Lab Report - Tutorial Writing
Assignment no. 03
CSE 4614 - Technical Report Writing

Assignment topic : Stack

S. M. Rayeed
ID : 160041045
Lab Group - 1B

July 27, 2019

1 Introduction

1.1 What is Stack?

A stack is an Abstract Data Type which allows operations at one end only. It contains data in reverse order. At any given time, we can only access the top element of a stack. This feature makes it LIFO data structure – where the element which is placed last, is accessed first.

In other words, a stack is a recursive data structure. Here are the key factors of a stack:

- a stack is a container of objects or simply data
- a stack is either empty or full
  - if empty, no data can be removed or popped
  - if full, no data can be added or pushed
- it consists of a top which is the only accessible position

In stack terminology, insertion operation is called PUSH operation and removal operation is called POP operation.

1.2 Stack Representation

As mentioned earlier, stack has two basic operations – PUSH for insertion and POP for deletion; and only the TOS (Top of Stack) is accessible. Now, if we focus on how a stack can be represented along with its operations in a graphical way, the following figure is a good representation –

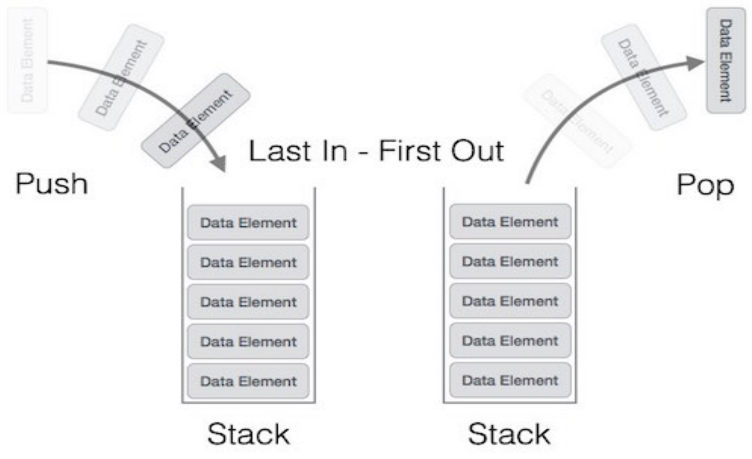

Figure 1: Stack Representation

2 Data Storage in Stack

In a stack, data or objects are stored in reverse order. If any insertion operation occurs in an empty stack, the data will be TOS element. But after that, the next data inserted, will be placed on top of the first data. The following figure will give us a clear view about data storage in stack –

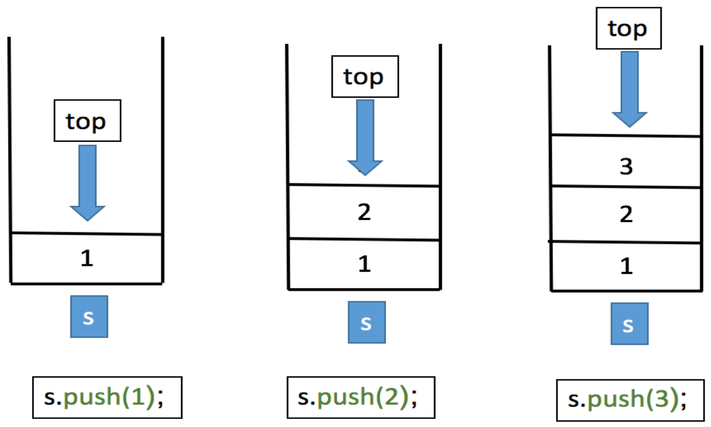

Figure 2: Stack Storage

In the above figure, at first 1 has been inserted into an empty stack named "s" using the PUSH command. Now, this 1 is the TOS (top of stack) element. After that, 2 has been inserted and has been placed on top of 1 – meaning that now it is the TOS and similarly for every insertion, the storage of data will be in reverse order. Also one more thing to remember, we can only access the TOS at any given time.

One more thing to mention, size of a stack can either be fixed or it can dynamically change. If the size is fixed, once the stack is full, we cannot store any further data. But the size can be adjusted dynamically through stack-implementation using linked-list which we will focus later on.

3 Operations in Stack

Operations in stack is pretty-much straight-forward – we only have the freedom to insert and delete any element. Moreover, we have restrictions in insertion and deletion –

- insertion can only be at the top of stack (TOS)
- deletion can only occur at the top of stack (TOS)

These restrictions simply means – only the top element of stack is accessible. In a stack, there can be two operations – PUSH operation and POP operation.

3.1 PUSH Operation – Inserting Elements

When we want to insert an element into a stack, PUSH operation is conducted using the command stackname.push(element_to _be _pushed)) in standard template library. Steps in PUSH operation is mentioned below –

- Step 1 – Checks if the stack is full.
- Step 2 – If the stack is full, produces an error and exit.
- Step 3 – If the stack is not full, increments the TOS(Top of Stack) to point the immediate next empty space.
- Step 4 – Adds data element to the stack location where TOS is pointing to.
- Step 5 – Returns successful insertion.

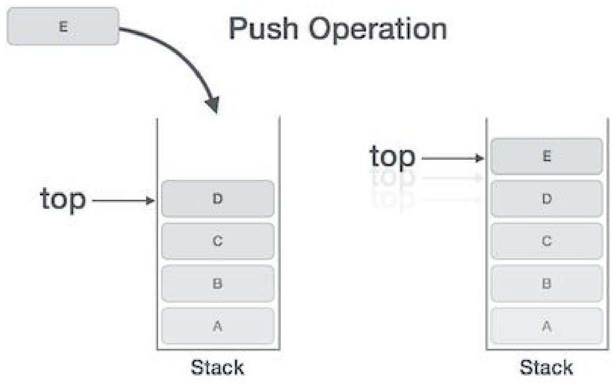

Figure 3: Push Operation in Stack

3.2 POP Operation – Deleting Elements

When we want to delete an element into a stack, POP operation is conducted using the command stackname.pop() in standard template library. Steps in POP operation is mentioned below –

- Step 1 – Checks if the stack is empty.
- Step 2 – If the stack is empty, produces an error and exit.
- Step 3 – If the stack is not empty, accesses the data element at which top is pointing.
- Step 4 – Decreases the value of top by 1.
- Step 5 – Returns successful deletion.

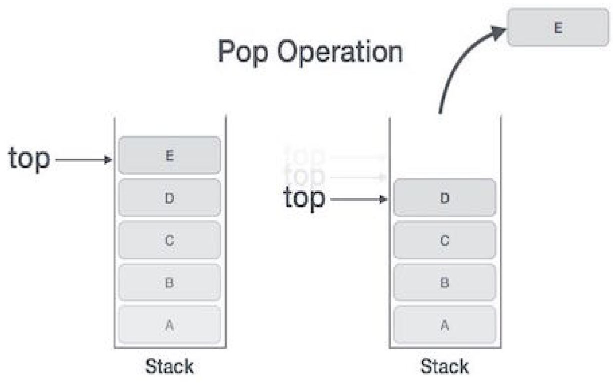

Figure 4: Pop Operation in Stack

3.3 Push and POP Operation – Graphical Representation

Here is an exmple how the push and pop operation operates in a stack

- Stage 1 – 1 is the top element of the stack
- Stage 2 – stack.push(4) operates on the stack; which means 4 has been inserted and now it is the top element
- Stage 3 – stack.pop() will remove the Top element i.e. 4 from the stack
- Finally, again 1 is at the Top of stack

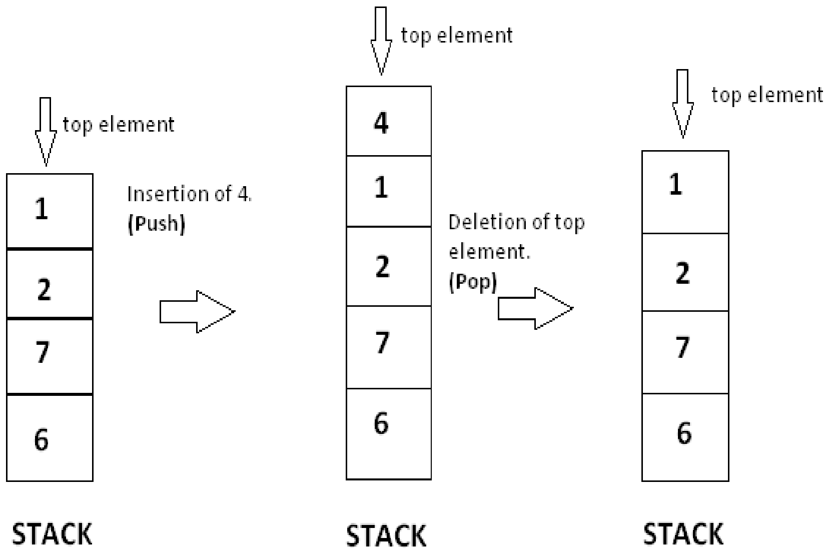

Figure 5: Operations in Stack – An Example

3.4 Additional Operations

To ensure the efficient use of a stack, we need to check the states of the stack. The following operations suffice that –

- isFull() :
  - checks if stack is full.
  - if full, push() operation cannot be conducted.
- isEmpty() :
  - checks if stack is empty.
  - if empty, pop() operation cannot be conducted.
- peek() : returns stack[top] – provides access to the top element of the stack. Unlike pop, does not remove the top element. Also known as top() operation.
- size() : determines the size of the stack.

Also, the position of the Top element determines the status of the stack. Have a look at the following table –

Determining Status of Stack via TOS	
Position of TOS	Status of Stack
-1	Stack is Empty
0	One Element in the Stack
N-1	Stack is Full
N	Overflow State of Stack

4 Implementations of Stack

Stack can be easily implemented using an Array or a Linked List. Arrays are quick, but are limited in size and Linked List requires overhead to allocate, link, unlink and deallocate; but is not limited in size. There are many different approaches in implementation of stack.

4.1 Implementations of Stack – Standard Template Library

In the standard template library of classes, the data type stack is an adapter class, meaning that a stack is built on top of other data structures. The functions associated with stack are :

- empty() – Returns whether the stack is empty
- size() – Returns the size of the stack
- top() – Returns a reference to the top most element of the stack
- push(A) – Adds the element A at the top of the stack
- pop() – Deletes the top most element of the stack

4.1.1 Stack in STL - Code

Here is an exemplary C++ code that depicts Stack and its operations in STL. The Function showstack() displays the current state of the stack while being called –

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

void showstack(stack <int> s)
{
        while (!s.empty()){
                cout << '\t' << s.top() << endl;
                s.pop();
        }
        cout << '\n';
}

int main ()
{
        stack <int> s;
        s.push(10);
        s.push(30);
        s.push(20);
        s.push(5);
        s.push(1);
        cout << "Stack:" << endl;
        showstack(s);
        cout << "Size:\t" << s.size() << "\nTOS:\t" << s.top();
        cout << "\n\nPOP:" << endl;
        s.pop();
        showstack(s);

        return 0;
}
```

4.2 Implementations of Stack – Arrays

Array implementation of stack is easy but the size is static – means the stack cannot be dynamically increasing or decreasing. Unlike the STL implementation, in array implementation, the stack-operations are not built-in; hence we need to define the operational functions like – push(), pop(), top(), isEmpty(), isFull() etc.

As we need to define the functions, it is very much necessary to know how each functions work and for that, the best way is to know the algorithms that each of these follows.

4.2.1 Algorithm - isFull()

isFull() is a boolean function that returns a true value if the stack is full, otherwise returns false. Now, how do we know when the stack is full? Well, in array representation, we define the maximum size of the array – in this case which is our stack. Now, while insertion, the value of the TOS gradually increases. If the TOP equals to the maximum size, it means that it has reached the limit; no value can be inserted unless popping. Here, the algorithm is given below –

```
bool isFull()
{
    if (top == MAXSIZE)
        return true ;
    else
        return false;
}
```

One thing to notice, if the size of the array is declared as N, then the maxsize will be N-1; because array elements starts from 0.

4.2.2 Algorithm - isEmpty()

isEmpty() is also a boolean function that returns a true value if the stack is empty, otherwise returns false. Now, how do we know when the stack is empty? Well, quite easy! In array representation, we generally initialize the index of the array at -1; because the array indexing starts from 0, so when the first element of the array is being inserted, the index should be 0. It implies if the index value is -1 then the array is empty. The concept is very much straightforward and is shown below –

```
bool isEmpty ( )
{
    if (top == -1)
        return true ;
    else
        return false;
}
```

As index-value of -1 implies that the array is empty, the first element inserted will have the index-value of 0 – which suffices the array-condition.

4.2.3 Algorithm - TOS()

TOS() is a function that returns the element that is on the top position of a stack. The type of this function depends on the type of the array. The function is necessary because in stack, we only can operate on the top element. And after every insertion or deletion operation, the top element changes. This function has no functionality rather than this –

```
int TOS( )
{
    return stack[top];
}
```

Here, stack is an array where the indexing starts from 0 and gradually increases by 1 after an insertion and decreases by 1 after a deletion operation. The integer-type variable "TOP" stores the index-value of the element which is in the top of stack. So, when this function is called, it returns the element having the TOP index – that is how it works.

4.2.4 Algorithm - push()

push() is one of the most-influential function in stack as it operates to insert an element. It is a parameterized function that checks whether the stack is full or not; and if not, it stores the element on top of the stack and increases the value of TOP (as TOP stores the index-value of the top-most element, after every insertion, its value should be increased by 1). Well, needless to say, but if the stack is full, then the function will show "Overflow" and quit its operation. Here one example of the push function is being shown –

```
void push (int stack[ ] , int x , int n)
{
    if ( top == n−1 )
    {
        cout << "Overflow" ;
    }

    else
    {
        top = top +1 ;
        stack[ top ] = x ;
    }
}
```

Here, n is the size of the stack, x is the element that needs to be inserted in the stack. As array-indexing starts from 0; in an array of size n, the top-most element will have an index of n-1 (0 to n-1 : Total n number of elements). So, if the value of top reaches the limit, push() will show "Overflow", otherwise it will increase the value of top - that means top will point to the next position and in that position of stack, the element x will be inserted.

Well, one thing to mention, as we have defined isFull() function, instead of checking manually, we could simply call that function to determine whether the stack is full or not. In pop() operation, it has been shown.

4.2.5 Algorithm - pop()

pop() is also a very influential function in stack as it operates to delete an element. It is a parameterized function that checks whether the stack is empty or not; and if not, it deletes the element on top of the stack and decreases the value of TOP (as TOP stores the index-value of the top-most element, after every deletion, its value should be decreased by 1). Well, needless to say, but if the stack is empty, then the function will show "Underflow" and quit its operation. Here one example of the pop function is being shown –

```
void pop(int stack[ ] , int n )
{
    if(isEmpty())
        cout << "Underflow" << endl ;
    else
        top = top - 1;
}
```

Here, n is the size of the stack. As we have already defined the isEmpty() function, we can easily implement pop() by calling the isEmpty() function. If the stack is empty, the isEmpty() function will return True, hence nothing is there to be popped. So, the pop() will show "Underflow Condition", but otherwise it will decrease the value of top – meaning, the element that was next to the top-most element is now being pointed by top; hence the top-most element is gone.

4.2.6 Stack using Array - Code

Here is an exemplary C++ code that depicts Stack and its operations using array. Unlike STL, the functions are not pre-defined –

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

int top = -1;
int n = 3;

bool isEmpty ()
{
    if ( top == -1 )
        return true ;
    else
        return false;
}

bool isFull()
{
    if (top == n-1)
        return true ;
    else
        return false;
}

void push (int stack[ ] , int x)
{
    if (isFull())
        cout << "Overflow" << endl;
    else
    {
        top += 1 ;
        stack[ top ] = x ;
    }
}

void pop ( )
{

    if( isEmpty())
        cout << "Underflow" << endl ;
    else
        top -= 1 ;
}

int size ( )
```

```
{
    return top + 1;
}

int topElement(int stack[])
{
    return stack[top];
}


int main( )
{

    int stack[n];
    push(stack , 5 ) ;
    cout << "Size:\t" << size ( ) << endl ;
    push(stack , 10 );
    push (stack , 24 ) ;
    cout << "Size:\t" << size( ) << endl ;
    push(stack , 12 ) ;
    cout << "TOS:\t" << topElement(stack) << endl;

    for(int i = 0 ; i < 3; i++ )
        pop( );

    cout << "Size:\t" << size( ) << endl ;
    pop ( );

    return 0;
}
```

4.3 Implementations of Stack – Linked Lists

A stack can be easily implemented through the linked list. In such stack implementation, a stack contains a top pointer. which is âĂIJheadâĂİ of the stack where pushing and popping items happens at the head of the list. first node have null in link field and second node link have first node address in link field and so on and last node address in "top" pointer.

The main advantage of using linked list over an arrays is that it is possible to implements a stack that can shrink or grow as much as needed. In using array will put a restriction to the maximum capacity of the array which can lead to stack overflow. Here each new node will be dynamically allocate. so overflow is not possible.

4.3.1 Implementation of Stack using Linked Lists – Functions

In linked list implementation, we have the same functions as before – having the same analogy, but with different approach of implementations. Also as mentioned before, since the stack can dynamically expand or shrink, there's no need of overflow. The generic functions are mentioned below –

1. push() : Insert the element into linked list at the top node of Stack.
2. pop() : Return top element from the Stack and move the top pointer to the second node of linked list.
3. peek(): Return the top element.
4. display(): Print all elements of Stack.

4.3.2 Stack using Linked Lists – push() function

- Utility function to add an element data in the stack
- Steps in push() operation –
  - creates new node temp and allocate memory
  - initializes data into temp data field
  - puts top pointer reference into temp link
  - makes temp as top of Stack
- An example of the push() function is attached below –

```
void push(int data)
{
    struct Node* temp;
    temp = new Node();
    if (!temp)
    {
        cout << "\nOverflow";
        exit(1);
    }
    temp->data = data;
    temp->link = top;
    top = temp;
}
```

4.3.3 Stack using Linked Lists – pop() function

- Utility function to pop the top element data from the stack
- Steps in pop() operation –
  - checks for stack underflow
  - top assign into temp
  - assigns second node to top
  - destroys connection between first and second
  - releases memory of top node
- An example of the pop() function is attached below –

```
void pop()
{
    struct Node* temp;
    if (top == NULL)
    {
        cout << "\nUnderflow" << endl;
        exit(1);
    }
    else
    {
        temp = top;
        top = top->link;
        temp->link = NULL;
        free(temp);
    }
}
```

4.3.4 Stack using Linked Lists – peek() function

- Checks for empty stack and if empty, exits
- Otherwise, returns top element data
- An example of the peek() function is attached below –

```
int peek()
{
    if (!isEmpty()) return top->data;
    else exit(1);
}
```

4.3.5 Stack using Linked Lists – display() function

- Function to print all the elements of the stack
- Steps in display() operation –
  - check for stack underflow
  - if yes, exits
  - otherwise, print node data until temp gets zero
- An example of the display() function is attached below –

```
void display()
{
    struct Node* temp;
    if (top == NULL)
    {
        cout << "\nUnderflow";
        exit(1);
    }
    else
    {
        temp = top;
        while (temp != NULL)
        {
            cout << temp->data << "␣";
            temp = temp->link;
        }
    }
}
```

4.3.6 Stack using Linked Lists – Complete Code

Here is an exemplary C++ code that depicts Stack and its operations in Linked-lists –

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

struct Node
{
    int data;
    struct Node* link;
};
```

```
struct Node* top;

void push(int data)
{
    struct Node* temp;
    temp = new Node();
    if (!temp)
    {
        cout << "\nOverflow";
        exit(1);
    }
    temp->data = data;
    temp->link = top;
    top = temp;
}

int isEmpty()
{
    return top == NULL;
}

int peek()
{
    if (!isEmpty()) return top->data;
    else exit(1);
}

void pop()
{
    struct Node* temp;
    if (top == NULL)
    {
        cout << "\nUnderflow" << endl;
        exit(1);
    }
    else
    {
        temp = top;
        top = top->link;
        temp->link = NULL;
        free(temp);
    }
}

void display()
{
    struct Node* temp;
    if (top == NULL)
    {
        cout << "\nUnderflow";
        exit(1);
    }
    else
    {
        temp = top;
        while (temp != NULL)
        {
            cout << temp->data << "␣";
            temp = temp->link;
        }
    }
}

int main()
{
    push(11);
    push(22);
    push(33);
    push(44);
    display();
    cout << "\nTOS:%d\n" << peek();
    pop();
    pop();
    display();
    cout << "\nTOS:%d\n" << peek();

    return 0;
}
```

5 Example – Infix to Postfix Conversion

There are huge number of examples and real-life implementations of stack. Such as Deck of cards, pile of dishes in kitchen, the tower of Hanoi problem and so on. From the data-structure perspective, a good example of stack is conversion of expressions. There are 3 types of expression –

1. Prefix – Expression of the form operator a b. When an operator is in front of every pair of operands.
2. Infix – Expression of the form a operator b. When an operator is in-between every pair of operands.
3. Postfix – Expression of the form a b operator. When an operator is at the back of every pair of operands.

Infix to Postfix is a very renowned conversion technique that is implemented by stack. We are going to have a look into this –

5.1 Why Postfix representation of the expression?

The compiler scans the expression either from left to right or from right to left. Consider the below expression:

a op1 b op2 c op3 d ;   where –
op1 = +, op2 = *, op3 = +

- The compiler first scans the expression to evaluate the expression : b * c
- Then again scan the expression to add a to it
- The result is then added to d after another scan
- The repeated scanning makes it very in-efficient
- That's why it is better to convert the expression to Postfix form before evaluation

The corresponding expression in postfix form is:

$$abc * +d+$$

The postfix expressions can be evaluated easily using a stack by following the following algorithm –

5.2 Infix to Postfix Conversion – Algorithm

1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else –

- If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a "("), then push it.
- 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 an "(", push it to the stack.
5. If the scanned character is an ")", pop the stack and and output it until a "(" is encountered, and discard both the parenthesis.
6. Repeat steps 2-6 until infix expression is scanned.
7. Print the output
8. Pop and output from the stack until it is not empty.

5.3 Conversion of A * B + C * D : –

Figure 6 Shows the conversion algorithm working on the expression –

$$A * B + C * D$$

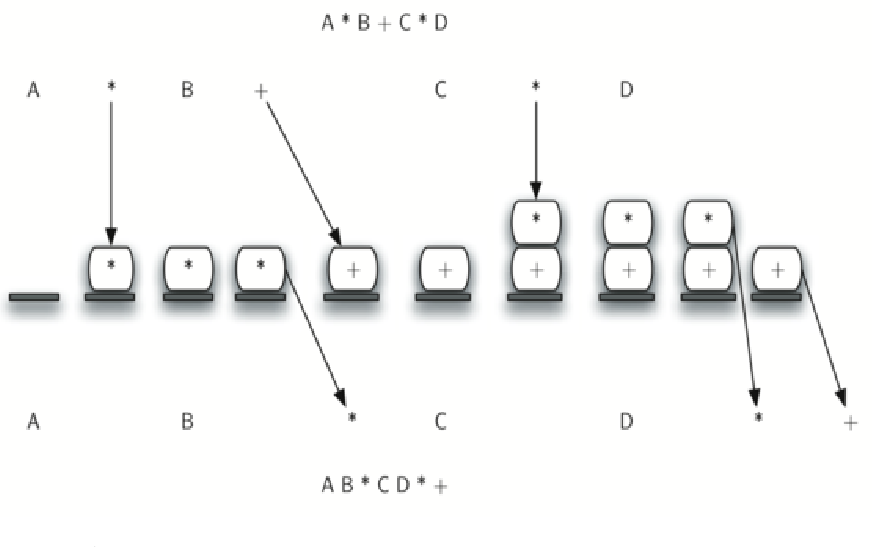

Figure 6: Infix to Postfix Conversion - An Example

Note that the first * operator is removed upon seeing the + operator. Also, + stays on the stack when the second * occurs, since multiplication has precedence over addition. At the end of the infix expression the stack is popped twice, removing both operators and placing + as the last operator in the postfix expression.

5.4 Infix to Postfix Conversion – Implementation using C++

Here is an exemplary C++ code that converts an infix experession into its corresponding postfix expression –

```
#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;
}

void infixToPostfix(string s)
{
    std::stack<char> st;
    st.push('N');
    int l = s.length();
    string ns;
    for(int i = 0; i < l; i++)
    {

        if((s[i] >= 'a' && s[i] <= 'z')||(s[i] >= 'A' && s[i] <= 'Z'))
            ns+=s[i];
        else if(s[i] == '(')
            st.push('(');
        else if(s[i] == ')')
        {
            while(st.top() != 'N' && st.top() != '(')
            {
                char c = st.top();
                st.pop();
                ns += c;
            }
            if(st.top() == '(')
            {
                char c = st.top();
                st.pop();
            }
        }
        else
        {
            while(st.top() != 'N' && prec(s[i]) <= prec(st.top()))
            {
                char c = st.top();
                st.pop();
                ns += c;
            }
            st.push(s[i]);
        }
    }

    while(st.top() != 'N')
    {
        char c = st.top();
        st.pop();
        ns += c;
    }

    cout << ns << endl;
}

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

6 Complexity Analysis

Below mentioned are the time complexities for various operations that can be performed on the Stack data structure –

1. Push Operation : O(1) –
    - While pushing an element, we can only insert in top of the stack.
    - As the insertion can be in one-way, it's a one-step process.
    - That's why the time complexity is fixed to O(1).

2. Pop Operation : O(1) –
    - While popping element, we can only pop the top element from the stack
    - As the deletion can be in one-way, it's a one-step process.
    - That's why the time complexity is fixed to O(1).
3. Top Operation : O(1) –
    - Only the top element of the stack is accessible.
    - Accessing the top-most element in a stack is a one-step process as we do not have to search elsewhere.
    - That's why the time complexity is fixed to O(1).
4. Other cases –
    - Access O(n) – Stacks offer random access to their contents by popping the top element off the stack. You have to do this repeatedly to access an arbitrary element somewhere in the stack. Therefore, arbitrary access is O(n).
    - Search O(n) – Searching for a given value in the stack requires repeatedly popping elements off the top of the stack until you find the value you seek. So search is O(n).

To talk about space complexity, we need to know what the problem is. If n items are needed to be stored in the stack at the same time, then space complexity is O(n). But n items, can be stored in O(1) space too. By pushing and popping every item, we can use only 1 space.

7 Applications

There are many applications of stack in data structure. Some of them are mentioned below in short –

1. Expression evaluation
    - Stack is used to evaluate prefix, postfix and infix expressions.
    - In particular let's consider arithmetic expressions. Suppose that, there are boolean and logical expressions that can be evaluated in the same way. Control structures can also be treated similarly in a compiler.
    - This study of arithmetic expression evaluation is an example of problem solving where we have to solve a simpler problem and then transform the actual problem to the simpler one.

- We are accustomed to write arithmetic expressions with the operation between the two operands: a+b or c/d. If we write a+b*c, however, we have to apply precedence rules to avoid the ambiguous evaluation.
- But there's no real reason to put the operation between the variables or values. They can just as well precede or follow the operands. Moreover, the advantage of prefix and postfix is the need for precedence rules and parentheses are eliminated.

2. Backtracking

    - Backtracking is used in algorithms in which there are steps along some path (state) from some starting point to some goal
        - Find your way through a maze
        - Find a path from one point in a graph (roadmap) to another point
        - Play a game in which there are moves to be made (checkers, chess)
    - In all of these cases, there are choices to be made among a number of options. We need some way to remember these decision points in case we want/need to come back and try the alternative.
    - Consider the maze. At a point where a choice is made, we may discover that the choice leads to a dead-end. We want to retrace back to that decision point and then try the other (next) alternative.
    - Again, stacks can be used as part of the solution. Recursion is another, typically more favored, solution, which is actually implemented by a stack.

3. Memory management

    - Any modern computer environment uses a stack as the primary memory management model for a running program. Whether it's native code (x86, Sun, VAX) or JVM, a stack is at the center of the run-time environment for Java, C++, Ada, FORTRAN, etc.
    - The discussion of JVM in the text is consistent with NT, Solaris, VMS, Unix runtime environments.
    - Each program that is running in a computer system has its own memory allocation containing the typical layout as shown below.

4. Call and return process

    - When a method/function is called –
        (a) An activation record is created; its size depends on the number and size of the local variables and parameters.
        (b) The Base Pointer value is saved in the special location reserved for it
        (c) The Program Counter value is saved in the Return Address location

   (d) The Base Pointer is now reset to the new base (top of the call stack prior to the creation of the AR)
   (e) The Program Counter is set to the location of the first bytecode of the method being called
   (f) Copies the calling parameters into the Parameter region
   (g) Initializes local variables in the local variable region

- While the method executes, the local variables and parameters are simply found by adding a constant associated with each variable/parameter to the Base Pointer.
- When a method returns –
   (a) Get the program counter from the activation record and replace what's in the PC
   (b) Get the base pointer value from the AR and replace what's in the BP
   (c) Pop the AR entirely from the stack.

8 Conclusion

Stack is a very well-known and one of the most-frequently used data structures. Having less complexity, ease of understanding and numerous applications in our life – stack has been playing a vital role in development and implementation of new techniques.

Thank You ...