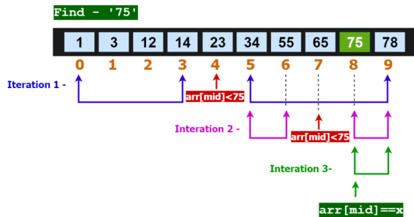



## Searching techniques

### 1.Binary search

Binary search algorithm falls under the category of interval search algorithms. This algorithm is much more efficient compared to linear search algorithm. Binary search only works on sorted data structures. This algorithm repeatedly target the center of the sorted data structure & divide the search space into half till the match is found.

The time complexity of binary search algorithm is O(Log n).

Working –

1. Search the sorted array by repeatedly dividing the search interval in half

2. Begin with an interval covering the whole array.

3. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half.

4. Otherwise narrow it to the upper half.

5. Repeatedly check until the value is found or the interval is empty.

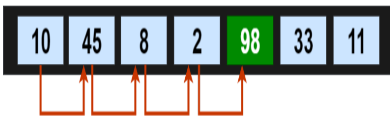
Program-

```
#include < iostream >

#include<conio.h>

int binarySearch(int arr[], int left, int right, int x) {

 while (left <= right) {

  int mid = left + (right - left) / 2;

  if (arr[mid] == x) {

   return mid;

  } else if (arr[mid] < x) {

   left = mid + 1;

  } else {

   right = mid - 1;

  }

 }

 return -1;

}

int main() {

 int myarr[10];

 int num;

 int output;
```

```
 cout << "Please enter 10 elements ASCENDING order" << endl;

 for (int i = 0; i < 10; i++) {

  cin >> myarr[i];

 }

 cout << "Please enter an element to search" << endl;

 cin >> num;

 output = binarySearch(myarr, 0, 9, num);

 if (output == -1) {

  cout << "No Match Found" << endl;

 } else {

  cout << "Match found at position: " << output << endl;

 }

getch();
```

**2.Linear search-**

**In** linear search algorithm 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.




As you can see in the diagram , we have an integer array data structure with some values. We want to search for the value (98) which is at 5th position in this array. Since we are performing the linear search algorithm we start from the beginning of the array and check for matching values till we find a match.

Program-

```
#include < iostream >

#include<conio.h>

void linearSearch(int a[], int n) {

 int temp = -1;

 for (int i = 0; i < 5; i++) {

  if (a[i] == n) {

   cout << "Element found at position: " << i + 1 << endl;

   temp = 0;

   break;

  }

 }
```

```
  if (temp == -1) {
    cout << "No Element Found" << endl;
  }
}
int main() {
  int arr[5];
  cout << "Please enter 5 elements of the Array" << endl;
  for (int i = 0; i < 5; i++) {
    cin >> arr[i];
  }
  cout << "Please enter an element to search" << endl;
  int num;
  cin >> num;
  linearSearch(arr, num);
  getch();
}
```

**Stack**

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

**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**

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

push – Adds or pushes an element into the stack.

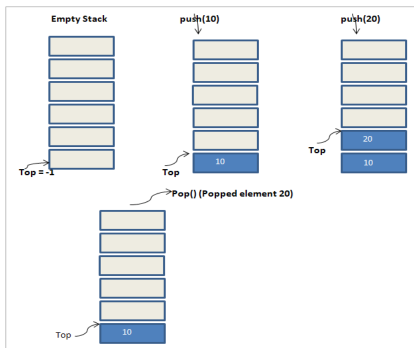
pop – Removes or pops an element out of the stack.

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.

Illustration-




Program-

```
# include<iostream>

#include<conio.h>

class Stack

{

   int top;

public:

int a[10];  //Maximum size of Stack

Stack()

{

   top = -1;

}

// declaring all the function

   void push(int x);

   int pop();

   void isEmpty();

};

// function to insert data into stack

void Stack::push(int x)

{

   if(top >= 10)

   {

      cout << "Stack Overflow \n";

   }

   else

   {
```

```
        a[++top] = x;

        cout << "Element Inserted \n";

    }

}

// function to remove data from the top of the stack

int Stack::pop()

{

    if(top < 0)

    {

        cout << "Stack Underflow \n";

        return 0;

    }

    else

    {

        int d = a[top--];

        return d;

    }

}

// function to check if stack is empty

void Stack::isEmpty()

{

    if(top < 0)

    {

        cout << "Stack is empty \n";

    }

    else

    {

        cout << "Stack is not empty \n";

    }

}

// main function
```

```
int main()

{

   Stack s1;

   s1.push(10);

   s1.push(100);

}
```

**Implementation using array**-

A program that implements a stack using array is given as follows.

Example

```
#include <iostream>

#include<conio.h>

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

## Application of stack

The Stack is Last In First Out (LIFO) data structure. This data structure has some important applications in different aspect. These are like below –

**1.Expression Handling –**

Infix to Postfix or Infix to Prefix Conversion –

The stack can be used to convert some infix expression into its postfix equivalent, or prefix equivalent. These postfix or prefix notations are used in computers to express some expressions. These expressions are not so much familiar to the infix expression, but they have some great advantages also. We do not need to maintain operator ordering, and parenthesis.

Postfix or Prefix Evaluation –

After converting into prefix or postfix notations, we have to evaluate the expression to get the result. For that purpose, also we need the help of stack data structure.

**2.Backtracking Procedure –**

Backtracking is one of the algorithm designing technique. For that purpose, we dive into some way, if that way is not efficient, we come back to the previous state and go into some other paths. To get back from current state, we need to store the previous state. For that purpose, we need stack. Some examples of backtracking is finding the solution for Knight Tour problem or N-Queen Problem etc.

3.Another great use of stack is during the function call and return process. When we call a function from one other function, that function call statement may not be the first statement. After calling the function, we also have to come back from the function area to the place, where we have left our control. So we want to resume our task, not restart. For that reason, we store the address of the program counter into the stack, then go to the function body to execute it. After completion of the execution, it pops out the address from stack and assign it into the program counter to resume the task again.

## Recursion

1."Recursion" is technique of solving any problem by calling same function again and again until some breaking (base) condition where recursion stops and it starts calculating the solution from there on. For eg. calculating factorial of a given number

2.Thus in recursion last function called needs to be completed first.

3.Now Stack is a LIFO data structure i.e. ( Last In First Out) and hence it is used to implement recursion.

4.The High level Programming languages, such as Pascal , C etc. that provides support for recursion use stack for book keeping.

5.In each recursive call, there is need to save the

a.current values of parameters,

b.local variables and

c.the return address (the address where the control has to return from the call).

6.Also, as a function calls to another function, first its arguments, then the return address and finally space for local variables is pushed onto the stack.

7.Recursion is extremely useful and extensively used because many problems are elegantly specified or solved in a recursive way.

## Mathematical Expression

Evaluate an expression represented by a String. Expression can contain parentheses, you can assume parentheses are well-matched. For simplicity, you can assume only binary operations allowed are +, -, *, and /. Arithmetic Expressions can be written in one of three forms:

1.**Infix Notation**: Operators are written between the operands they operate on, e.g. 3 + 4 .

2.**Prefix Notation**: Operators are written before the operands, e.g + 3 4

3.**Postfix Notation**: Operators are written after operands.

Infix Expressions are harder for Computers to evaluate because of the addional work needed to decide precedence. Infix notation is how expressions are written and recognized by humans and, generally, input to programs. Given that they are harder to evaluate, they are generally converted to one of the two remaining forms

## Conversion of Infix To Postfix Expression Using Stack

To convert Infix to Postfix, The variables are passed directly to Postfix and operands are passed to Stack.If ) is encounterd, Stack is popped and element is passed to Postix Expression

Code:

```
#include<iostream>

#include<stack>

using namespace std;

void Postfix(char *a)

{   stack<char> s;

    char output[50],t;

    for(int i=0;a[i]!='\0';i++)

    {   char ch = a[i];

        switch(ch)

      {

      case '^':

      case '-':

      case '+':

      case '/':

      case '*':   s.push(ch);

                 break;

      case ')': t=s.top();

                 s.pop();
```

```
                cout<<t;
                 break;
        }
        if (isalpha(ch))
           cout<<ch;
    }
}
int main()
{
    char a[] = "(((a*b)+(c/d))-e)";
    Postfix(a);
    return 0;
}
```

Output:

ab*cd/+e-

## Conversion of  Infix To Prefix Expression Using Stack

To solve expressions by the computer, we can either convert it in postfix form or to the prefix form. Here we will see how infix expressions are converted to prefix form.

At first infix expression is reversed. Note that for reversing the opening and closing parenthesis will also be reversed.

for an example: The expression: A + B * (C - D)

after reversing the expression will be: ) D – C ( * B + A

so we need to convert opening parenthesis to closing parenthesis and vice versa.

After reversing, the expression is converted to postfix form by using infix to postfix algorithm. After that again the postfix expression is reversed to get the prefix expression.

```
#include<iostream>
#include<string>
using namespace std;
class stack
{
    string item;
    int top;
    public:
```

```
stack()
{
    top=-1;
}
void push(char ch)
{
    top++;
    item[top]=ch;
}
char pop()
{
    char ele;
    if(isempty())
    {
        printf("\n stack overflow!!");
        return '@';
    }
    ele=item[top];
    top--;
    return ele;
}
char peep()
{
    return(item[top]);
}
int isempty()
{
    if(top==-1)
    return 1;
    return 0;
}
```

```
        int priority(char ch)
        {
            if(ch==')')
                return 1;
            else if(ch=='+' || ch=='-')
                return 2;
            else if(ch=='*' || ch=='/')
                return 3;
            else
                return 4;
        }
};
int main()
{
    string infix,prefix;
    char ch,ch2,token,temp;
    int i,j=0,len;
    stack st;
    cout<<"Enter valid infix expression: ";
    cin>>infix;
    len=infix.size();
    for(i=len-1;i>=0;i--)
    {
        token=infix[i];
        if(token==')')
            st.push(token);
        else if(token=='(')
        {
            ch=st.pop();
            while(ch!=')')
            {
```

```
            prefix.push_back(ch);
            ch=st.pop();
          }
        }
        else if(token=='+' || token=='-' || token=='*' || token=='/' || token=='^')
        {
          ch2=st.peep();
          while(!st.isempty() && st.priority(token)<st.priority(ch2))
          {
            ch=st.pop();
            prefix.push_back(ch);
            ch2=st.peep();
          }
          st.push(token);
        }
        else
          st.push(token);
    }
    while(!st.isempty())
    {
        ch=st.pop();
        prefix.push_back(ch);
    }
    cout<<"\n The Corresponding Prefix Expression: ";
      for(j = prefix.size() - 1;j>=0;j--)
      {
          cout<<prefix[j];
    }
     return 0;
}
```

OUTPUT:

Enter valid infix expression: (a+(b*c)/(d-e))

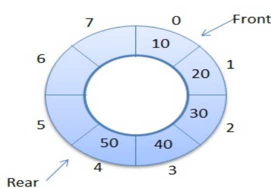
The corresponding Prefix Expression: +a/*bc-de

## Queue

A Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO). A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first. The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory.

### operations




enqueue() − add (store) an item to the queue.

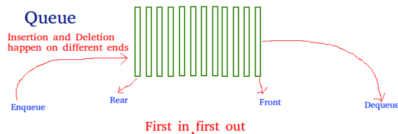
dequeue() − remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are −

peek() − Gets the element at the front of the queue without removing it.

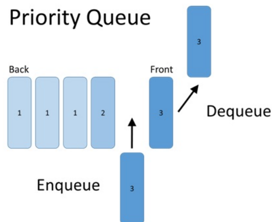
isfull() − Checks if the queue is full.

isempty() − Checks if the queue is empty.




## Types of Queues in Data Structure

Queue is an important structure for storing and retrieving data and hence is used extensively among all the data structures. Queue, just like any queue (queues for bus or tickets etc.) follows a FIFO mechanism for data retrieval which means the data which gets into the queue first will be the first one to be taken out from it, the second one would be the second to be retrieved and so on.




There are three types of Queues in Data Structure

## Simple Queue/linear queue

As is clear from the name itself, simple queue lets us perform the operations simply. i.e., the insertion and deletions are performed likewise. Insertion occurs at the rear (end) of the queue and deletions are performed at the front (beginning) of the queue list.

All nodes are connected to each other in a sequential manner. The pointer of the first node points to the value of the second and so on.

## Circular Queue

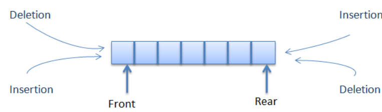
Unlike the simple queues, in a circular queue each node is connected to the next node in sequence but the last node’s pointer is also connected to the first node’s address. Hence, the last node and the first node also gets connected making a circular link overall.

## Priority Queue

Priority queue makes data retrieval possible only through a pre determined priority number assigned to the data items.

While the deletion is performed in accordance to priority number (the data item with highest priority is removed first), insertion is performed only in the order.

**Doubly Ended Queue (Dequeue)**




The doubly ended queue or dequeue allows the insert and delete operations from both ends (front and rear) of the queue.

Queues are an important concept of the data structures and understanding their types is very necessary for working appropriately with them.

**Implementation of linear queue using array**

Queue is a non-primitive linear data structure in which insertion and deletion takes place from different ends, Rear and Front respectively. It is also known as FIFO ( First In First Out ) Data Structure.

Program Code:

```
#include <iostream>

using namespace std;

const int Max=10;

void Qinsert(int Q[],int &R,int F)

{

   if ((R+1)%Max!=F)

   {

      R=(R+1)%Max;

      cout<<"Data:";

      cin>>Q[R];

   }

   else

      cout<<"Queue is Full!"<<endl;

}

void Qdelete(int Q[],int R,int &F)

{

   if (R!=F)

   {

      F=(F+1)%Max;
```

```
        cout<<Q[F]<<" Deleted!";
    }
    else
        cout<<"Queue is empty"<<endl;
}
void Qdisplay(int Q[],int R,int F)
{
    int Cn=F;
    while (Cn!=R)
    {
        Cn=(Cn+1)%Max;
        cout<<Q[Cn]<<endl;
    }
}
int main()
{   //Initialisation Steps
    int Que[Max],Rear=0,Front=0;
    char Ch;
    do
    {
        cout<<"\n I:Insert/D:Delete/S:Show/Q:Quit ";
        cin>>Ch;
        switch(Ch)
        {
            case 'I':Qinsert(Que,Rear,Front);
                    break;
            case 'D':Qdelete(Que,Rear,Front);
                    break;
            case 'S':Qdisplay(Que,Rear,Front);
                    break;
        }
```

```
   }while (Ch!='Q');
   return 0 ;
}
```

**Implementation of circular queue using array**

A circular queue is a type of queue in which the last position is connected to the first position to make a circle.

A program to implement circular queue in C++ is given as follows –

Example

```
#include <iostream>
int cqueue[5];
int front = -1, rear = -1, n=5;
void insertCQ(int val) {
  if ((front == 0 && rear == n-1) || (front == rear+1)) {
    cout<<"Queue Overflow \n";
    return;
  }
  if (front == -1) {
    front = 0;
    rear = 0;
  } else {
    if (rear == n - 1)
    rear = 0;
    else
    rear = rear + 1;
  }
  cqueue[rear] = val ;
}
void deleteCQ() {
  if (front == -1) {
    cout<<"Queue Underflow\n";
    return ;
  }
```

```
    cout<<"Element deleted from queue is : "<<cqueue[front]<<endl;
    if (front == rear) {
      front = -1;
      rear = -1;
    } else {
      if (front == n - 1)
      front = 0;
      else
      front = front + 1;
    }
  }
  void displayCQ() {
    int f = front, r = rear;
    if (front == -1) {
      cout<<"Queue is empty"<<endl;
      return;
    }
    cout<<"Queue elements are :\n";
    if (f <= r) {
      while (f <= r){
        cout<<cqueue[f]<<" ";
        f++;
      }
    } else {
      while (f <= n - 1) {
        cout<<cqueue[f]<<" ";
        f++;
      }
      f = 0;
      while (f <= r) {
        cout<<cqueue[f]<<" ";
```

```
      f++;
    }
  }
  cout<<endl;
}
int main() {
  int ch, val;
  cout<<"1)Insert\n";
  cout<<"2)Delete\n";
  cout<<"3)Display\n";
  cout<<"4)Exit\n";
do {
  cout<<"Enter choice : "<<endl;
  cin>>ch;
  switch(ch) {
    case 1:
      cout<<"Input for insertion: "<<endl;
      cin>>val;
      insertCQ(val);
      break;
    case 2:
      deleteCQ();
      break;
    case 3:
      displayCQ();
      break;
    case 4:
      cout<<"Exit\n";
      break;
      default: cout<<"Incorrect!\n";
  }
```

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

The output of the above program is as follows −

```
1)Insert
2)Delete
3)Display
4)Exit
Enter choice : 1
Input for insertion:
Enter choice : 1
Input for insertion:
Enter choice : 1
Input for insertion:
Enter choice : 1
Input for insertion:
Enter choice : 1
Input for insertion:
Enter choice : 2
Element deleted from queue is : 5
Enter choice : 2
Element deleted from queue is : 3
Enter choice : 2
Element deleted from queue is : 2
Enter choice : 1
Input for insertion: 6
Enter choice : 3
Queue elements are :7 9 6
Enter choice : 4
Exit
```

**Application of queue**

Queue, as the name suggests is used whenever we need to manage any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios:

1.Serving requests on a single shared resource, like a printer, CPU task scheduling etc.

2.In real life scenario, Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free.

3.Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e First come first served.