University of Anbar
College of Computer Science and Information Technology
Computer Network Systems Department

Data Structures

Lecture Four

Second Stage

First Course - 2024-2025

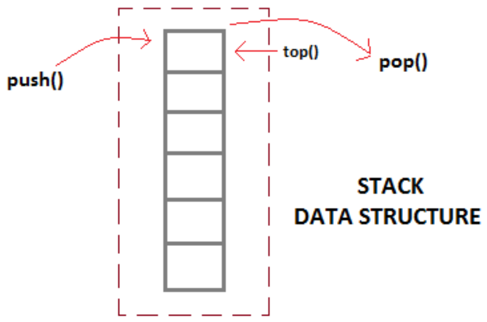
Muhmmad shihab muayad Shihab

Master of Computer Engineering

muhmmad.shihab@uoanbar.edu.iq

Stack

Stack is an abstract data type with a bounded(predefined) capacity. It is a simple data structure that allows adding and removing elements in a particular order. Every time an element is added, it goes on the top of the stack and the only element that can be removed is the element that is at the top of the stack, just like a pile of objects.



It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

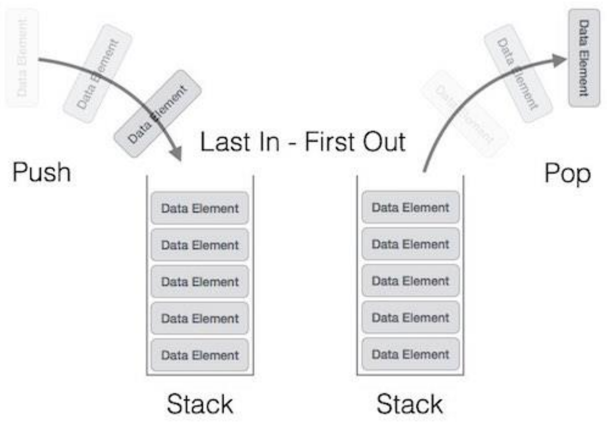
Basic features of Stack

1. Stack is an ordered list of similar data type.
2. Stack is a LIFO(Last in First out) structure or we can say FILO(First in Last out).
3. push() function is used to insert new elements into the Stack and pop() function is used to remove an element from the stack. Both insertion and removal are allowed at only one end of Stack called Top.

4. Stack is said to be in Overflow state when it is completely full and is said to be in Underflow state if it is completely empty.

Stack Representation

The following diagram depicts a stack and its operations:



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations:

- push() − Pushing (storing) an element on the stack.
- pop() − Removing (accessing) an element from the stack.

When data is **<u>PUSHed</u>** onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

- peek() − get the top data element of the stack, without removing it.
- isFull() − check if stack is full.
- isEmpty() − check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named top. The top pointer provides top value of the stack without actually removing it.

First we should learn about procedures to support stack functions:

- **peek()**

Implementation of peek() function in C++ programming language.

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

- **isfull()**

Implementation of isfull() function in C++ programming language.

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

- **isempty()**

Implementation of isempty() function in C++ programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty.

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

Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps:

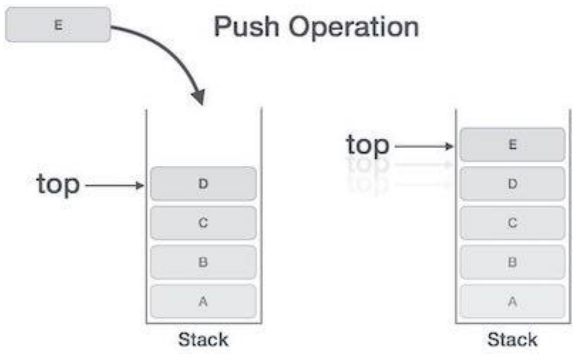
Step 1: Checks if the stack is full.

Step 2: If the stack is full, produces an error and exit.

Step 3: If the stack is not full, increments top to point next empty space.

Step 4: Adds data element to the stack location, where top is pointing.

Step 5: Returns success.



If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically. Implementation of this algorithm in C++, is very easy. See the following code:

```
void push(int data) {
   if(!isFull()) {
      top = top + 1;
      stack[top] = data;
   } else {
      printf("Could not insert data, Stack is full.\n");
   }
}
```

Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead top is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space. A Pop operation may involve the following steps:

Step 1: Checks if the stack is empty.

Step 2: If the stack is empty, produces an error and exit.

Step3: 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 success.

Stack Pop Operation

Implementation of this algorithm in C++, is as :

```
int pop(int data) {
   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   } else {
      printf("Could not retrieve data, Stack is empty.\n");
   }
}
```

Full stack program:

```
#include<iostream.h>
#define n 30
void push(char stack[n],int &top,char x);
void pop(char stack[n],int &top,char x);
void clear(int &top);
int full(int top);
int empty (int top);
void main()
{
char stack[n],x;
int top,c;
do{
cout<<"1: clear stack\n";
cout<<"2: push for the stack\n";
cout<<"3: pop from the stack\n";
cout<<"4: stack full or No\n";
cout<<"5: stack empty or No\n";
cout<<"6: Exist\n";
cout<<" inter your choice :";
cin>>c;
  switch(c)
           {
            case 1: clear(top);
                                  break;
            case 2: cout <<"enter the value : ";
                                 cin>>x;
```

```
                            push(stack,top,x);
                            cout<<"\n";   break;
            case 3: pop (stack,top,x);
                             cout<<"\n";
                              break;
            case 4: if(full(top)==1)
                                          cout<<" :::stack full\n";
                                    else cout<<"\n stack not full\n";
                                              break;
            case 5: if((empty (top)==1) )
                                 cout<<":::stack empty\n";
                              else cout<<"\n :::stack not empty \n";
                                             break;
            case 6: break;
                             }
       } while(c!=6);
       }
       void push(char stack[n],int&top,char x)
                 { if(top>=n-1) cout<<"stack over flow \n";
                      else
                            {
                            top++;
                            stack[top]=x;
                            }
                 }
       void pop(char stack[n],int&top,char x)
            {
             if (top==-1)
                            cout<<"stack is under flow\n";
                      else
                       {  x=stack[top];
                       top--;
           cout<<"\n the value pop is :"<<x;
                       }
            }
       void clear(int &top)
         {
        top=-1;
          }
```

```
int full(int top)
{
          if(top==n-1)
                    return 1;
           else
                     return 0;
}
int empty (int top)
{
if(top==-1)
                    return 1;
          else
                    return 0;
}
```

References:

- Frank Carrano, D.J. Henry: Data Abstraction and Solving with C++, 2012, 6th edition, Pearson Education, Inc.
- Mark Allen Weiss: Data Structures and Algorithm Analysis in C++, 2014, 4th edition, Pearson Education, Inc.