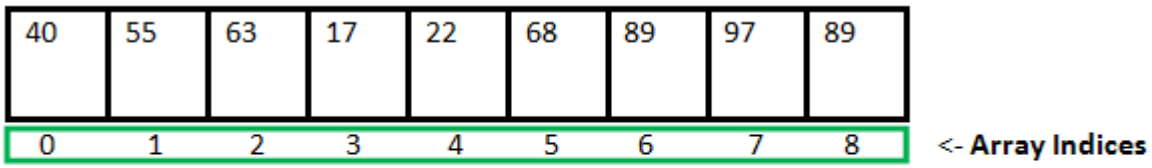
An array in C/C++ or be it in any programming language is a collection of similar data items stored at contiguous memory locations and elements that can be accessed randomly using indices of an array. They can be used to store the collection of primitive data types such as int, float, double, char, etc of any particular type. To add to it, an array in C/C++ can store derived data types such as structures, pointers, etc. Given below is the picture representation of an array.
an array is a container that can hold a fixed number of elements and these elements should be of the same type. Most of the data structures make use of arrays to implement their algorithms.

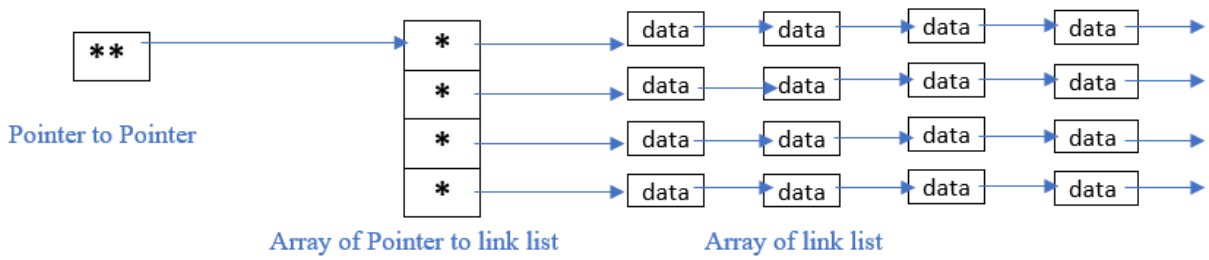


Array Length = 9
First Index = 0
Last Index = 8

A linked list is a linear data structure consisting of nodes where each node contains a reference to the next node. To create a link list we need a pointer that points to the first node of the list.
Approach: To create an array of linked lists below are the main requirements:

- An array of pointers.
- For keeping the track of the above-created array of pointers then another pointer is needed that points to the first pointer of the array. This pointer is called pointer to pointer.

Below is the pictorial representation of the array of linked lists:



Below is the C++ program to implement the array of linked lists:

- C++

```
// C++ program to illustrate the array
// of Linked Lists
#include <iostream>
using namespace std;

// Structure of Linked Lists
struct info {
    int data;
    info* next;
};

// Driver Code
int main()
{
    int size = 10;

    // Pointer To Pointer Array
    info** head;

    // Array of pointers to info struct
    // of size
```

```
head = new info*[size];


// Initialize pointer array to NULL
for (int i = 0; i < size; ++i) {
    *(head + i) = NULL;
}


// Traverse the pointer array
for (int i = 0; i < size; ++i) {

    // To track last node of the list
    info* prev = NULL;

    // Randomly taking 4 nodes in each
    // linked list
    int s = 4;

    while (s--) {

        // Create a new node
        info* n = new info;
```

```
            // Input the random data
            n->data = i * s;
            n->next = NULL;

            // If the node is first
            if (*(head + i) == NULL) {
                *(head + i) = n;
            }
            else {
                prev->next = n;
            }
            prev = n;
        }
    }

    // Print the array of linked list
    for (int i = 0; i < size; ++i) {
        info* temp = *(head + i);

        // Linked list number
        cout << i << "-->\t";
```

```
        // Print the Linked List
        while (temp != NULL) {
            cout << temp->data << " ";
            temp = temp->next;
        }

        cout << '\n';
    }

    return 0;
}
```

Output:

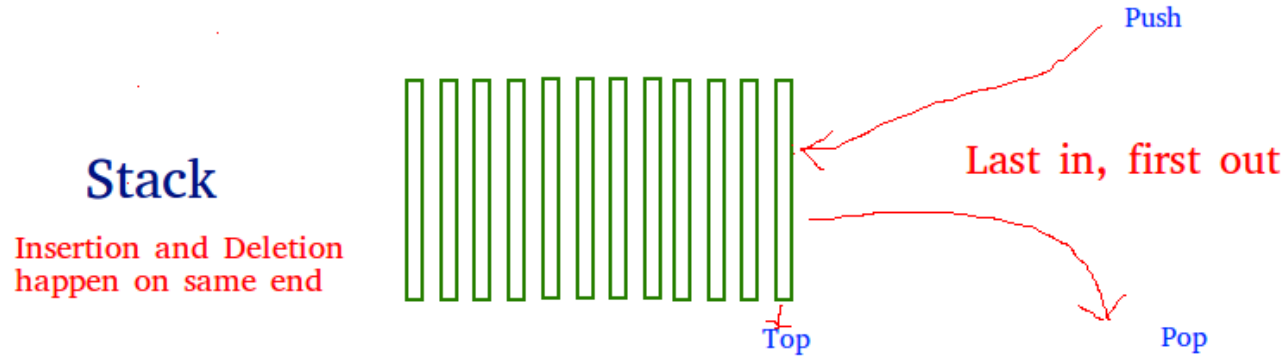
```
0-->    0 0 0 0
1-->    3 2 1 0
2-->    6 4 2 0
3-->    9 6 3 0
4-->    12 8 4 0
5-->    15 10 5 0
6-->    18 12 6 0
7-->    21 14 7 0
8-->    24 16 8 0
9-->    27 18 9 0
```

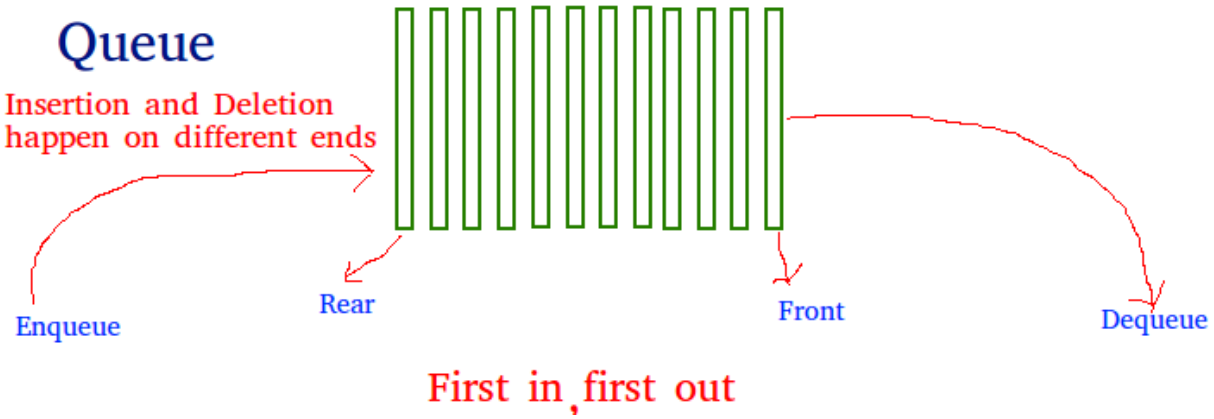
Time Complexity: O(size*4)
Here size is the number of rows of lists

Auxiliary Space: O(size*4)
The extra space is used to store the elements of the lists.

Implement a stack using queues. The stack should support the following operations:

1. **Push(x)**: Push an element onto the stack.
2. **Pop()**: Pop the element from the top of the stack and return it.





A Stack can be implemented using two queues. Let Stack to be implemented be ‘s’ and queues used to implement are ‘q1’ and ‘q2’.
Stack ‘s’ can be implemented in two ways:

By making push() operation costly – Push in O(n) and Pop() in O(1)

*The idea is to keep newly entered element at the front of **‘q1’** so that pop operation dequeues from **‘q1’**. **‘q2’** is used to move every new element in front of **‘q1’**.*

Follow the below steps to implement the **push(s, x)** operation:

- Enqueue x to q2.
- One by one dequeue everything from q1 and enqueue to q2.
- Swap the queues of q1 and q2.

Follow the below steps to implement the **pop(s)** operation:

- Dequeue an item from q1 and return it.

C++JavaPythonC#JavaScript

```cpp
/* Program to implement a stack using
two queue */
#include <bits/stdc++.h>

using namespace std;

class Stack {
    // Two inbuilt queues
    queue<int> q1, q2;

public:
    void push(int x)
    {
        // Push x first in empty q2
        q2.push(x);

        // Push all the remaining
        // elements in q1 to q2.
        while (!q1.empty()) {
            q2.push(q1.front());
            q1.pop();
        }

        // swap the names of two queues
        swap(q1, q2);
    }

    void pop()
    {
        // if no elements are there in q1
        if (q1.empty())
            return;
        q1.pop();
    }

    int top()
    {
        if (q1.empty())
            return -1;
        return q1.front();
    }

    int size() { return q1.size(); }
};

// Driver code
int main()
{
    Stack s;
    s.push(1);
    s.push(2);
```

```
    s.push(3);

    cout << "current size: " << s.size() << endl;
    cout << s.top() << endl;
    s.pop();
    cout << s.top() << endl;
    s.pop();
    cout << s.top() << endl;

    cout << "current size: " << s.size() << endl;
    return 0;
}
```

Output

```
current size: 3

3

2

1

current size: 1
```

Time Complexity:

- **Push operation:** O(n), As all the elements need to be popped out from the Queue (q1) and push them back to Queue (q2).
- **Pop operation**: O(1), As we need to remove the front element from the Queue.

Auxiliary Space: O(n), As we use two queues for the implementation of a Stack.

By making pop() operation costly – Push in O(1) and Pop() in O(n)

*The new element is always enqueued to **q1**. In **pop()** operation, if **q2** is empty then all the elements except the last, are moved to **q2**. Finally, the last element is dequeued from **q1** and returned.*

Follow the below steps to implement the **push(s, x)** operation:

- Enqueue x to q1 (assuming the size of q1 is unlimited).

Follow the below steps to implement the **pop(s)** operation:

- One by one dequeue everything except the last element from q1 and enqueue to q2.
- Dequeue the last item of q1, the dequeued item is the result, store it.
- Swap the names of q1 and q2
- Return the item stored in step 2.

C++JavaPythonC#JavaScript

```
// Program to implement a stack
// using two queue
#include <bits/stdc++.h>
using namespace std;

class Stack {
    queue<int> q1, q2;
```

```
public:
    void pop()
    {
        if (q1.empty())
            return;

        // Leave one element in q1 and
        // push others in q2.
        while (q1.size() != 1) {
            q2.push(q1.front());
            q1.pop();
        }

        // Pop the only left element
        // from q1
        q1.pop();

        // swap the names of two queues
        swap(q1, q2);
    }

    void push(int x) { q1.push(x); }

    int top()
    {
        if (q1.empty())
            return -1;

        while (q1.size() != 1) {
            q2.push(q1.front());
            q1.pop();
        }

        // last pushed element
        int temp = q1.front();

        // to empty the auxiliary queue after
        // last operation
        q1.pop();

        // push last element to q2
        q2.push(temp);

        // swap the two queues names
        queue<int> q = q1;
        q1 = q2;
        q2 = q;
        return temp;
    }

    int size() { return q1.size(); }
};
```

```
// Driver code
int main()
{
    Stack s;
    s.push(1);
    s.push(2);
    s.push(3);

    cout << "current size: " << s.size() << endl;
    cout << s.top() << endl;
    s.pop();
    cout << s.top() << endl;
    s.pop();
    cout << s.top() << endl;
    cout << "current size: " << s.size() << endl;
    return 0;
}
```

Output

```
current size: 3

3

2

1

current size: 1
```

Time Complexity:

- **Push operation:** O(1), As, on each push operation the new element is added at the end of the Queue.
- **Pop operation:** O(n), As, on each pop operation, all the elements are popped out from the Queue (q1) except the last element and pushed into the Queue (q2).

Auxiliary Space: O(n) since 2 queues are used.

Using single queue and Recursion Stack

Using only one queue and make the queue act as a Stack in modified way of the above discussed approach.

Follow the below steps to implement the idea:

- The idea behind this approach is to make one queue and push the first element in it.
- After the first element, we push the next element and then push the first element again and finally pop the first element.
- So, according to the FIFO rule of the queue, the second element that was inserted will be at the front and then the first element as it was pushed again later and its first copy was popped out.

- So, this acts as a Stack and we do this at every step i.e. from the initial element to the second last element, and the last element will be the one that we are inserting and since we will be pushing the initial elements after pushing the last element, our last element becomes the first element.

C++JavaPythonC#JavaScript

```cpp
#include <bits/stdc++.h>
using namespace std;

// Stack Class that acts as a queue
class Stack {

    queue<int> q;

public:
    void push(int data)
    {
        int s = q.size();

        // Push the current element
        q.push(data);

        // Pop all the previous elements and put them after
        // current element

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

            // Add the front element again
            q.push(q.front());

            // Delete front element
            q.pop();
        }
    }
    void pop()
    {
        if (q.empty())
            cout << "No elements\n";
        else
            q.pop();
    }
    int top() { return (q.empty()) ? -1 : q.front(); }
    int size() { return q.size(); }
    bool empty() { return (q.empty()); }
};

int main()
{
    Stack st;
    st.push(1);
    st.push(2);
    st.push(3);
```

```
    cout << "current size: " << st.size() << "\n";
    cout << st.top() << "\n";
    st.pop();
    cout << st.top() << "\n";
    st.pop();
    cout << st.top() << "\n";
    cout << "current size: " << st.size();
    return 0;
}
```

Output

```
current size: 3

3

2

1

current size: 1
```

Time Complexity:

- **Push operation:** O(n)
- **Pop operation**: O(1)

Auxiliary Space: O(n) since 1 queue is used.

Write a program to convert an Infix expression to Postfix form.
Infix expression: The expression of the form “a operator b” (a + b) i.e., when an operator is in-between every pair of operands.
Postfix expression: The expression of the form “a b operator” (ab+) i.e., When every pair of operands is followed by an operator.
Examples:
Input: *s = “A*(B+C)/D”*
Output: *ABC+*D/*
Input: *s = “a+b*(c^d-e)^(f+g*h)-i”*
Output: *abcd^e-fgh*+^*+i-*

Why postfix representation of the expression?

The compiler scans the expression either from left to right or from right to left.
Consider the expression: **a + b * c + d**

- The compiler first scans the expression to evaluate the expression b * c, then again scans the expression to add a to it.
- The result is then added to d after another scan.

The repeated scanning makes it very inefficient. Infix expressions are easily readable and solvable by humans whereas the computer cannot differentiate the operators and parenthesis easily so, it is better to convert the expression to postfix (or prefix) form before evaluation.
The corresponding expression in postfix form is **abc*+d+**. The postfix expressions can be evaluated easily using a stack.

Conversion of an Infix expression to Postfix expression

To convert infix expression to postfix expression, use the ***stack data structure****. Scan the infix expression from left to right. Whenever we get an operand, add it to the postfix expression and if we get an operator or parenthesis add it to the stack by maintaining their precedence.*

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 of the current scanned operator is higher than the precedence of the operator on top of the stack, or if the stack is empty, or if the stack contains a '(', then push the current operator onto the stack.
    - Else, pop all operators from the stack that have precedence higher than or equal to that of the current operator. After that push the current operator onto 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.

Illustration:

```cpp
#include <bits/stdc++.h>
using namespace std;

int prec(char c) {
    if (c == '^')
        return 3;
    else if (c == '/' || c == '*')
        return 2;
    else if (c == '+' || c == '-')
        return 1;
    else
        return -1;
}

string infixToPostfix(string s) {
    stack<char> st;
    string res;

    for (int i = 0; i < s.length(); i++) {
        char c = s[i];

        // If the scanned character is
        // an operand, add it to the output string.
        if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || (c >= '0' && c
<= '9'))
            res += c;

        // If the scanned character is an
        // ‘(‘, push it to the stack.
        else if (c == '(')
            st.push('(');

        // If the scanned character is an ‘)’,
        // pop and add to the output string from the stack
        // until an ‘(‘ is encountered.
        else if (c == ')') {
            while (st.top() != '(') {
                res += st.top();
                st.pop();
            }
            st.pop();
        }

        // If an operator is scanned
        else {
            while (!st.empty() && prec(c) <= prec(st.top())) {
                res += st.top();
                st.pop();
            }
            st.push(c);
        }
    }
```

```
    // Pop all the remaining elements from the stack
    while (!st.empty()) {
        res += st.top();
        st.pop();
    }

    return res;
}

int main() {
    string exp = "a+b*(c^d-e)^(f+g*h)-i";
    cout << infixToPostfix(exp);
    return 0;
}
```

Output

```
abcd^e-fgh*+^*+i-
```

Time Complexity: O(n), where n is the size of the infix expression
Auxiliary Space: O(n), where n is the size of the infix expression

nfix : An expression is called the Infix expression if the operator appears in between the operands in the expression. Simply of the form (operand1 operator operand2).
Example : (A+B) * (C-D)
Prefix : An expression is called the prefix expression if the operator appears in the expression before the operands. Simply of the form (operator operand1 operand2).
Example : *+AB-CD (Infix : (A+B) * (C-D))
Given a Prefix expression, convert it into a Infix expression.
Computers usually does the computation in either prefix or postfix (usually postfix). But for humans, its easier to understand an Infix expression rather than a prefix. Hence conversion is need for human understanding.
Examples:
Input : *Prefix : *+AB-CD*
Output : *Infix : ((A+B)*(C-D))*

Input : *Prefix : *-A/BC-/AKL*
Output : *Infix : ((A-(B/C))*((A/K)-L))*

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

Postfix: An expression is called the postfix expression if the operator appears in the expression after the operands. Simply of the form (operand1 operand2 operator).
Example : AB+CD-* (Infix : (A+B) * (C-D))
Prefix : An expression is called the prefix expression if the operator appears in the expression before the operands. Simply of the form (operator operand1 operand2).
Example : *+AB-CD (Infix : (A+B) * (C-D))
Given a Postfix expression, convert it into a Prefix expression.
Conversion of Postfix expression directly to Prefix without going through the process of converting them first to Infix and then to Prefix is much better in terms of computation and better understanding the expression (Computers evaluate using Postfix expression).

Examples:
Input : *Postfix : AB+CD-**
Output : *Prefix : *+AB-CD*
Explanation : *Postfix to Infix : (A+B) * (C-D)*
*Infix to Prefix : *+AB-CD*

Input : *Postfix : ABC/-AK/L-**
Output : *Prefix : *-A/BC-/AKL*
Explanation : *Postfix to Infix : ((A-(B/C))*((A/K)-L))*
*Infix to Prefix : *-A/BC-/AKL*

Algorithm for Postfix to Prefix:

- *Read the Postfix expression from left to right*
- *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 before them.
  string = operator + operand2 + operand1
  And push the resultant string back to Stack
- *Repeat the above steps until end of Postfix expression.*

Given an **infix expression** consisting of operators (+, -, *, /, ^) and operands (lowercase characters), the task is to convert it to a **prefix expression**.
Infix Expression: *The expression of type* ***a 'operator' b*** *(a+b, where + is an operator) i.e., when the operator is between two operands.*
Prefix Expression: *The expression of type '****operator' a b*** *(+ab where + is an operator) i.e., when the operator is placed before the operands.*

Examples:
Input: *a*b+c/d*
Output: *+*ab/cd*
Input: *(a-b/c)*(a/k-l)*
Output: **-a/bc-/akl*

Approach:
The idea is to first reverse the given infix expression while swapping '(' with ')' and vice versa, then convert this modified expression to postfix

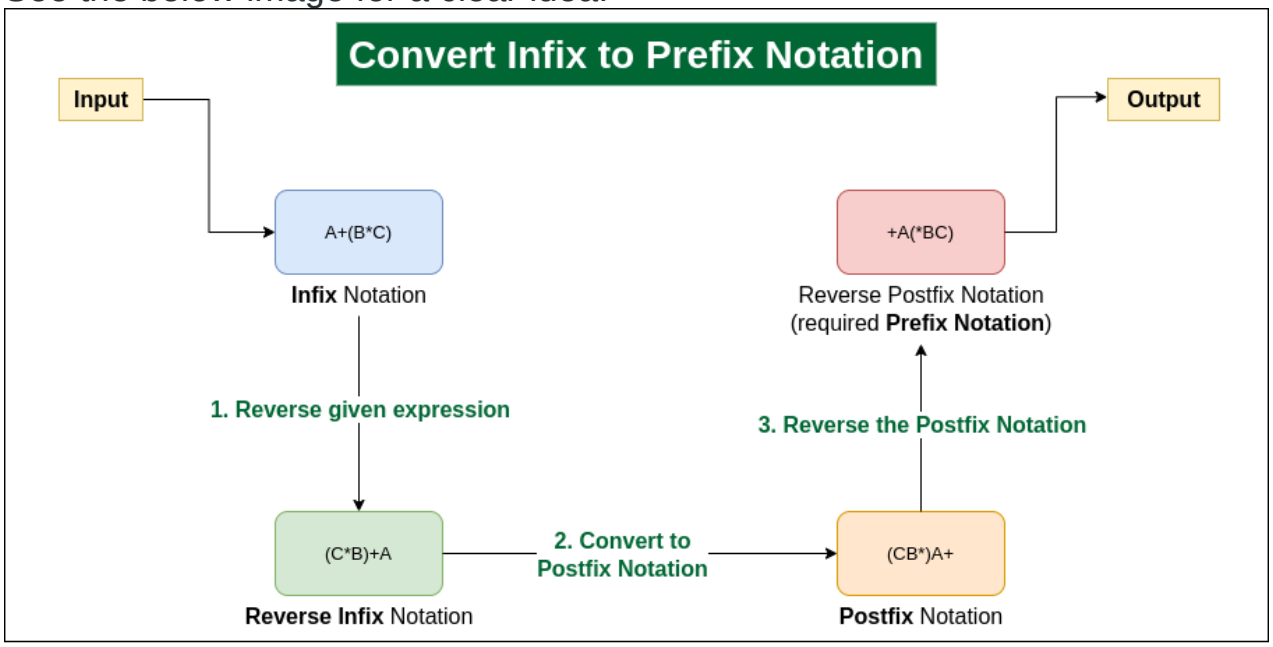
notation using a stack-based approach that follows operator precedence and associativity rules, and finally reverse the obtained postfix expression to get the prefix notation.

Step by step approach:

1. Reverse the infix expression. Note while reversing each ‘(‘ will become ‘)’ and each ‘)’ becomes ‘(‘.
2. Convert the reversed infix expression to postfix expression.
   - **Initialize an empty stack** to store operators and an empty string for the postfix expression.
   - Scan the infix expression from left to right.
   - **If the character is an operand,** append it to the postfix expression.
   - **If the character is ‘(‘,** push it onto the stack.
   - **If the character is ‘)’,** pop from the stack and append to the postfix expression until ‘(‘ is found, then pop ‘(‘ without appending.
   - **If the character is an operator,** pop and append operators from the stack until the stack is empty or a lower precedence operator is found, then push the current operator onto the stack.
   - **After scanning the expression,** pop and append all remaining operators from the stack to the postfix expression.
3. Reverse the postfix expression and return it.

Illustration:

See the below image for a clear idea:



Convert infix expression to prefix expression