Data Structures and Algorithms

Mr. Tahir Iqbal

tahir.iqbal@bahria.edu.pk

Lecture 07: Dynamic Stack

Stack applications

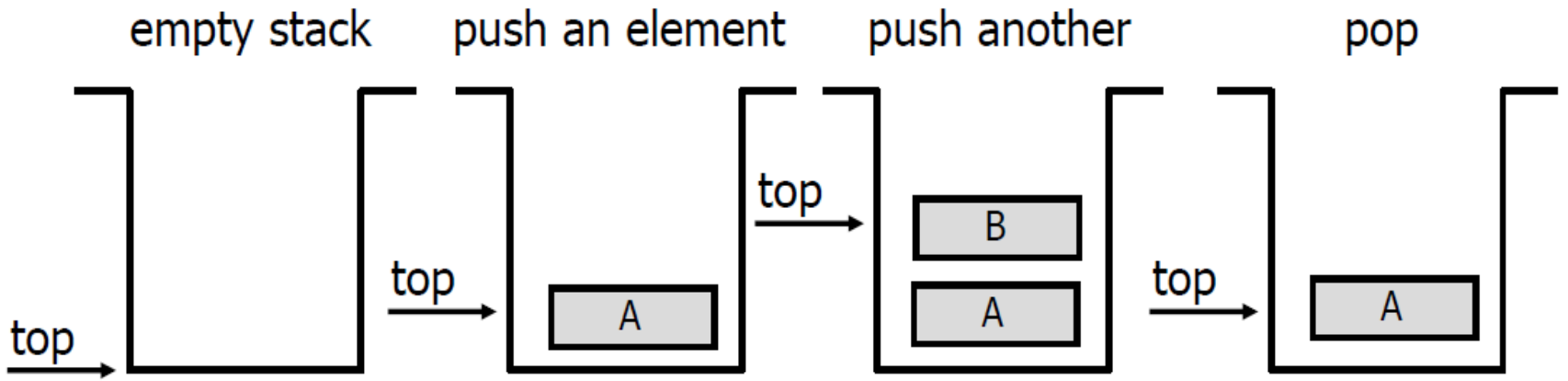
- “Back” button of Web Browser
  - History of visited web pages is pushed onto the stack and popped when “back” button is clicked
- “Undo” functionality of a text editor
- Reversing the order of elements in an array
- Saving local variables when one function calls another, and this one calls another, and so on.

Operation On Stack

- Creating a stack
- Checking stack---- either empty or full
- Insert (PUSH) an element in the stack
- Delete (POP) an element from the stack
- Access the top element
- Display the elements of stack

Push and Pop

- Primary operations: Push and Pop
- Push
  - Add an element to the top of the stack.
- Pop
  - Remove the element at the top of the stack.



Stack-Related Terms

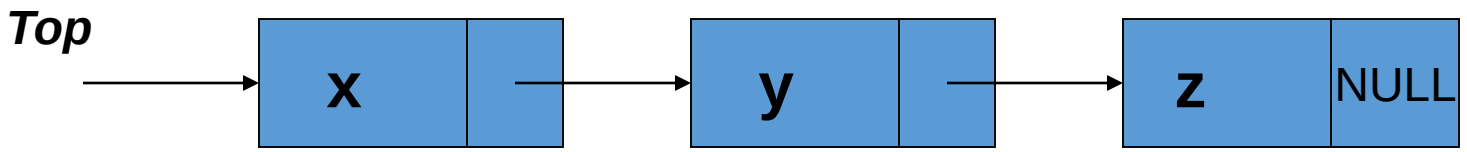
- Top
  - A pointer that points the top element in the stack.
- Stack Underflow
  - When there is no element in the stack, the status of stack is known as stack underflow.
- Stack Overflow
  - When the stack contains equal number of elements as per its capacity and no more elements can be added, the status of stack is known as stack overflow

Stack Implementation

- Implementation can be done in two ways
  - Static implementation
  - Dynamic Implementation
- Static Implementation
  - Stacks have **fixed size**, and are implemented as **arrays**
  - It is also inefficient for utilization of memory
- Dynamic Implementation
  - Stack **grow in size** as needed, and implemented as **linked lists**
  - Dynamic Implementation is done through pointers
  - The memory is efficiently utilize with Dynamic Implementations

Dynamic Implementation of Stacks

- As we know that dynamic stack is implemented using linked-list.
- In dynamic implementation stack can expand or shrink with each PUSH or POP operation.
- PUSH and POP operate only on the first/top cell on the list.



Dynamic Implementation of Stack

Class Definition

```
class ListStack{
   private:
                        struct node{
                        int num;
              node *next;
               }*top;
  public:
              ListStack(){ top=NULL;}
              void push();
              void pop();
              void display();
};
```

Push( ) Function

- This function creates a new node and ask the user to enter the data to be saved on the newly created node.

```
void ListStack::push()
{
   node *newNode;
   newNode= new node;
   cout<<“Enter number to add on stack";
   cin>> newNode->num;
   newNode->next=top;
   top=newNode;
}
```

Pop( ) Function

```
void ListStack::pop()
{
    node *temp;
    temp=top;
    if(top==NULL)
            cout<<"Stack UnderFlow"<<endl;
    else
    {
            cout<<"deleted Number from the stack =";
            cout<<top->num;
            top=top->next;
            delete temp;
    }
}
```

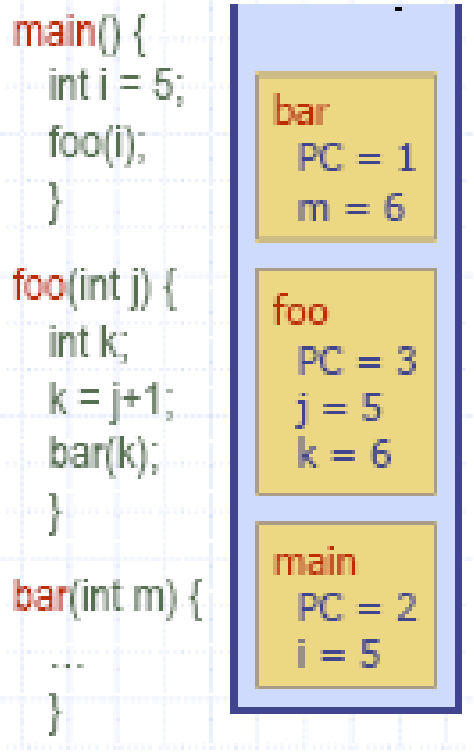
Main( ) Function

```
void main()
{
   clrscr();
  ListStack LS;
   int choice;
   do{
            cout<<"Menu "<<endl;
            cout<<"1.Push" <<endl;
            cout<<"2.Pop"<<endl;
            cout<<"3.Show"<<endl;
            cout<<"4.EXIT"<<endl;
            cin>>choice;
```

```
switch(choice){
   case 1:
            LS.push();
            break;
   case 2:
            LS.pop();
            break;
   case 3:
            LS.display();
            break;
   }
   }while(choice!=4);
}
```

C++ Run-time Stack

- The C++ run-time system keeps track of the chain of active functions with a stack
- When a function is called, the run-time system pushes on the stack a frame containing
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When a function returns, its frame is popped from the stack and control is passed to the method on top of the stack



Problem Solving with Stacks

Problem Solving with Stacks

- Many mathematical statements contain nested parenthesis like :-
  - (A+(B*C)) + (C – (D + F))
- We have to ensure that the parenthesis are nested correctly, i.e. :-
  1. There is an equal number of left and right parenthesis
  2. Every right parenthesis is preceded by a left parenthesis
- Expressions such as ((A + B) violate condition 1
- And expressions like) A + B (- C violate condition 2

Problem Solving (Cont….)

- To solve this problem, think of each left parenthesis as opening a scope, right parenthesis as closing a scope
- Nesting depth at a particular point in an expression is the number of scopes that have been opened but not yet closed
- Let “parenthesis count” be a variable containing number of left parenthesis minus number of right parenthesis, in scanning the expression from left to right

Problem Solving (Cont….)

- For an expression to be of a correct form following conditions apply
  - Parenthesis count at the end of an expression must be 0
  - Parenthesis count should always be non-negative while scanning an expression
- Example :
  - Expr: 7 – (A + B) + ((C – D) + F)
  - ParenthesisCount: 0 0 1 1 1 1 0 0 1 2 2 2 2 1 1 1 0
  - Expr: 7 – ((A + B) + ((C – D) + F)
  - ParenthesisCount: 0 0 1 2 2 2 2 1 1 23 3 3 3 2 2 21

Problem Solving (Cont….)

- Evaluating the correctness of simple expressions like this one can easily be done with the help of a few variables like “Parenthesis count”
- Things start getting difficult to handle by your program when the requirements get complicated e.g.
- Let us change the problem by introducing three different types of scope de-limiters i.e. (parenthesis), {braces} and [brackets].
- In such a situation we must keep track of not only the number of scope delimiters but also their types
- When a scope ender is encountered while scanning an expression, we must know the scope delimiter type with which the scope was opened
- We can use a stack ADT to solve this problem

Problem Solving with Stack

- A stack ADT can be used to keep track of the scope delimiters encountered while scanning the expression
- Whenever a scope “opener” is encountered, it can be “pushed” onto a stack
- Whenever a scope “ender” is encountered, the stack is examined:
  - If the stack is “empty”, there is no matching scope “opener” and the expression is invalid.
  - If the stack is not empty, we pop the stack and check if the “popped” item corresponds to the scope ender
  - If match occurs, we continue scanning the expression
- When end of the expression string is reached, the stack must be empty, otherwise one or more opened scopes have not been closed and the expression is invalid

Why the need for a Stack

- Last scope to be opened must be the first one to be closed.
- This scenario is simulated by a stack in which the last element arriving must be the first one to leave
- Each item on the stack represents a scope that has been opened but has yet not been closed
- Pushing an item on to the stack corresponds to opening a scope
- Popping an item from the stack corresponds to closing a scope, leaving one less scope open

Stack Applications

Converting Decimal to Binary: Consider the following pseudocode

```
Read (number)
Loop (number > 0)
                digit = number modulo 2
        print (digit)
        number = number / 2
// from Data Structures by Gilbert and Forouzan
```

The problem with this code is that it will print the binary number backwards. (ex: 19 becomes 11001000 instead of 00010011.) To remedy this problem, instead of printing the digit right away, we can push it onto the stack. Then after the number is done being converted, we pop the digit out of the stack and print it.

Practice Problem

- Write a program that gets an Infix arithmetic expression and converts it into postfix notation
- The program should then evaluate the postfix expression and output the result
- Your program should define the input and output format, enforce the format and handle Exceptions (exceptional conditions).
- Use appropriate comments at every stage of programming