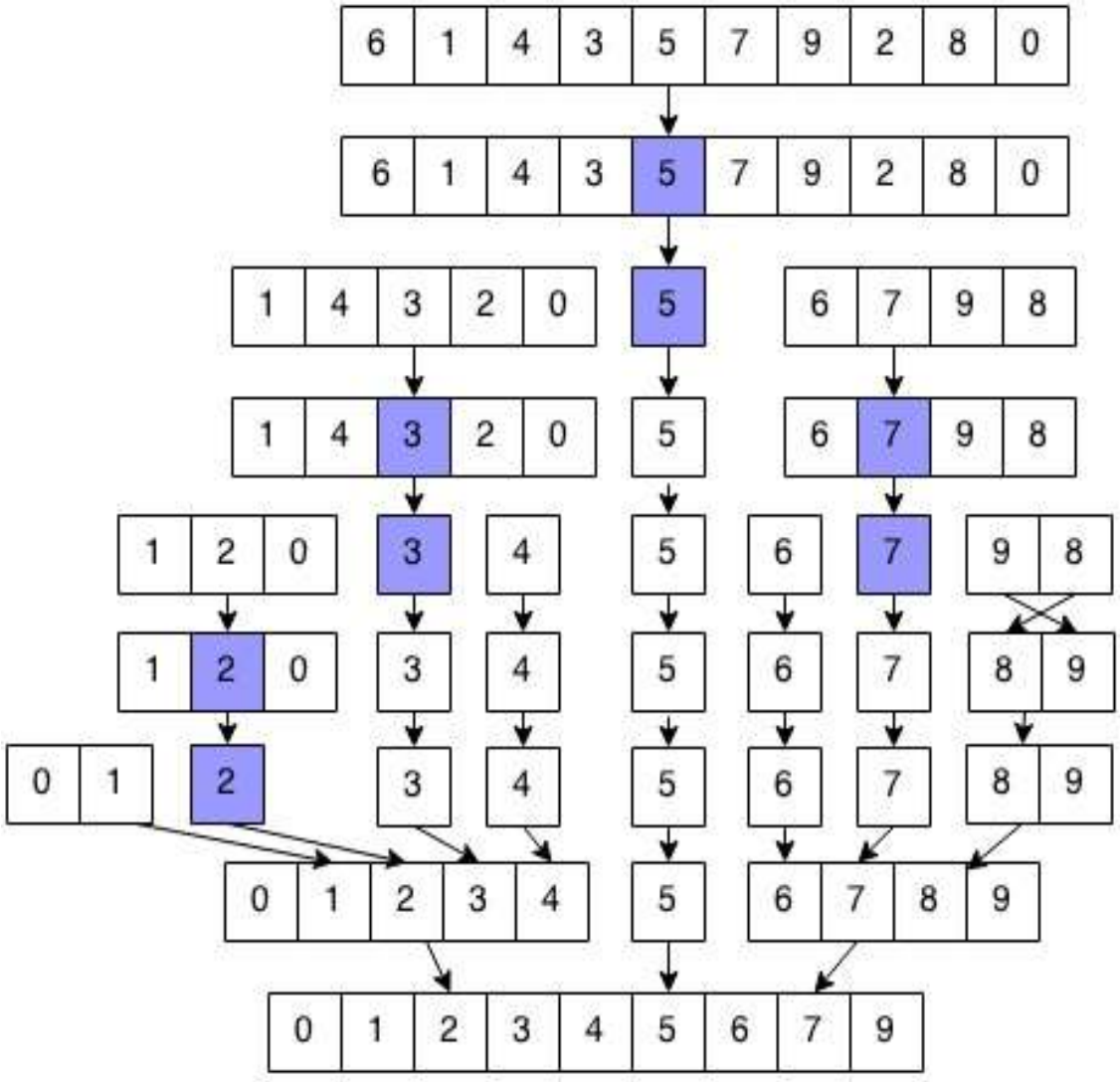
SSBT's College of Arts, Commerce & Science, Bambhori, JalgaonBachelor Of Computer Application (B.C.A)

Experiment: 02

DOP: **DOC:**

Aim: Write a program to implement Quick sort.

Theory: Quicksort is a highly efficient sorting technique that divides a large data array into smaller ones. A vast array is divided into two arrays, one containing values smaller than the provided value, say pivot, on which the partition is based. The other contains values greater than the pivot value.



Working of Quick Sort Algorithm :

The quick sort algorithm works in a divide-and-conquer fashion :

- **Divide :-**

  Choose a pivot element first from the array. After that, divide the array into two subarrays, with each element in the left sub-array being less than or equal to the pivot element and each element in the right sub-array being greater. This will continue until only a single element is left in the sub-array.

- **Conquer :**

  Recursively, sort the two subarrays formed with a quick sort.

- **Merge :**

  Merge the already sorted array.

Program:

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

void swap(int* a, int* b)
{
   int t = *a;
   *a = *b;
   *b = t;
}

int partition (int arr[], int low, int high)
{
   int pivot = arr[high];
   int i = (low - 1);

   for(int j = low; j <= high- 1; j++)
   {
      if (arr[j] <= pivot)
      {
         i++;
         swap(&arr[i], &arr[j]);
      }
   }
   swap(&arr[i + 1], &arr[high]);
   return (i + 1);
}


void quickSort(int arr[], int low, int high)
{
   if (low < high)
   {
            int pivot = partition(arr, low, high);

      quickSort(arr, low, pivot - 1);
      quickSort(arr, pivot + 1, high);
   }
}

void displayArray(int arr[], int size)
{
   int i;
   for (i=0; i < size; i++)
      cout<<arr[i]<<"\t";

}
```

```
int main()
{
   int arr[] = {12,23,3,43,51,35,19,45};
   int n = sizeof(arr)/sizeof(arr[0]);
   cout<<"Input array"<<endl;
   displayArray(arr,n);
   cout<<endl;
   quickSort(arr, 0, n-1);
   cout<<"Array sorted with quick sort"<<endl;
   displayArray(arr,n);
   return 0;
}
```

Output:

Input array: 12 23 3 43 51 35 19 45

Array sorted with quicksort: 3 12 19 23 35 43 45 51

Conclusion:_____
_____
_____

Submitted By :
Name:
Sign:
Roll No :

Checked By :
Name: Ms. Pratiksha S. Patil.
Sign:

SSBT's College of Arts, Commerce & Science, Bambhori, Jalgaon

Bachelor Of Computer Application (B.C.A)

Experiment: 03

DOP: **DOC:**

Aim: Write a program to implement Selection sort.

Theory: **Selection Sort** is a simple comparison-based sorting algorithm. Its primary idea is to divide the input list into two parts: a sorted sublist and an unsorted sublist. Initially, the sorted sublist is empty, and the unsorted sublist contains all the elements. The algorithm repeatedly selects the smallest (or largest, depending on the sorting order) element from the unsorted sublist and moves it to the end of the sorted sublist. This process continues until the unsorted sublist is empty and the sorted sublist contains all the elements in the desired order.

- *For the first position in the sorted array, the whole array is traversed from index 0 to 4 sequentially. The first position where **64** is stored presently, after traversing whole array it is clear that **11** is the lowest value.*

64	25	12	22	11

- *Thus, replace 64 with 11. After one iteration **11**, which happens to be the least value in the array, tends to appear in the first position of the sorted list.*

11	25	12	22	64

Second Pass:

- *For the second position, where 25 is present, again traverse the rest of the array in a sequential manner.*

11	**25**	12	22	64

- *After traversing, we found that **12** is the second lowest value in the array and it should appear at the second place in the array, thus swap these values.*

11	**12**	25	22	64

Third Pass:

- *Now, for third place, where **25** is present again traverse the rest of the array and find the third least value present in the array.*

11	12	**25**	22	64

- *While traversing, **22** came out to be the third least value and it should appear at the third place in the array, thus swap **22** with element present at third position.*

11	12	**22**	25	64

Fourth pass:

- *Similarly, for fourth position traverse the rest of the array and find the fourth least element in the array*
- *As **25** is the 4th lowest value hence, it will place at the fourth position.*

11	12	22	**25**	64

Fifth Pass:

- *At last the largest value present in the array automatically get placed at the last position in the array*
- *The resulted array is the sorted array.*

11	**12**	**22**	**25**	**64**

Applications of Selection Sort Algorithm:

The following are some applications of how to use selection sort:

- Selection sort consistently outperforms bubble and gnome sort.
- When memory writing is a costly operation, this can be useful.
- In terms of the number of writes ((n) swaps versus O(n2) swaps), selection sort is preferable to insertion sort.
- It almost always far outnumbers the number of writes made by cycle sort, even though cycle sort is theoretically optimal in terms of the number of writes.
- This is important if writes are significantly more expensive than reads, as with EEPROM or Flash memory, where each write reduces the memory's lifespan.

Algorithm of the Selection Sort Algorithm:
The selection sort algorithm is as follows:

Step 1: Set Min to location 0 in Step 1.
Step 2: Look for the smallest element on the list.
Step 3: Replace the value at location Min with a different value.
Step 4: Increase Min to point to the next element
Step 5: Continue until the list is sorted.

Code:-

```
#include <iostream.h >
using namespace std;

 void selectionSort(int arr[], int n)
{
   for (int i = 0; i < n - 1; i++)
   {
          int indexMin = i;           // Taken first index as minimum.
         for (int j = i + 1; j < n; j++)
         {
               //Making comparisons
              if (arr[j] < arr[indexMin])
              {
                  // Updating the minimum value.
                   indexMin = j;
              }
         }
         // Swapping the values using swap() method.
        swap(arr[indexMin], arr[i]);
   }

 cout << "Array after sorting:" << endl;

 for (int i = 0; i <= n - 1; i++)
 {
  // Printing the array after being sorted.
  cout << arr[i] << " ";
 }
}

int main()
{
 int array[] = {10, 8, 5, 6, 9, 15, 2};
 int size = 7;
 selectionSort(array, size);
}
```

Output:

```
Array after sorting:
2 5 6 8 9 10 15
```

Conclusion:_____

_____

_____

Submitted By :
Name:
Sign:
Roll No :

Checked By :
Name: Ms. Pratiksha S. Patil.
Sign:

SSBT's College of Arts, Commerce & Science, Bambhori, Jalgaon

Bachelor Of Computer Application (B.C.A)

Experiment: 04

DOP: **DOC:**

Aim: Write a program to implement Insertion sort.

Theory:

A step-by-step explanation of the sorting process is as follows:

Insertion sort works by iterating through a list of items, comparing each element to the ones that come before it, and inserting it into the correct position in the list. This process is repeated until the list is fully sorted.

To illustrate this process, let's use the example of a list of numbers that we want to sort in ascending order: [22, 6, 15, 48, 1].

Compare the first element, 22, to the second element, 6. Since 15 is smaller than 22, we swap them, resulting in the list [6, 22, 15, 48, 1].

Compare the second element, 22, to the third element, 15. Since 15 is smaller than 22, we swap them, resulting in the list [6, 15, 22, 48, 1].

Compare the third element, 22, to the fourth element, 48. Since 48 is larger than 22, no swap is necessary.

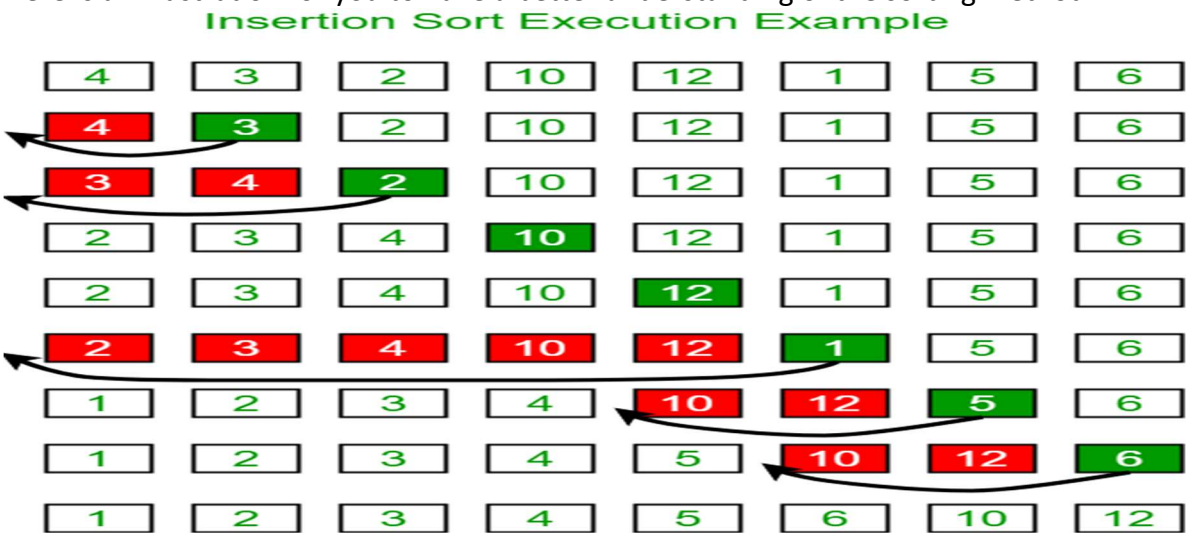
Compare the fourth element, 48, to the fifth element, 1. Since 1 is smaller than 48, we swap them, resulting in the list [6, 15, 22, 1, 48].

Compare the third element, 22, to the fourth element, 1. Since 1 is smaller than 22, we swap them, resulting in the list [6, 15, 1, 22, 9].

Compare the second element, 15, to the third element, 1. Since 1 is smaller than 15, we swap them, resulting in the list [6, 1, 15, 22, 48].

Compare the first element, 6, to the second element, 1. Since 1 is smaller than 6, we swap them, resulting in the final sorted list of [1, 6, 15, 22, 48].

Here is an illustration for you to have a better understanding of the sorting method.



Code : -

```
#include <iostream.h >
using namespace std;


void insertionSort(int arr[], int n)
{
   int i, key, j;
   for (i = 1; i < n; i++)
   {
      key = arr[i];
      j = i - 1;

     while (j >= 0 && arr[j] > key)
      {
          arr[j + 1] = arr[j];
          j = j - 1;
      }
      arr[j + 1] = key;
   }
}

void printArray(int arr[], int n)
{
   int i;
   for (i = 0; i < n; i++)
   cout << arr[i] << " ";
   cout << endl;
}

int main()
{
   int arr[] = { 12, 11, 13, 5, 6 };
   int N = sizeof(arr) / sizeof(arr[0]);

   insertionSort(arr, N);
   printArray(arr, N);

   return 0;
}
```

O/P : - 5 6 11 12 13

Conclusion:

Submitted By :	**Checked By :**
Name:	**Name: Ms. Pratiksha S. Patil.**
Sign:	**Sign:**
Roll No :	

SSBT's College of Arts, Commerce & Science, Bambhori, Jalgaon

Bachelor Of Computer Application (B.C.A)

Experiment: 5

DOP: **DOC:**

Aim: Write a program to implement linear search.

Theory: **Linear search** or **sequential search** is a method for finding an element within a list.

It sequentially checks each element of the list until a match is found or the whole list has been searched.

A linear search runs in at worst linear time and makes at most *n* comparisons, where *n* is the length of the list.

If each element is equally likely to be searched, then linear search has an average case of *n+1*/2 comparisons, but the average case can be affected if the search probabilities for each element vary.

Linear search is rarely practical because other search algorithms and schemes, such as the binary search algorithm and hash tables, allow significantly faster searching for all but short lists.[

The procedures for implementing linear search are as follows:

Step 1: First, read the search element (Target element) in the array.

Step 2: In the second step compare the search element with the first element in the array.

Step 3: If both are matched, display "Target element is found" and terminate the Linear Search function.

Step 4: If both are not matched, compare the search element with the next element in the array.

Step 5: In this step, repeat steps 3 and 4 until the search (Target) element is compared with the last element of the array.

Step 6 - If the last element in the list does not match, the Linear Search Function will be terminated, and the message "Element is not found" will be displayed.

Program:

```
#include  <iostream>
using namespace std;
int search(int array[], int n, int x)
{
   for (int i = 0; i  <  n; i++)
  {
      if (array[i] == x)
     {
        return i;
      }
   }
   return -1;
}
int main()
{
   int array[] = {2, 4, 7, 8, 0, 1, 9};
   int x = 8;
   int n = sizeof(array) / sizeof(array[0]);
   int result = search(array, n, x);
   if (result == -1)
  {
      cout  <<  "Element not found";
   }
  else
 {
      cout  <<  "Element found at index: "  <<  result;

 }

 return 0;
}
```

Element found at index: 3

Conclusion:_____

_____

_____

Submitted By :
Name:
Sign:
Roll No :

Checked By :
Name: Ms. Pratiksha S. Patil.
Sign:

SSBT's College of Arts, Commerce & Science, Bambhori, Jalgaon

Bachelor Of Computer Application (B.C.A)

Experiment: 06

DOP: **DOC:**

Aim: Write a program to implement Binary search.

Theory: The binary search algorithm is a divide and conquer algorithm that you can use to search for and find elements in a sorted array.

The algorithm is fast in searching for elements because it removes half of the array every time the search iteration happens.

So instead of searching through the whole array, the algorithm removes half of the array where the element to be searched for can't be found. It does this continuously until the element is found.

In a case where the element to be searched for doesn't exist, it returns a value of -1. If the element exists, then it returns the index of the element.

To start with, we created a method called binarySearch which had four parameters:

- array[] represents the array to be searched through.
- low represents the first element of the array.
- high represents the last element of the array.
- number_to_search_for represents the number to be searched for in array[].

Program:

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

int binarySearch(int array[], int low, int high, int number_to_search_for)
{

   while (low <= high)
   {
      int mid = low + (high - low) / 2;

      if (number_to_search_for == array[mid])
      {
         return mid;
      }

      if (number_to_search_for > array[mid])
      {
         low = mid + 1;
      }

      if (number_to_search_for < array[mid])
      {
         high = mid - 1;
      }

   }

 return -1;
}

int main(void)
{
 int arrayOfNums[] = {2,4,7,9,10,13,20};

 int n = sizeof(arrayOfNums) / sizeof(arrayOfNums[0]);

 int result = binarySearch(arrayOfNums, 0, n - 1, 13);

 if (result == -1)
 {
    printf("Element doesn't exist in the array");
 }
 else
 {
    printf("The index of the element is %d", result);
 }

 }
```

Output:_____
_____
_____

In the code above, we passed in the values of the parameters created in the binary Search method: binary Search(arrayOfNums, 0, n -1, 13).

- arrayOfNums represents the array to be searched for: {2,4,7,9,10,13,20}.
- 0 represents the first index of the array.
- n - 1 represents the last index of the array. Have a look at the code to see how n was created.
- 13 is the number to be searched for in arrayOfNums.

Conclusion:_____
_____
_____

Submitted By :
Name:
Sign:
Roll No :

Checked By :
Name: Ms. Pratiksha S. Patil.
Sign:

SSBT's College of Arts, Commerce & Science, Bambhori, Jalgaon

Bachelor Of Computer Application (B.C.A)

Experiment: 07

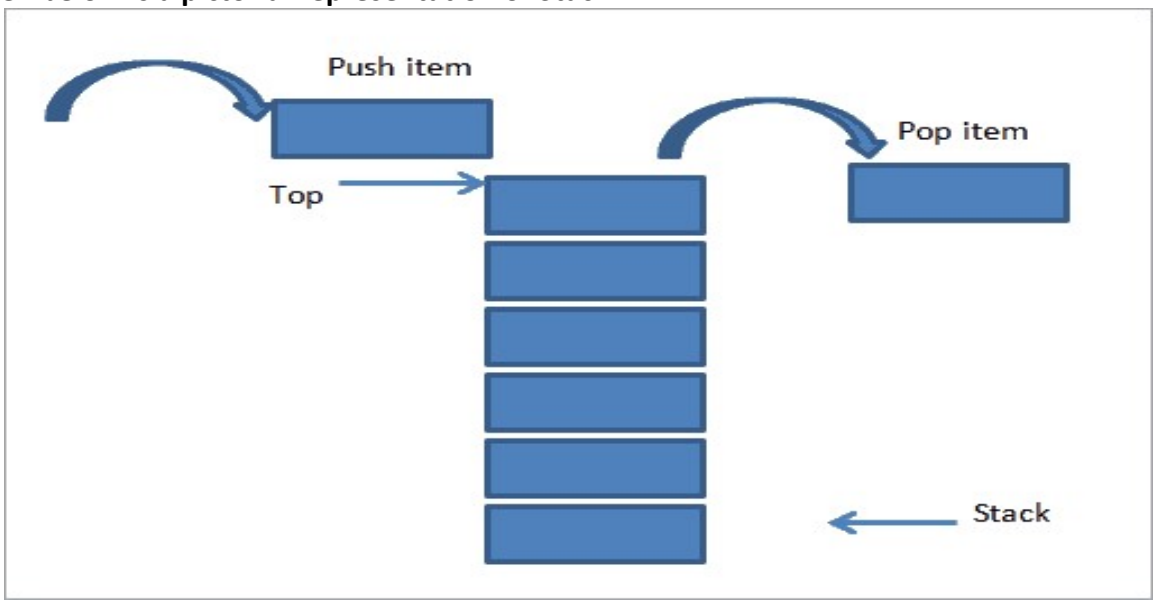
DOP: **DOC:**

Aim: Write a program implement Stack Operations: Push, Pop, Display.

Theory: Stack is a fundamental data structure which is used to store elements in a linear fashion.

Stack follows **LIFO (last in, first out)** order or approach in which the operations are performed. This means that the element which was added last to the stack will be the first element to be removed from the stack.

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**".

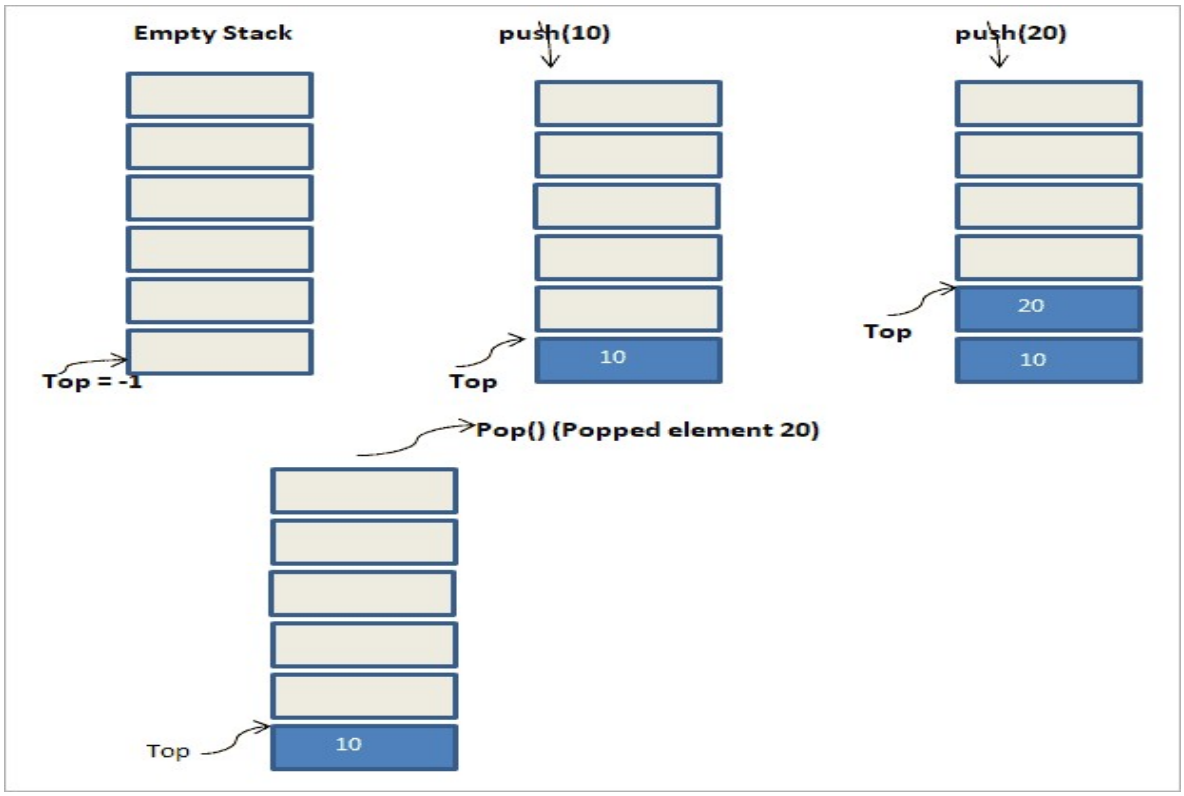
As shown in the above figure, we see that push and pop are carried out from the same end. This makes the stack to follow LIFO order. The position or end from which the items are pushed in or popped out to/from the stack is called the "**Top of the stack**".
Initially, when there are no items in the stack, 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.

Next, we will see some of the basic operations of the stack data structure that we will require while implementing the stack.

Basic Operations:

Following are the basic operations that are supported by the stack.

- **push –** Adds or pushes an element into the stack.
- **pop –** Removes or pops an element out of the stack.
- **peek –** Gets the top element of the stack but doesn't remove it.
- **isFull –** Tests if the stack is full.
- **isEmpty –** Tests if the stack is empty.




```
#include <iostream>
using namespace std;
int stack[100], n=100, top=-1;

void push(int val)
{
  if(top>=n-1)
  cout<<"Stack Overflow"<<endl;
  else
  {
    top++;
    stack[top]=val;
  }
}




void pop()
{
```

```
  if(top<=-1)
   cout<<"Stack Underflow"<<endl;
   else
   {
      cout<<"The popped element is "<< stack[top] <<endl;
      top--;
   }
}
void display()
{
   if(top>=0)
   {
      cout<<"Stack elements are:";
      for(int i=top; i>=0; i--)
      cout<<stack[i]<<" ";
      cout<<endl;
   }
   else
   cout<<"Stack is empty";
}
int main()
{
   int ch, val;
   cout<<"1) Push in stack"<<endl;
   cout<<"2) Pop from stack"<<endl;
   cout<<"3) Display stack"<<endl;
   cout<<"4) Exit"<<endl;
   do
   {
      cout<<"Enter choice: "<<endl;
      cin>>ch;
      switch(ch)
      {
         case 1:
     {
            cout<<"Enter value to be pushed:"<<endl;
            cin>>val;
            push(val);
            break;
         }
         case 2:
         {
            pop();
            break;
         }
         case 3:
         {
            display();
            break;
         }
         case 4:
         {
            cout<<"Exit"<<endl;
            break;
         }
         default:
         {
            cout<<"Invalid Choice"<<endl;
         }
```

```
}
  }while(ch!=4);
  return 0;
}
```

Output

1) Push in stack

2) Pop from stack

3) Display stack

4) Exit

Enter choice: 1

Enter value to be pushed: 2

Enter choice: 1

Enter value to be pushed: 6

Enter choice: 1

Enter value to be pushed: 8

Enter choice: 1

Enter value to be pushed: 7

Enter choice: 2

The popped element is 7

Enter choice: 3

Stack elements are:8 6 2

Enter choice: 5

Invalid Choice

Enter choice: 4

Exit

Conclusion:_____

_____

_____

Submitted By :
Name:
Sign:
Roll No :

Checked By :
Name: Ms. Pratiksha S. Patil.
Sign:

SSBT's College of Arts, Commerce & Science, Bambhori, Jalgaon

Bachelor Of Computer Application (B.C.A)

Experiment: 08

DOP: **DOC:**

Aim: Write a program to implement Queue operations: i) insert ii)delete iii)display

Theory: A Queue in C++ is similar to the Queue in real life. You all must be part of some real-life queues, like a queue in school prayer, a queue in front of the ticket window, etc.

For example, you want to buy your favourite ice cream, and you know there is a long queue to buy ice cream from the shop. What will you do? Go as early as possible; the one standing in the Queue will get the ice cream first.

In a real-life queue, the person who enters first in the Queue will be the one who Will exit first. Similarly, a queue in C++ also follows the same principle.

It processes data in a FIFO order.

Types of Queues in C++

- **Simple Queue**

It is the most basic Queue in data structure and uses the FIFO principle for inserting and removing data from the Queue.

- **Priority Queue**

Like its name, this Queue has a priority associated with its elements. It follows the FIFO principle only when two data have similar priorities. Here, data can be removed either in ascending order or descending order. It is of two types:

1. Ascending Order Priority Queue: Data with the highest priority will be removed first.
2. Descending Order Priority Queue: Data with the lowest priority will be removed first.

- **Circular Queue**

In a Circular Queue, the Front and Rear ends are connected, forming a complete circle. A circular resolved the memory wastage drawback of the simple Queue in C++.

Syntax of Queue:

Queue <data_type> queue_name;

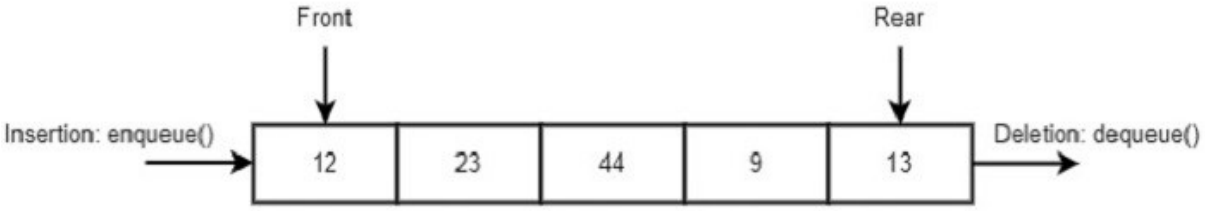
You can use two data types: integer and string, for the Queue.

For example:

1. queue<int> queue_integer :- Create an integer data type queue with name queue_integer.
2. queue<string> queue_string:- Create a string data type queue with name queue_string.

A Queue in C++ has two endpoints:

1. **Front:** Elements/data are removed from this end.
2. **Rear:** Elements/data are inserted from this end.



Queue: FIFO Operation

Program:

```
#include <iostream>
using namespace std;
int queue[100], n = 100, front = - 1, rear = - 1;

void Insert()
{
  int val;
  if (rear==n-1)
  cout<<"Queue Overflow"<<endl;
  else
  {
    if(fron==-1)
    front=0;
    cout<<"Insert the element in queue : "<<endl;
    cin>>val;
    rear++;
    queue[rear]=val;
  }
}
void Delete()
{
  if(front==-1||front>rear)
  {
    cout<<"Queue Underflow ";
    return ;
  }
  else
  {
  cout<<" deleted from queue is :"<< queue[front] <<endl;
    front++;;
  }
}
void Display()
```

```
{
  if(front==-1)
  cout<<"Queue is empty"<<endl;
  else
  {
    cout<<"Queue elements are : ";
    for (int i = front; i <= rear; i++)
    cout<<queue[i]<<" ";
      cout<<endl;
  }
}
int main()
{
  int ch;
  cout<<"1) Insert element to queue"<<endl;
  cout<<"2) Delete element from queue"<<endl;
  cout<<"3) Display all the elements of queue"<<endl;
  cout<<"4) Exit"<<endl;
  do
  {
    cout<<"Enter your choice : "<<endl;
    cin>>ch;
    switch(ch)
    {
      case 1: Insert();
      break;
      case 2: Delete();
      break;
      case 3: Display();
      break;
      case 4: cout<<"Exit"<<endl;
      break;
      default: cout<<"Invalid choice"<<endl;
    }
  } while(ch!=4);
  return 0;
}
```

The output of the above program is as follows

```
1) Insert element to queue
2) Delete element from queue
3) Display all the elements of queue
4) Exit
Enter your choice : 1
Insert the element in queue : 4
Enter your choice : 1
Insert the element in queue : 3
Enter your choice : 1
Insert the element in queue : 5
Enter your choice : 2
Element deleted from queue is : 4
Enter your choice : 3
```

Queue elements are : 3 5
Enter your choice : 7
Invalid choice
Enter your choice : 4
Exit

Conclusion:_____

_____

_____

Submitted By :
Name:
Sign:
Roll No :

Checked By :
Name: Ms. Pratiksha S. Patil.
Sign:

SSBT's College of Arts, Commerce & Science, Bambhori, Jalgaon

Bachelor Of Computer Application (B.C.A)

Experiment: 09

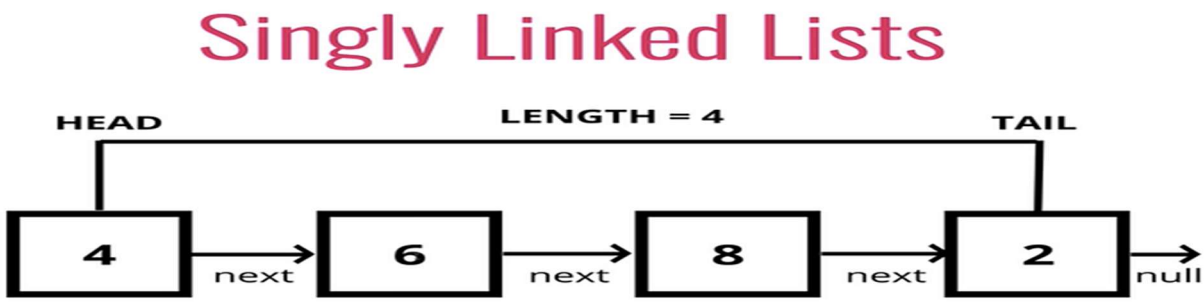
DOP: **DOC:**

Aim: Write a program to implement singly link list with operations: i) create ii)insert iii) delete

Theory: A **singly linked list** is a type of linked list that is *unidirectional*, that is, it can be traversed in only one direction from head to the last node (tail).

Each element in a linked list is called a **node**. A single node contains *data* and a pointer to the *next* node which helps in maintaining the structure of the list.

The first node is called the **head**; it points to the first node of the list and helps us access every other element in the list. The last node, also sometimes called the **tail**, points to *NULL* which helps us in determining when the list ends.



Linked List

- A Linked List is a collection of objects called nodes that are randomly stored in the memory.
- Each node contains two fields that hold a particular address where data is stored and the pointer which contains the address of the next node in the memory.

Singly-Linked List

- A Singly-linked list can be defined as the collection of an ordered set of elements.
- Nodes in the singly linked list consist of two parts such as the ***data part*** and ***link part.***
- The data part of the node stores actual information that is to be represented by the node while the linked part of the node stores the address of its immediate successor.
- In a single linked list, the address of the first node is always stored in a reference node known as **front.** Sometimes it is also known as **head**
- Always the last node of the list contains a pointer to the null.
- In the Singly linked list data navigation happened in the forwarding direction only.
- Basic operations supported by a list are ***insertion, deletion, display, and search***.

- Here we see **Insertion** and **Deletion** in detail.

Insertion Operation

The insertion into a singly linked list can be performed at different positions. Based on the position of the new node being inserted, the insertion is categorized into the following categories.

Insertion at the beginning -

It involves inserting an element at the front of the list. It needs to make the new node the head of the list.

- ***Step 1*** **-** Create a newNode with given value.
- ***Step 2*** **-** Check whether list is Empty (head == NULL)
- ***Step 3*** **-** If it is Empty then, set newNode→next = NULL and head = newNode.
- ***Step 4*** **-** If it is Not Empty then, set newNode→next = head and head = newNode.

Insertion at end of the list -

It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one. Different logics are implemented in each scenario.

- ***Step 1*** **-** Create a newNode with a given value and newNode → next as NULL.
- ***Step 2*** **-** Check whether list is Empty (head == NULL).
- ***Step 3*** **-** If it is Empty then, set head = newNode.
- ***Step 4*** **-** If it is Not Empty then, define a node pointer temp and initialize with head.
- ***Step 5*** **-** Keep moving the temp to its next node until it reaches the last node in the list (until temp → next is equal to NULL).
- ***Step 6*** **-** Set temp → next = newNode.

Insertion after the specified node -

It involves insertion after the specified node of the linked list. It needs to skip the desired number of nodes to reach the node after which the new node will be inserted.

- ***Step 1*** **-** Create a newNode with the given value.
- ***Step 2*** **-** Check whether list is Empty (head == NULL)
- ***Step 3*** **-** If it is Empty then, set newNode → next = NULL and head = newNode.
- ***Step 4*** **-** If it is Not Empty then, define a node pointer temp and initialize with head.
- ***Step 5*** **-** Keep moving the temp to its next node until it reaches the node after which we want to insert the newNode (until temp1 → data is equal to location, here location is the node value after which we want to insert the newNode).
- ***Step 6*** **-** Every time check whether temp is reached to the last node or not. If it is reached to the last node then display 'Given node is not found in the list!!! Insertion not possible!!!' and terminate the function. Otherwise, move the temp to the next node.

- ***Step 7 -*** Finally, Set 'newNode → next = temp → next' and 'temp → next = newNode'

Deletion Operation

The Deletion of a node from a singly linked list can be performed at different positions same as in insertion. Based on the position of the node being deleted, the operation is categorized into the following categories.

Deletion at the beginning -

It involves the deletion of a node from the beginning of the list. This is the simplest operation of all. It just needs a few adjustments in the node pointers.

- ***Step 1 -*** Check whether the list is Empty (head == NULL)
- ***Step 2 -*** If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminates the function.
- ***Step 3 -*** If it is Not Empty then, define a Node pointer 'temp' and initialize with head.
- ***Step 4 -*** Check whether the list is having only one node (temp → next == NULL)
- ***Step 5 -*** If it is TRUE then set head = NULL and delete temp (Setting Empty list conditions)
- ***Step 6 -*** If it is FALSE then set head = temp → next, and delete temp.

Deletion at the end of the list -

It involves deleting the last node of the list. The list can either be empty or full. A different logic is implemented for the different scenarios.

- ***Step 1 -*** Check whether list is Empty (head == NULL)
- ***Step 2 -*** If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminates the function.
- ***Step 3 -*** If it is Not Empty then, define two Node pointers 'temp1' and 'temp2', and initialize 'temp1' with head.
- ***Step 4 -*** Check whether the list has only one Node (temp1 → next == NULL)
- ***Step 5 -*** If it is TRUE. Then, set head = NULL and delete temp1. And terminate the function. (Setting Empty list condition)
- ***Step 6 -*** If it is FALSE. Then, set 'temp2 = temp1 ' and move temp1 to its next node. Repeat the same until it reaches the last node in the list. (until temp1 → next == NULL)
- ***Step 7 -*** Finally, Set temp2 → next = NULL and delete temp1.

Deletion after the specified node -

It involves deleting the node after the specified node in the list. It needs to skip the desired number of nodes to reach the node after which the node will be deleted. This requires traversing through the list.

- ***Step 1*** **-** Check whether list is Empty (head == NULL)
- ***Step 2*** **-** If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminates the function.
- ***Step 3*** **-** If it is Not Empty then, define two Node pointers 'temp1' and 'temp2', and initialize 'temp1' with head.
- ***Step 4*** **-** Keep moving the temp1 until it reaches the exact node to be deleted or to the last node. And every time set 'temp2 = temp1' before moving the 'temp1' to its next node.
- ***Step 5*** **-** If it is reached to the last node then display 'Given node not found in the list! Deletion not possible!!!'. And terminate the function.
- ***Step 6*** **-** If it is reached to the exact node which we want to delete, then check whether the list is having only one node or not
- ***Step 7*** **-** If the list has only one node and that is the node to be deleted, then set head = NULL and delete temp1 (free(temp1)).
- ***Step 8*** **-** If the list contains multiple nodes, then check whether temp1 is the first node in the list (temp1 == head).
- ***Step 9*** **-** If temp1 is the first node then move the head to the next node (head = head → next) and delete temp1.
- ***Step 10*** **-** If temp1 is not the first node then check whether it is the last node in the list (temp1 → next == NULL).
- ***Step 11*** **-** If temp1 is last node then set temp2 → next = NULL and delete temp1 (free(temp1)).
- ***Step 12*** **-** If temp1 is not first node and not last node then set temp2 → next = temp1 → next and delete temp1 (free(temp1)).

Program:

```
#include <iostream>
#include<conio.h>
#include<stdlib.h>
#include<stdio.h>
using namespace std;
struct node {
   int value;
   struct node *next;
};

void insert();
void display();
void delete_node();
int count();
typedef struct node DATA_NODE;
DATA_NODE *head_node, *first_node, *temp_node = 0, *prev_node, next_node;
int data;
```

```
int main()
{
    int option = 0;

    cout << "Singly Linked List C++ Example - All Operations\n";

    while (option < 5)
    {
        cout << "\nOptions\n";
        cout << "1 : Insert into Linked List \n";
        cout << "2 : Delete from Linked List \n";
        cout << "3 : Display Linked List\n";
        cout << "4 : Count Linked List\n";
        cout << "Others : Exit()\n";
        cout << "Enter your option:";
        scanf("%d", &option);
        switch (option)
        {
            case 1:
                insert();
                break;
            case 2:
                delete_node();
                break;
            case 3:
                display();
                break;
            case 4:
                count();
                break;
            default:
                break;
        }
    }
     return 0;
}

void insert()
{
    cout << "\nEnter Element for Insert Linked List : \n";
    scanf("%d", &data);

    temp_node = (DATA_NODE *) malloc(sizeof (DATA_NODE));

    temp_node->value = data;

    if (first_node == 0)
   {
        first_node = temp_node;
    }
   else
   {
        head_node->next = temp_node;
    }
    temp_node->next = 0;
    head_node = temp_node;
    fflush(stdin);
}
```

```
void delete_node()
{
    int countvalue, pos, i = 0;
    countvalue = count();
    temp_node = first_node;
    cout << "\nDisplay Linked List : \n";

    cout << "\nEnter Position for Delete Element : \n";
    scanf("%d", &pos);

    if (pos > 0 && pos <= countvalue)
   {
        if (pos == 1)
       {
           temp_node = temp_node -> next;
           first_node = temp_node;
           cout << "\nDeleted Successfully \n\n";
        }
       else
       {
           while (temp_node != 0)
           {
               if (i == (pos - 1))
               {
                   prev_node->next = temp_node->next;
                   if (i == (countvalue - 1))
                  {
                      head_node = prev_node;
                   }
                   cout << "\nDeleted Successfully \n\n";
                   break;
               }

              else
               {
                   i++;
                   prev_node = temp_node;
                   temp_node = temp_node -> next;
               }
           }
       }
   }
   else
   cout << "\nInvalid Position \n\n";
}

void display()
{
    int count = 0;
    temp_node = first_node;
    cout << "\nDisplay Linked List : \n";
    while (temp_node != 0)
    {
        cout << "# " << temp_node->value;
        count++;
        temp_node = temp_node -> next;
     }
    cout << "\nNo Of Items In Linked List : %d" << count;
}
```

```
int count()
        {
    int count = 0;
    temp_node = first_node;
    while (temp_node != 0)
    {
        count++;
        temp_node = temp_node -> next;
    }
    cout << "\nNo Of Items In Linked List : %d" << count;
    return count;
}
```

Output:

```
Singly Linked List Example - All Operations
        Options
1 : Insert into Linked List
2 : Delete from Linked List
3 : Display Linked List
4 : Count Linked List
Others : Exit()
Enter your option:1

Enter Element for Insert Linked List :
100

Options
1 : Insert into Linked List
2 : Delete from Linked List
3 : Display Linked List
4 : Count Linked List
Others : Exit()
Enter your option:1

Enter Element for Insert Linked List :
200

Options
1 : Insert into Linked List
2 : Delete from Linked List
3 : Display Linked List
4 : Count Linked List
Others : Exit()
Enter your option:1

Enter Element for Insert Linked List :
300

Options
1 : Insert into Linked List
2 : Delete from Linked List
3 : Display Linked List
4 : Count Linked List
Others : Exit()
Enter your option:1

Enter Element for Insert Linked List :
400
```

```
Options
1 : Insert into Linked List
2 : Delete from Linked List
3 : Display Linked List
4 : Count Linked List
Others : Exit()
Enter your option:1

Enter Element for Insert Linked List :
500

Options
1 : Insert into Linked List
2 : Delete from Linked List
3 : Display Linked List
4 : Count Linked List
Others : Exit()
Enter your option:3

Display Linked List :
# 100 # # 200 # # 300 # # 400 # # 500 #
No Of Items In Linked List : 5

Options
1 : Insert into Linked List
2 : Delete from Linked List
3 : Display Linked List
4 : Count Linked List
Others : Exit()
Enter your option:4

No Of Items In Linked List : 5

Options
1 : Insert into Linked List
2 : Delete from Linked List
3 : Display Linked List
4 : Count Linked List
Others : Exit()
Enter your option:2

No Of Items In Linked List : 5
```

Conclusion:_____

_____

_____

Submitted By :
Name:
Sign:
Roll No :

Checked By :
Name: Ms. Pratiksha S. Patil.
Sign: