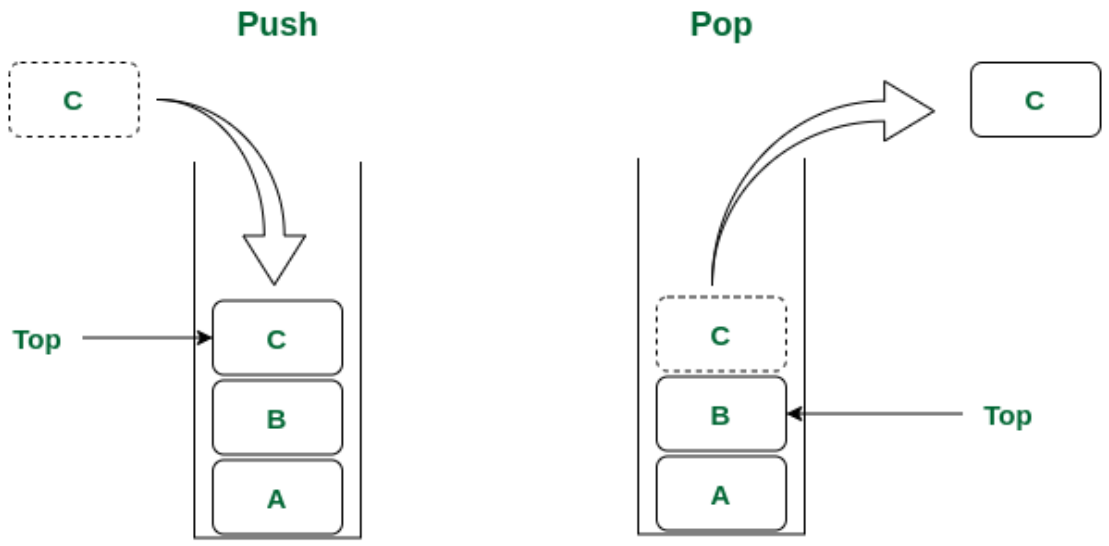
Stack in C++ STL

Stack is a linear data structure that follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out). LIFO implies that the element that is inserted last, comes out first and FILO implies that the element that is inserted first, comes out last.



Stack Data Structure

Some of its main operations are: **push(), pop(), top(), isEmpty(), size(), etc.** In order to make manipulations in a stack, there are certain operations provided to us. When we want to insert an element into the stack the operation is known as the push operation whereas when we want to remove an element from the stack the operation is known as the pop operation. If we try to pop from an empty stack then it is known as underflow and if we try to push an element in a stack that is already full, then it is known as overflow.

Primary Stack Operations:

- void push(int data): When this operation is performed, an element is inserted into the stack.
- int pop(): When this operation is performed, an element is removed from the top of the stack and is returned.

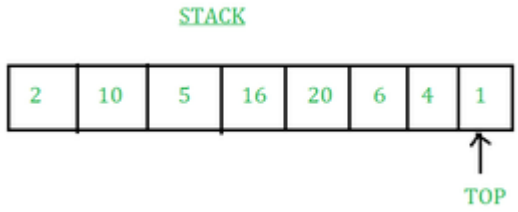
Auxiliary Stack Operations:

- **int top():** This operation will return the last inserted element that is at the top without removing it.
- **int size():** This operation will return the size of the stack i.e. the total number of elements present in the stack.
- **int isEmpty():** This operation indicates whether the stack is empty or not.
- **int isFull():** This operation indicates whether the stack is full or not.

- Register Stack: This type of stack is also a memory element present in the memory unit and can handle a small amount of data only. The height of the register stack is always limited as the size of the register stack is very small compared to the memory.
- Memory Stack: This type of stack can handle a large amount of memory data. The height of the memory stack is flexible as it occupies a large amount of memory data.

What is meant by Top of the Stack?

- When a new element is added to the stack, it is placed on top of the existing elements. Similarly, when an element is removed from the stack, the topmost element is removed first. The top of the stack is always the element that is currently accessible for viewing or manipulation.
- The pointer through which the elements are accessed, inserted, and deleted in the stack is called the **top of the stack**. It is the pointer to the topmost element of the stack.



Application of Stack Data Structure:

- **Function calls and recursion:** When a function is called, the current state of the program is pushed onto the stack. When the function returns, the state is popped from the stack to resume the previous function's execution.
- **Undo/Redo operations:** The undo-redo feature in various applications uses stacks to keep track of the previous actions. Each time an action is performed, it is pushed onto the stack. To undo the action, the top element of the stack is popped, and the reverse operation is performed.
- **Expression evaluation:** Stack data structure is used to evaluate expressions in infix, postfix, and prefix notations. Operators and operands are pushed onto the stack, and operations are performed based on the stack's top elements.
- **Browser history:** Web browsers use stacks to keep track of the web pages you visit. Each time you visit a new page, the URL is pushed onto the stack, and when you hit the back button, the previous URL is popped from the stack.
- **Balanced Parentheses:** Stack data structure is used to check if parentheses are balanced or not. An opening parenthesis is pushed onto the stack, and a closing parenthesis is popped from the stack. If the stack is empty at the end of the expression, the parentheses are balanced.
- **Backtracking Algorithms:** The backtracking algorithm uses stacks to keep track of the states of the problem-solving process. The current state is pushed onto the stack, and when the algorithm backtracks, the previous state is popped from the stack.

Application of Stack in real life:

- CD/DVD stand.
- Stack of books in a book shop.
- Call center systems.
- Undo and Redo mechanism in text editors.
- The history of a web browser is stored in the form of a stack.
- Call logs, E-mails, and Google photos in any gallery are also stored in form of a stack.
- YouTube downloads and Notifications are also shown in LIFO format(the latest appears first).
- Allocation of memory by an operating system while executing a process.

Advantages of Stack:

- **Easy implementation:** Stack data structure is easy to implement using arrays or linked lists, and its operations are simple to understand and implement.
- **Efficient memory utilization**: Stack uses a contiguous block of memory, making it more efficient in memory utilization as compared to other data structures.
- **Fast access time:** Stack data structure provides fast access time for adding and removing elements as the elements are added and removed from the top of the stack.
- **Helps in function calls:** Stack data structure is used to store function calls and their states, which helps in the efficient implementation of recursive function calls.
- **Supports backtracking:** Stack data structure supports backtracking algorithms, which are used in problem-solving to explore all possible solutions by storing the previous states.
- **Used in Compiler Design:** Stack data structure is used in compiler design for parsing and syntax analysis of programming languages.
- **Enables undo/redo operations**: Stack data structure is used to enable undo and redo operations in various applications like text editors, graphic design tools, and software development environments.

Disadvantages of Stack:

- **Limited capacity:** Stack data structure has a limited capacity as it can only hold a fixed number of elements. If the stack becomes full, adding new elements may result in stack overflow, leading to the loss of data.
- **No random access:** Stack data structure does not allow for random access to its elements, and it only allows for adding and removing elements from the top of the stack. To access an element in the middle of the stack, all the elements above it must be removed.
- **Memory management:** Stack data structure uses a contiguous block of memory, which can result in memory fragmentation if elements are added and removed frequently.
- **Not suitable for certain applications:** Stack data structure is not suitable for applications that require accessing elements in the middle of the stack, like searching or sorting algorithms.

- **Stack overflow and underflow**: Stack data structure can result in stack overflow if too many elements are pushed onto the stack, and it can result in stack underflow if too many elements are popped from the stack.
- **Recursive function calls limitations:** While stack data structure supports recursive function calls, too many recursive function calls can lead to stack overflow, resulting in the termination of the program

Stack implementation using linked List

```
// C++ program to Implement a stack

// using singly linked list

#include <bits/stdc++.h>

using namespace std;


// creating a linked list;

class Node {

public:

    int data;

    Node* link;


    // Constructor

    Node(int n)

    {

        this->data = n;

        this->link = NULL;

    }

};


class Stack {

    Node* top;


public:

    Stack() { top = NULL; }
```

```
void push(int data)
{

    // Create new node temp and allocate memory in heap
    Node* temp = new Node(data);

    // Check if stack (heap) is full.
    // Then inserting an element would
    // lead to stack overflow
    if (!temp) {
        cout << "\nStack Overflow";
        exit(1);
    }

    // Initialize data into temp data field
    temp->data = data;

    // Put top pointer reference into temp link
    temp->link = top;

    // Make temp as top of Stack
    top = temp;
}

// Utility function to check if
// the stack is empty or not
bool isEmpty()
{
    // If top is NULL it means that
```

```
    // there are no elements are in stack
    return top == NULL;
}

// Utility function to return top element in a stack
int peek()
{
    // If stack is not empty , return the top element
    if (!isEmpty())
        return top->data;
    else
        exit(1);
}

// Function to remove
// a key from given queue q
void pop()
{
    Node* temp;

    // Check for stack underflow
    if (top == NULL) {
        cout << "\nStack Underflow" << endl;
        exit(1);
    }
    else {

        // Assign top to temp
        temp = top;
```

```
            // Assign second node to top
            top = top->link;

            // This will automatically destroy
            // the link between first node and second node

            // Release memory of top node
            // i.e delete the node
            free(temp);
        }
    }

    // Function to print all the
    // elements of the stack
    void display()
    {
        Node* temp;

        // Check for stack underflow
        if (top == NULL) {
            cout << "\nStack Underflow";
            exit(1);
        }
        else {
            temp = top;
            while (temp != NULL) {

                // Print node data
                cout << temp->data;
```

```
                // Assign temp link to temp
                temp = temp->link;
                if (temp != NULL)
                    cout << " -> ";
            }
        }
    }
};

// Driven Program
int main()
{
    // Creating a stack
    Stack s;

    // Push the elements of stack
    s.push(11);
    s.push(22);
    s.push(33);
    s.push(44);

    // Display stack elements
    s.display();

    // Print top element of stack
    cout << "\nTop element is " << s.peek() << endl;

    // Delete top elements of stack
    s.pop();
    s.pop();
```

```
    // Display stack elements
    s.display();

    // Print top element of stack
    cout << "\nTop element is " << s.peek() << endl;

    return 0;
}
```

```
44 -> 33 -> 22 -> 11
Top element is 44
22 -> 11
Top element is 22
```


Output

```
44 -> 33 -> 22 -> 11
```

```
Top element is 44

22 -> 11

Top element is 22
```

Time Complexity: O(1), for all push(), pop(), and peek(), as we are not performing any kind of traversal over the list. We perform all the operations through the current pointer only.
Auxiliary Space: O(N), where N is the size of the stack
In this implementation, we define a Node class that represents a node in the linked list, and a Stack class that uses this node class to implement the stack. The head attribute of the Stack class points to the top of the stack (i.e., the first node in the linked list).

To push an item onto the stack, we create a new node with the given item and set its next pointer to the current head of the stack. We then set the head of the stack to the new node, effectively making it the new top of the stack.

To pop an item from the stack, we simply remove the first node from the linked list by setting the head of the stack to the next node in the list (i.e., the node pointed to by the next pointer of the current head). We return the data stored in the original head node, which is the item that was removed from the top of the stack.

Benefits of implementing a stack using a singly linked list include:

Dynamic memory allocation: The size of the stack can be increased or decreased dynamically by adding or removing nodes from the linked list, without the need to allocate a fixed amount of memory for the stack upfront.
Efficient memory usage: Since nodes in a singly linked list only have a next pointer and not a prev pointer, they use less memory than nodes in a doubly linked list.
Easy implementation: Implementing a stack using a singly linked list is straightforward and can be done using just a few lines of code.
Versatile: Singly linked lists can be used to implement other data structures such as queues, linked lists, and trees.
In summary, implementing a stack using a singly linked list is a simple and efficient way to create a dynamic stack data structure in Python.

Real time examples of stack:

Stacks are used in various real-world scenarios where a last-in, first-out (LIFO) data structure is required. Here are some examples of real-time applications of stacks:

Function call stack: When a function is called in a program, the return address and all the function parameters are pushed onto the function call stack. The stack

allows the function to execute and return to the caller function in the reverse order in which they were called.
Undo/Redo operations: In many applications, such as text editors, image editors, or web browsers, the undo and redo functionalities are implemented using a stack. Every time an action is performed, it is pushed onto the stack. When the user wants to undo the last action, the top element of the stack is popped and the action is reversed.
Browser history: Web browsers use stacks to keep track of the pages visited by the user. Every time a new page is visited, its URL is pushed onto the stack. When the user clicks the "Back" button, the last visited URL is popped from the stack and the user is directed to the previous page.
Expression evaluation: Stacks are used in compilers and interpreters to evaluate expressions. When an expression is parsed, it is converted into postfix notation and pushed onto a stack. The postfix expression is then evaluated using the stack.
Call stack in recursion: When a recursive function is called, its call is pushed onto the stack. The function executes and calls itself, and each subsequent call is pushed onto the stack. When the recursion ends, the stack is popped, and the program returns to the previous function call.
In summary, stacks are widely used in many applications where LIFO functionality is required, such as function calls, undo/redo operations, browser history, expression evaluation, and recursive function calls.

Stack implementation using linked list

```
// C++ program to Implement a stack
// using singly linked list
#include <bits/stdc++.h>
using namespace std;

// creating a linked list;
class Node {
public:
    int data;
    Node* link;

    // Constructor
    Node(int n)
    {
        this->data = n;
        this->link = NULL;
    }
};

class Stack {
    Node* top;
```

```
public:
    Stack() { top = NULL; }

    void push(int data)
    {

        // Create new node temp and allocate memory in heap
        Node* temp = new Node(data);

        // Check if stack (heap) is full.
        // Then inserting an element would
        // lead to stack overflow
        if (!temp) {
            cout << "\nStack Overflow";
            exit(1);
        }

        // Initialize data into temp data field
        temp->data = data;

        // Put top pointer reference into temp link
        temp->link = top;

        // Make temp as top of Stack
        top = temp;
    }

    // Utility function to check if
    // the stack is empty or not
    bool isEmpty()
    {
        // If top is NULL it means that
        // there are no elements are in stack
        return top == NULL;
    }

    // Utility function to return top element in a stack
    int peek()
    {
        // If stack is not empty , return the top element
        if (!isEmpty())
            return top->data;
        else
            exit(1);
    }

    // Function to remove
    // a key from given queue q
```

```
void pop()
{
    Node* temp;

    // Check for stack underflow
    if (top == NULL) {
        cout << "\nStack Underflow" << endl;
        exit(1);
    }
    else {

        // Assign top to temp
        temp = top;

        // Assign second node to top
        top = top->link;

        // This will automatically destroy
        // the link between first node and second node

        // Release memory of top node
        // i.e delete the node
        free(temp);
    }
}

// Function to print all the
// elements of the stack
void display()
{
    Node* temp;

    // Check for stack underflow
    if (top == NULL) {
        cout << "\nStack Underflow";
        exit(1);
    }
    else {
        temp = top;
        while (temp != NULL) {

            // Print node data
            cout << temp->data;

            // Assign temp link to temp
            temp = temp->link;
            if (temp != NULL)
                cout << " -> ";
```

```
            }
        }
    }
};

// Driven Program
int main()
{
    // Creating a stack
    Stack s;

    // Push the elements of stack
    s.push(11);
    s.push(22);
    s.push(33);
    s.push(44);

    // Display stack elements
    s.display();

    // Print top element of stack
    cout << "\nTop element is " << s.peek() << endl;

    // Delete top elements of stack
    s.pop();
    s.pop();

    // Display stack elements
    s.display();

    // Print top element of stack
    cout << "\nTop element is " << s.peek() << endl;

    return 0;
}
```

Output

```
44 -> 33 -> 22 -> 11
Top element is 44
22 -> 11
Top element is 22
```

check for balanced parenthesis

The idea is to put all the opening brackets in the stack.

Whenever you hit a closing bracket, search if the top of the stack is the opening bracket of the same nature.

If this holds then pop the stack and continue the iteration.

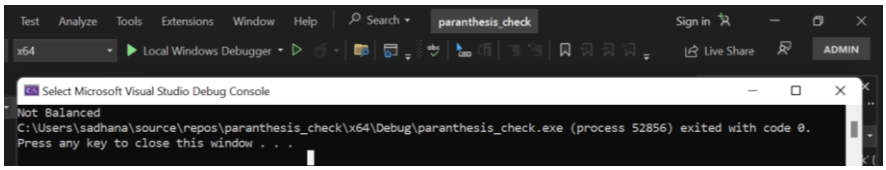
In the end if the stack is empty, it means all brackets are balanced or well-formed.

Otherwise, they are not balanced.

```
// C++ program to check for balanced brackets.

#include <iostream>
#include <stack>
#include <string>
using namespace std;

// Function to check if brackets are balanced
bool BracketsBalanced(string expr)
{
    // Declare a stack to hold the previous brackets.
    stack<char> st;
    for (int i = 0; i < expr.length(); i++) {
        if (st.empty()) {

            // If the stack is empty
            // just push the current bracket
            st.push(expr[i]);
        }
        else if ((st.top() == '(' && expr[i] == ')')
            || (st.top() == '{' && expr[i] == '}')
            || (st.top() == '[' && expr[i] == ']')) {

            // If we found any complete pair of bracket
            // then pop
            st.pop();
        }
        else {
            st.push(expr[i]);
        }
    }
    if (st.empty()) {

        // If stack is empty return true
        return 1;
    }
    return 0;
}

// Driver code
int main()
{
    string expr = "{()}[])";

    // Function call
    if (BracketsBalanced(expr))
        cout << "Balanced";
    else
        cout << "Not Balanced";
    return 0;
```

```
}
```

Output


Test Analyze Tools Extensions Window Help
Search
paranthesis_check
Sign in
x64
Local Windows Debugger
Live Share
ADMIN
Select Microsoft Visual Studio Debug Console
Not Balanced
C:\Users\sadhana\source\repos\paranthesis_check\x64\Debug\paranthesis_check.exe (process 52856) exited with code 0.
Press any key to close this window . . .

QUEUES

* Definition And Applications:-

→ A queue is a linear list in which insertions and deletions take place at different ends. The end at which new elements are added is called the back or rear or tail, and that from which old elements are deleted is called the front or head

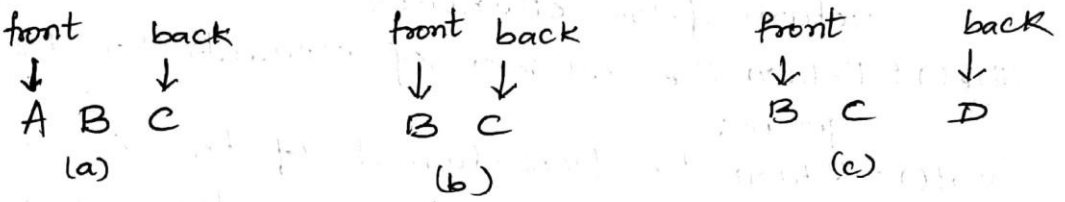


fig 1: Sample queues.

→ A queue with three elements is shown in fig. 1.

→ If the first element in fig 1(a). is deleted, it results in fig 1(b). Similary if you want to insert 'D' after 'C', it results in fig 1(c).

→ Queue is a FIFO list, whereas a stack is a LIFO list.

→ Real world examples for queues are,

1. If there is a queue to draw money infront of ATM machine, the person who enters first will draw the money & comes out first.
2. If you want to buy a movie ticket & you are the first person standing in line formed infront of ticket counter. Then you are the first person to get the ticket & leave the line.

* The Abstract data type :-

→ The ADT queue is given below,

```
AbstractDataType queue
{
  instances
     ordered list of elements; one end is called
     the front; the other is the back;
  operations
    empty() : Return true if the queue is empty,
              return false otherwise;
     size() : Return the number of elements in the
              queue;
    front() : Return the front element of the queue;
     back() : Return the back element of the queue;
      pop() : Remove an element from the front
              of the queue;
    push(x) : Add element 'x' at the back of the
              queue;
}
```

→ Program 1. gives the abstract class that corresponds to above ADT.

```
template<class T>
class queue
{
 public:
   virtual ~queue(){}
   virtual bool empty() const = 0; //return true iff queue
                                     is empty
   virtual int size() const = 0; //return number of
                                   elements in queue
   virtual T& front() = 0; //return reference to the
                             front element
   virtual T& back() = 0; //return reference to the
                            back element
   virtual void pop() = 0; //remove the front element
```

```
virtual void push(const T& theElement) = 0; // add
                theElement at the back of the queue
};
```

* **Array Representation:-**

* **The Representation:-**

→ Suppose that the queue elements are mapped into an array queue using eqn.(1),

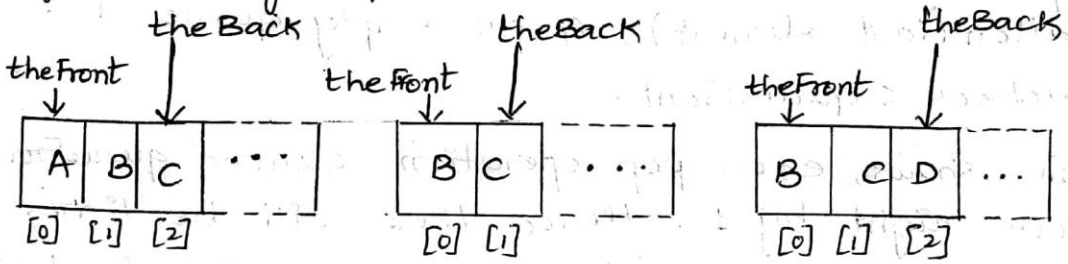
$$location(i) = i \longrightarrow (1)$$

→ Element 'i' of the queue is stored in queue[i], i≥0.

→ Let arrayLength be the length or capacity of the array queue and queueFront & queueBack respectively, be the location of the front & back elements of the queue.

→ When eqn.(1) is used, queueFront = 0 & the queue size is queueBack + 1. An empty queue has queueBack = -1

→ Fig.(2) depicts the representation of queues in Fig(1) using eqn.(1).



Fig(2): Queues of Fig(1) using eqn.(1).

→ To push an element into a queue, the queueBack should be increased by '1' & the new element is placed at queue[queueBack], which means the push operation requires Θ(1) time.

→ To pop an element, we must slide the elements in position 1 through queueBack one position down the

array.

→ sliding the elements takes $\Theta(n)$ time where 'n' is the number of elements in the queue following the pop.

→ If an element is popped in $\Theta(1)$ time using eqn.(2)

i.e. location(i) = location(front element) + i → (2),

the eqn.(2) does not require us to shift the queue one position left each time an element is popped from the queue. Instead we can simply increase location(front element) by 1.

→ Fig(3). depicts the representation of queues of fig(1) when eqn.(2) is used.

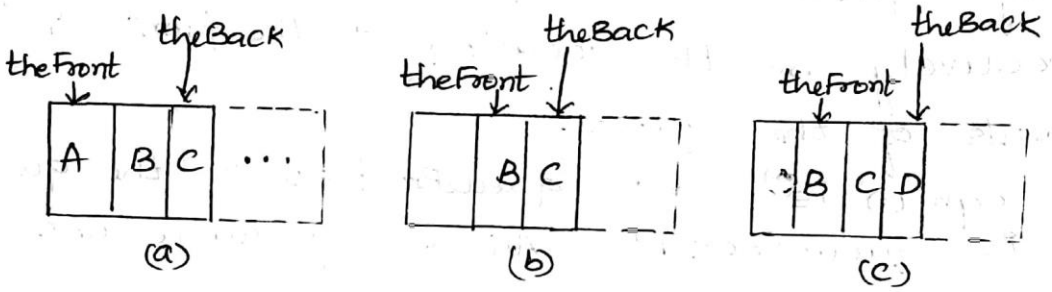


Fig.(3): Queues of fig.(1) using eqn.(2)

→ In fig(3). queueFront = location(front element), queueBack = location(last element), & an empty queue has queueBack < queueFront.

→ Fig(3.b). shows, each pop operation causes queueFront to move right by 1. Hence there will be times when queueBack = arrayLength - 1 and queueFront > 0.

→ At these times the number of elements in the queue is less than arrayLength, and there is space for additional elements at the left end of the array.

→ To insert some more elements to this queue, we have to shift the elements currently present in this queue towards left end of the queue & create space at the right end. as depicted in fig.

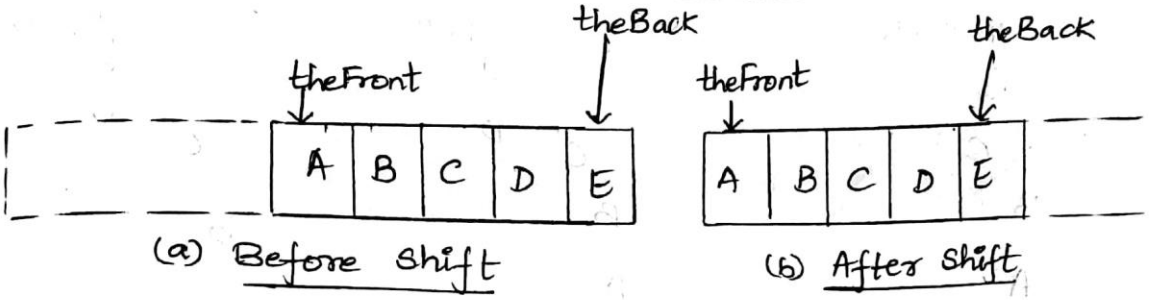


Fig4: Shifting a queue

→ This shifting increases the time for a push operation from $\Theta(1)$ to Θ(arrayLength) when eqn (1) is used.

→ ∴ Improving the efficiency of the pop operation decreases the efficiency of push operation.

→ To make push & pop time complexity $\Theta(1)$, we can allow the queue to wrap around the end of the array. as in fig 5.

→ When the array is viewed as a circle, each array position has a next & a previous position.

→ The position next to position 'arrayLength - 1' will have zero. & the position that precedes '0' index is of index 'arrayLength - 1'.

→ When the back of the queue is at arrayLength - 1, the next element is put into position '0'.

→ The circular array representation of a queue uses the mapping function,

$$\text{location}(i) = (\text{location}(\text{front}) + i) \,\%\, \text{arrayLength} \rightarrow (3).$$

→ In fig 5, theFront is pointing one position behind the first element of queue thereby changing the convention for the variable queueFront & the convention for queueBack is unchanged. This change simplifies the codes.

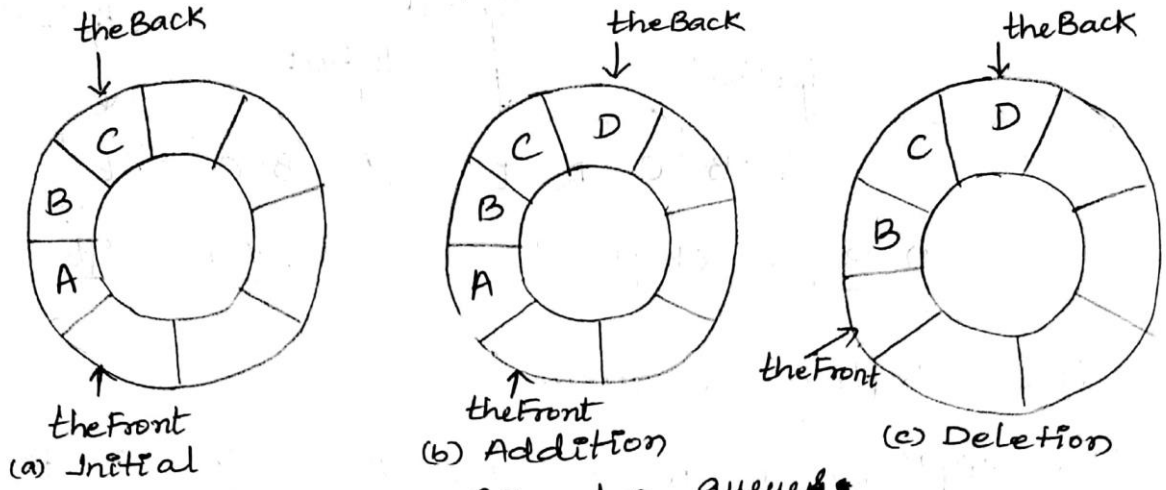


Fig 5: Circular queues

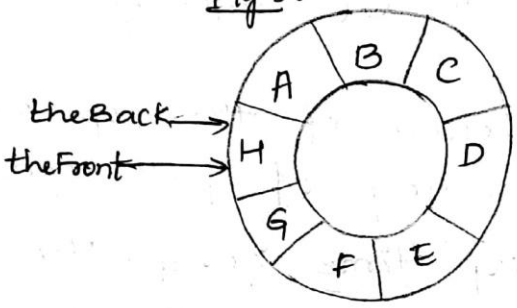


Fig. 6 : A circular queue with arrayLength elements

→ Pushing an element into the queue of fig 5(a) results in the queue of fig. 5(b).

→ Popping an element from queue of fig 5(b) results in the queue of fig 5(c).

→ A queue is empty if queueFront = queueBack.

→ The initial condition queueFront = QueueBack = 0 defines an initially empty queue.

→ If we push elements to fig 5(b) until the number of elements in the array queue equals to arrayLength, it results in fig 6.

→ In fig. 6. queueFront = queueBack which is the same condition for empty queue.

→ ∴ to distinguish between queue empty & queue full, we will not allow the queue to be full.

→ ∴ Before pushing an element into a queue, verify whether this push will cause the queue to get full. If so, double the length of the array queue & then proceed with the push.

★ The class arrayQueue :-

→ The class arrayQueue uses the equation,
location (i) = (location (front) + i) % arrayLength, to map a queue into a 1D array queue.

→ The data members of arrayQueue are queueFront, queueBack & queue.

→ Program 1 is for push. & Program 2. is to double the length of array.

Linked Representation:

* The major problem with the queue implemented using an array is, it will work for an only fixed number of data values.
* Queue using an array is not suitable when we don't know the size of data which we are going to use.
* Queue implemented using a linked list data structure can work for an unlimited number of values. ie it can work for variable data size.
* A queue, like a stack, can be represented as a chain. We need two variables queueFront & queueBack to keep track of the two ends of a queue.
* There a two possibilities for binding these two variables to the two ends of a chain.
  (i) linked from front to back
  (ii) from back to front.
* Relative difficulty of Push & Pop operations determines the direction of linkage.
* Both linkages are well suited for Push operations but the front to back linkage is more efficient for Pops.
* Hence Collectively, front to back linkage is used for both Push & Pop operations.
* We can use the initial values queueFront = queueBack = NULL & the boundary value queueFront = NULL if the queue is empty.

* To implement queue using linked list, we need to set the following things before implementing actual operations.

Step 1: Include all the header files which are used in the program. And declare all the user defined functions.

Step 2: Define a 'Node' structure with two members data and link (next).

Step 3: Define two node pointers 'front' & 'Back' & set both to NULL.

Step 4: Perform suitable user defined operations such as push, pop etc..

* Steps to implement push operation.:

Step 1: Create newNode with given value & set 'newNode → next' to NULL.

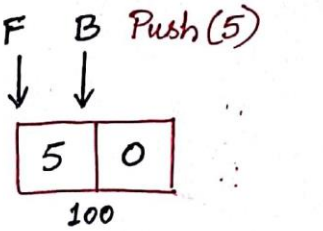
Step 2: Check whether queue is empty.
ie queueSize == 0;

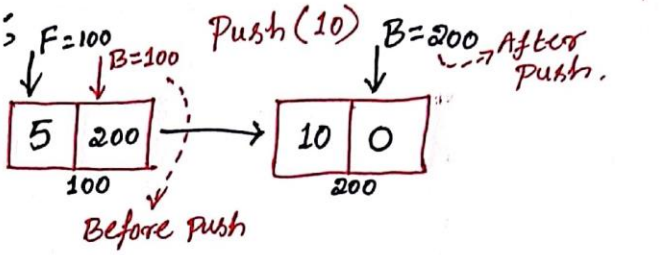
Step 3: If it is empty then, set front = newNode & Back = newNode

Step 4: If it is Not empty then, set Back → next = newNode and Back = newNode.

```
template <class T>
Void linkedQueue<T>::Push(Const T& theElement)
{ // Add the element to back of queue.
// create node for new element
ChainNode<T> * newNode = new ChainNode<T>(theElement, NULL);
```

```
// add new node to back of queue
if (queueSize == 0)
    queueFront = newNode
else
    queueBack -> next = newNode;
    queueBack = newNode;
    queueSize ++;
}
```





* Steps to implement pop operation.:

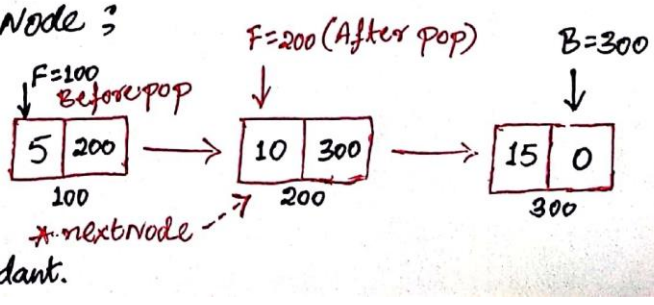
Step1: Check whether queue is empty.

Step2: if it is empty, display "queue is empty" & terminate.

Step3: if it is not empty then, define a Node pointer "nextNode" & set it to front.

Step4: Then set front = front -> nextNode & delete "front".

```
template<class T>
void linkedQueue<T>::pop()
{ // delete front element
 if (queueFront == NULL)
    throw queueEmpty();
ChainNode<T>* nextNode = queueFront -> next;
 delete queueFront;
 queueFront = nextNode;
 queueSize --;
}
```



Applications

(i) Rail Road Car Rearrangement

Problem description and solution strategy

→ the rail road car reaarangement problem holding tracks lie b/w the input and output track as shown in fig below
→ These tracks operated in a FIFO manner and so may be regarded as queue.
→ moving a car from a holding track to the input track or from the output track to a holding track is forbidden
→ All cars motion is in the direction indicated by the arrowheads shown in figure.
→ we reserve track H1 for moving cars directly from the input track to the output track, so only (k-1) tracks are available to hold cars that are not ready to be output.

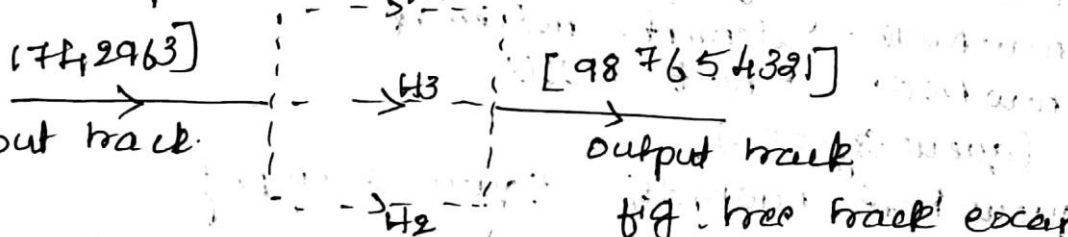


fig: three track example

→ consider rearranging nine cars that have the initial ordering 5, 8, 1, 7, 4, 2, 9, 6, 3.
→ Assume K=3, car3 cannot moved directly to the output track, as car1 and car2 must come before it
→ so car3 is moved to H1, car6 can be placed behind car3 in H1, as car6 is to be output after cars, car4 can now be placed after car6 in H1.
→ car2 cannot be placed after car4 as car2 is to be output before car4. So, it is placed at the front of H2
→ Car1 can now be placed after car2 in H2 and car7 can be placed after it, car1 can be moved to the output using H3. Next car2 is moved from H2 to the output.

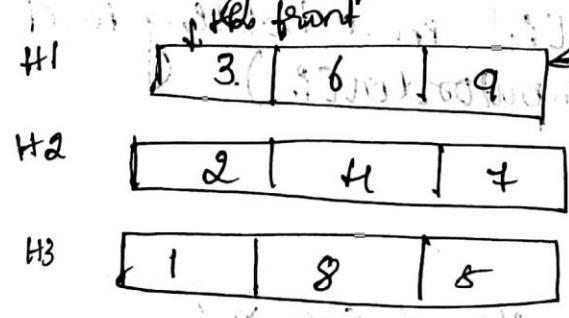
Then car 3 is moved from H1 to the output and car 6 is moved from H2 to the output. Car 5 is to be output next.

→ Car 8 is moved from the input tracks to H2 then car 5 is moved to the output track.

→ Now, cars 6, 7, 8 and 9 are moved from holding tracks to the output track.

Implementation

→ Program ① below gives the code for output from Holding Track and program ② gives the new code for putInHoldingTrack.



Program

```
arrayStack<int> * track; // array of holding tracks
int numberOfCars;
int numberOfTracks;
int smallestCar; // smallest car in any holding track
int theTrack; // holding track with car smallestCar
bool railroad(int inputOrder[], int theNumberOfCars, int
                                        theNumberOfTracks)
{ // Rearrange railroad cars & return true if successful
  numberOfCars = theNumberOfCars;
  numberOfTracks = theNumberOfTracks;
  // create queues for use as holding tracks
  track = new arrayStack<int>[numberOfTracks];
  int nextCarToOutput = 1;
  smallestCar = numberOfCars + 1; // no car in holding track
```

```
// rearrange cars
for (int i=1; i < number of cars; i++)
  { if (inputorder[i] == next car to output)
    { cout << "move car" << inputorder[i] << " from input -
        track to output track" << endl;
      next car to output++;
// output from holding track.
while (smallest car == next car to output)
  { output from holding track();
    next car to output ++;
  }
else // put car inputorder[i] in a holding track
  if (!put in holding track (inputorder[i]))
    return false;
  return true;
}

/* Function to output a railroad car */
void output from holding track()
{ // output the smallest car from the holding track
  track[its track].pop()
  cout << "move car" << smallest car << " from holding track
    its track << " to output track" << endl;
// find new smallest car and its track by checking all
                queue fronts.
  Smallest car = number of cars + 2;
for (int i=1; i <= number of tracks; i++)
if (! track[i].empty() && track[i].front() < smallest car)
    { smallest car = track[i].front();
      its track = i;
    }

/*
```

```
/* Function to put a railroad car into a holding tracks */
bool putInHoldingTrack(int c) //put car c into holding track.
{ int bestTrack=0; // best track so far
  int bestLast=0; // last car in best Track
  // Scan Tracks
  for(int i=1; i<=numberofTracks; i++)
  { if(!tracks[i].empty())
    { int lastCar=track[i].back();
      if(c>lastCar && lastCar>bestLast)
      { //Track i has bigger car at its rear
        bestLast=lastCar;
        bestTrack=i;
      }
    }
    else // track i empty
    { if(bestTrack==0)
      { bestTrack=i;
      }
      if(bestTrack==0)
        return false; // no feasible track
      // add c to best track.
      track[bestTrack].push(c);
      cout<<"move car"<<c<<" from i/p track"
        <<"to holding track "<<bestTrack<<endl;
      // update smallest car and its track.
      if(c<smallestCar)
      { smallestCar=c;
        itsTrack=bestTrack;
      }
      return true;
    }
```

Infix Postfix and Prefix

An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are –

- Infix
- Prefix
- Postfix

Infix notations are normal notations, that are used by us while write different mathematical expressions. The Prefix and Postfix notations are quite different.

- Prefix and Postfix notations are easier to parse for a machine.
- With prefix and postfix notation there is never any question like operator precedence.
- There is no issue of left-right associativity.

Prefix Notation

In this notation, the operator is **prefixed** to operands, i.e., operator is written ahead of operands. For example, **+ab**. This is equivalent to its infix notation **a + b**. Prefix notation is also known as **Polish Notation**.

Postfix Notation

This notation style is known as **Reversed Polish Notation**. In this notation style, the operator is **postfixed** to the operands i.e., the operator is written after the operands. For example, **ab+**. This is equivalent to its infix notation **a + b**.

Precedence

When an operand is in between two different operators, which operator will take the operand first, is decided by the precedence of an operator over others. For example –

□ + □ ∗ □ → □ + (□ ∗ □)

PRECEDENCE ORDER

^

/ *

+ -

decreasing from top to bottom

in same line it is same

Operator associativity.

^ is having right to left associativity, rest operators are having right to left associativity.

Below are the steps to implement the above idea:

1. Scan the infix expression **from left to right**.
2. If the scanned character is an operand, put it in the postfix expression.
3. Otherwise, do the following
    - If the precedence and associativity of the scanned operator are greater than the precedence and associativity of the operator in the stack [or the stack is empty or the stack contains a '**(**'], then push it in the stack. ['**^**' operator is right associative and other operators like '**+**','**–**','*****' and '***/***' are left-associative].
        - Check especially for a condition when the operator at the top of the stack and the scanned operator both are '**^**'. In this condition, the precedence of the scanned operator is higher due to its right associativity. So it will be pushed into the operator stack.
        - In all the other cases when the top of the operator stack is the same as the scanned operator, then pop the operator from the stack because of left associativity due to which the scanned operator has less precedence.
    - Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator.
        - After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
4. If the scanned character is a '**(**', push it to the stack.
5. If the scanned character is a '**)**', pop the stack and output it until a '**(**' is encountered, and discard both the parenthesis.
6. Repeat steps **2-5** until the infix expression is scanned.
7. Once the scanning is over, Pop the stack and add the operators in the postfix expression until it is not empty.
8. Finally, print the postfix expression.

Example:

a+b*(c^d-e)^(f+g*h)-i abcd^e-fgh*+^*+i-

Input Expression	Stack	Postfix Expression
a		a
+	+	a
b	+	ab
*	+*	ab
(	+*(	ab
c	+*(	abc
^	+*(^	abc
d	+*(^	abcd
-	+*(-	abcd^
e	+*(-	abcd^e
)	+*	abcd^e-
^	+*^	abcd^e-
(	+*^(	abcd^e-
f	+*^(	abcd^e-f
+	+*^(+	abcd^e-f
g	+*^(+	abcd^e-fg
*	+*^(+*	abcd^e-fg
h	+*^(+*	abcd^e-fgh
)	+*^	abcd^e-fgh*+
-	-	abcd^e-fgh*+^*+
i	-	abcd^e-fgh*+^*+i
		abcd^e-fgh*+^*+i-

Final postfix expression: abcd^e-fgh*+^*+i-

Write down the C++ code for infix to postfix conversion

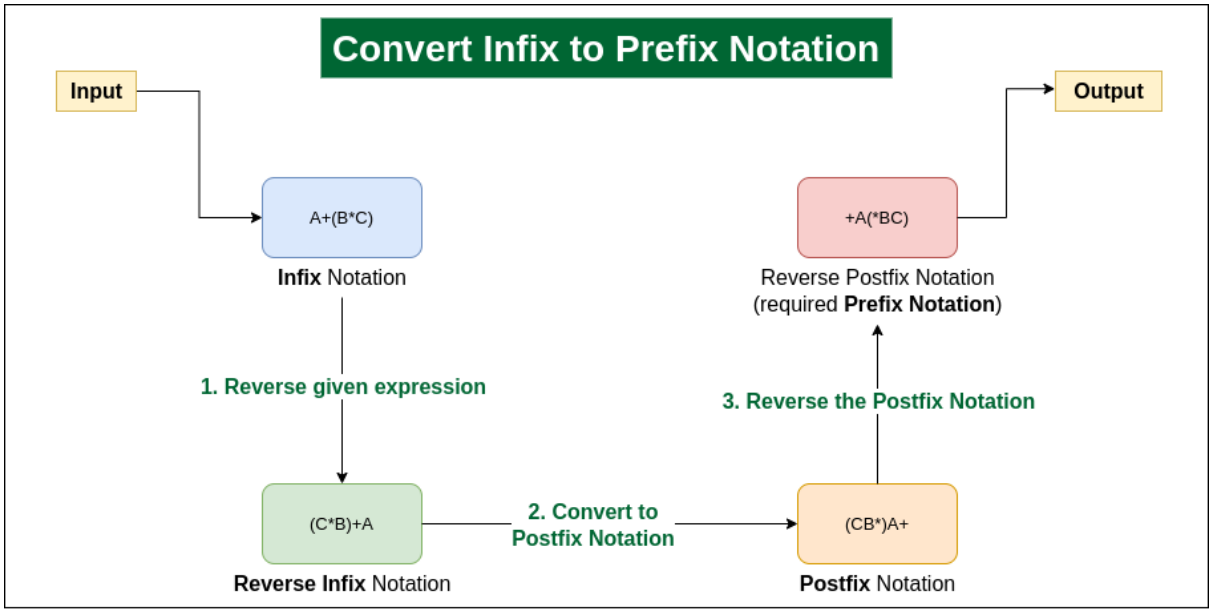
Infix to prefix conversion:

*o convert an infix expression to a prefix expression, we can use the **stack data structure**. The idea is as follows:*

- ***Step 1:** Reverse the infix expression. Note while reversing each '(' will become ')' and each ')' becomes '('.*
- ***Step 2:** Convert the reversed **infix expression to "nearly" postfix expression**.*
    - *While converting to postfix expression, instead of using pop operation to pop operators with greater than or equal precedence, here we will only pop the operators from stack that have greater precedence.*

- ***Step 3:*** *Reverse the postfix expression.*

The stack is used to convert infix expression to postfix form.



Example:

K+L

L+K

LK+

+KL

K+L-M*N+(O^P)*W/U/V*T+Q

Q+T*V/U/W*)P^O(+N*M-L+K

Input Expression	Stack	Prefix Expression
Q		Q
+	+	Q
T	+	QT
*	+*	QT
V	+*	QTV
/	+*/	QTV
U	+*/	QTVU
/	+*//	QTVU
W	+*//	QTVUW
*	+*//*	QTVUW
)	+*//*)	QTVUW
P	+*//*)	QTVUWP
^	+*//*)^	QTVUWP
O	+*//*)^	QTVUWPO
(	+*//*	QTVUWPO^
+	++	QTVUWPO^*//*
N	++	QTVUWPO^*//*N
*	++*	QTVUWPO^*//*N
M	++*	QTVUWPO^*//*NM
-	++-	QTVUWPO^*//*NM*

L	++-	QTVUWPO^*//*NM*L
+	++-+	QTVUWPO^*//*NM*L
K	++-+	QTVUWPO^*//*NM*LK
		QTVUWPO^*//*NM*LK+-++

Final prefix expression will be reverse of the postfix expression formed after reversal of the original expression. So, the final expression will be
++ - +KL*MN*//*^OPWUVTQ
Write down the code for postfix conversion.

Evaluation of Postfix Expression using Stack:

To evaluate a postfix expression we can use a stack.
Iterate the expression from left to right and keep on storing the operands into a stack. Once an operator is received, pop the two topmost elements and evaluate them and push the result in the stack again.

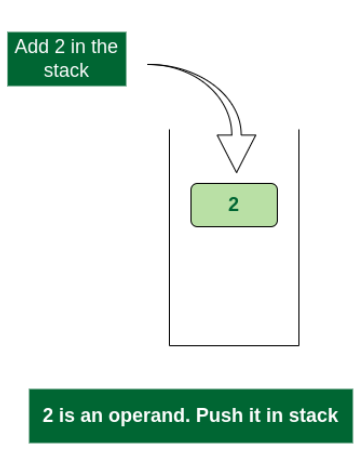
Follow the steps mentioned below to evaluate postfix expression using stack:

- Create a stack to store operands (or values).
- Scan the given expression from left to right and do the following for every scanned element.
    - If the element is a number, push it into the stack.
    - If the element is an operator, pop operands for the operator from the stack. Evaluate the operator and push the result back to the stack.
- When the expression is ended, the number in the stack is the final answer.

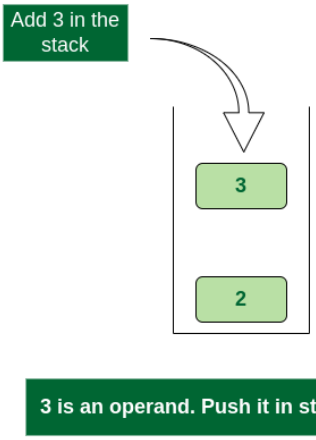
Illustration:
Follow the below illustration for a better understanding:

*Consider the expression: exp = **"2 3 1 * + 9 -"***

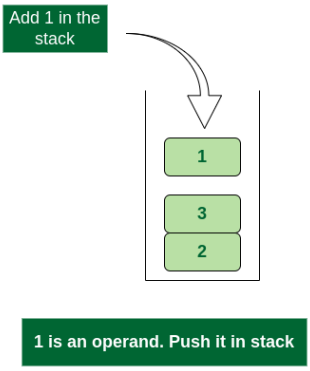
- *Scan **2**, it's a number, So push it into stack. Stack contains '2'.*



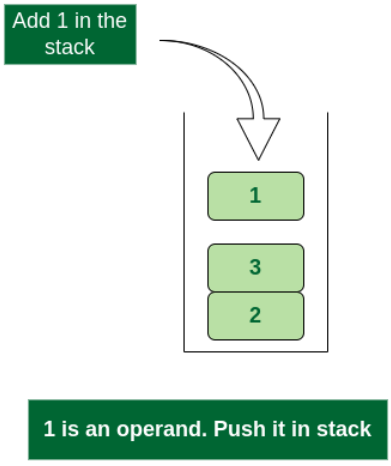
Scan 3, again a number, push it to stack, stack now contains '2 3' (from bottom to top)



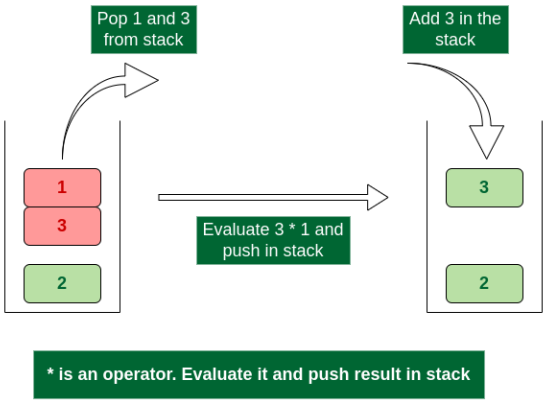
- *Scan 1, again a number, push it to stack, stack now contains '2 3 1'*



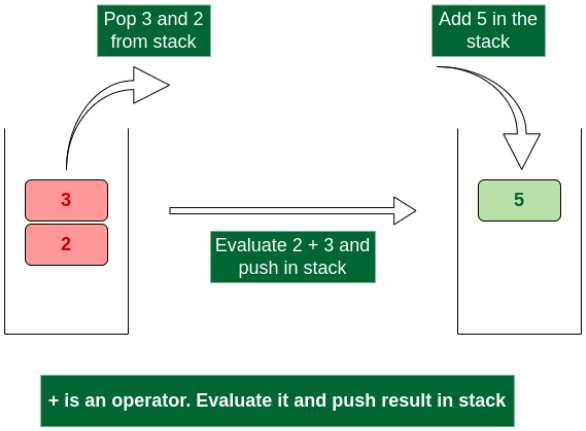
Scan 1, again a number, push it to stack, stack now contains '2 3 1'



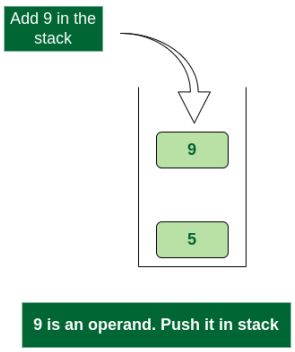
- *Scan *, it's an operator. Pop two operands from stack, apply the * operator on operands. We get 3*1 which results in 3. We push the result 3 to stack. The stack now becomes '2 3'.*



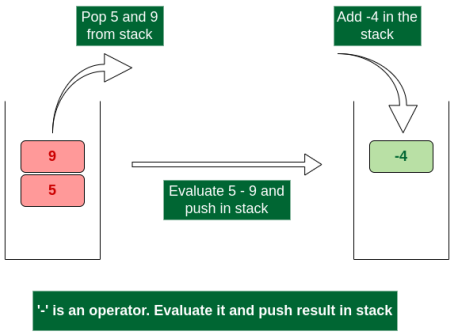
- *Scan +, it's an operator. Pop two operands from stack, apply the + operator on operands. We get 3 + 2 which results in 5. We push the result 5 to stack. The stack now becomes '5'.*



- *Scan 9, it's a number. So we push it to the stack. The stack now becomes '5 9'.*



scan -, it's an operator, pop two operands from stack, apply the – operator on operands, we get 5 – 9 which results in -4. We push the result -4 to the stack. The stack now becomes '-4'.



- *There are no more elements to scan, we return the top element from the stack (which is the only element left in a stack).*

*So the result becomes **-4**.*

Prefix to infix evaluation:

Algorithm for Prefix to Infix:

- Read the Prefix expression in reverse order (from right to left)
- If the symbol is an operand, then push it onto the Stack
- If the symbol is an operator, then pop two operands from the Stack
  Create a string by concatenating the two operands and the operator between them.
  string = (operand1 + operator + operand2)
  And push the resultant string back to Stack
- Repeat the above steps until the end of Prefix expression.
- At the end stack will have only 1 string i.e resultant string

`Input :  Prefix :  *+AB-CD`

Reverse of the above prefix: DC-BA+*
PUSH D in stack
Push C in stack
After getting – start evaluating (operand 1+operator+operand 2) operand 1 will be top of the stack which in this case is C (repeat the step 1 to 3) operand 2 will be D just below top
C-D
B
A
+
A+B
*
(A+B)*(C-D)

```
Prefix:  *-A/BC-/AKL
Reverse the string
LKA/-CB/A-*
L
K
```

```
A
/
A/K
Now in stack A/K will be on top and L will be below the top of the
stack
-
((A/K)-L)
C
B
/
B/C
Now in stack ((A/K)-L)
              B/C
A
-
(A-(B/C))
*
(A-(B/C))* ((A/K)-L)
```

NOTE: For postfix to infix evaluation we should use operand2+operator+operand1
While doing prefix to infix evaluation we should use operand1+operator+operand2 with reverse string
Where operand 1 is top element operand 2 is element just below top in the stack

Advantages of Postfix over Prefix Notations:

- Postfix notation has fewer overheads of parenthesis. i.e., it takes less time for parsing.
- Postfix expressions can be evaluated easily as compared to other notations.