Stack ADT
Stack ADT

An ADT that manages data elements linearly but provides access at only one end i.e. data elements can be inserted and removed from only one end.

insert (Push)  remove (pop)

Dr Muhammad Hussain  Lecture - Stack ADT
NOTE: In Stack ADT, the elements are removed only in the inverse order of their arrival i.e. the last inserted element is removed first of all. So, it has last-in/first-out (LIFO) behavior.

Why Stack ADT?
There are many practical situations which involve stack
- The papers put in the tray of a printer is an ideal example of stack. The papers from the tray are removed by the printer in inverse order they are put in the tray.
- Stack is used to evaluate expressions like $10 \times (12 + 7)$
- A runtime system uses a stack to handle function calls and return.
### Specification

**Elements:** Any valid data type  
**Structure:** Any arrangement of elements that allows to remove first of all the element that is stored last of all  
**Domain:** Number of elements is bounded  

**Operations:**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Specification</th>
</tr>
</thead>
</table>
| `boolean empty()` | **Precondition/Requires:** none  
                      **Processing/Results:** returns true if the stack is empty otherwise false. |
| `boolean full()`  | **Precondition/Requires:** none  
                      **Processing/Results:** returns true if no more elements can be inserted into the stack otherwise false. |
| `void push(Type)` | **Precondition/Requires:** stack is not full  
                      **Processing/Results:** inserts a given element into the stack at the top and top points to it. |
| `Type pop()`      | **Precondition/Requires:** stack is not empty  
                      **Processing/Results:** the element at the top of the stack is removed from the stack and returned. |
Representation of Stack ADT

Stack ADT can be represented as
- Array
- Linked List

Array

Linked List

Null
Array based Implementation of Stack ADT

```java
public class ArrayStack<T> {
    // Data Members
    private int maxsize;
    private int top;
    private T[] nodes;
    // Operations
    public ArrayStack(int n)
    Public boolean empty()
    public boolean full()
    public void push(T e)
    public T pop()
}
```
Implementation of Operations

```java
public ArrayStack(int n) {
    maxsize = n;
top = 0;
nodes = (T[]) new Object[n];
}

public boolean empty() {
    return top == 0;
}
```
Implementation of Operations

```java
public boolean full()
{
    return top == maxsize;
}

public void push(T e)
{
    if(full()) return;
    nodes[top++] = e;
}

public T pop()
{
    if(!empty())
    {
        return nodes[--top];
    }
}
Linked List based Implementation of Stack ADT

```java
public class LinkStack<T> {
    // Data Members
    private Node<T> top;
    // Operations
    public LinkStack()
    public boolean empty()
    public boolean full()
    public void push(T e)
    public T pop()
}
```
Implementation of Operations

```java
public LinkStack()
{
    top = null;
}

public boolean empty()
{
    return top == null;
}

public boolean full()
{
    return false;
}
```
Implementation of Operations

public void push(T e)
{
    Node<T> tmp = new Node(e);
    tmp.next = top;
    top = tmp;
}

public T pop()
{
    Node<T> tmp = top;
    T e = top.data;
    top = top.next;
    return e;
}
Performance Comparison

- **Array Based Implementation**
  - An array based implementation suffers from the drawback that all the storage space must be reserved in advance and the maximum depth of the stack is limited to this array’s size.
  - Time complexity of all stack operations is $O(1)$.

- **Linked List based Implementation**
  - The time complexity of all operations is $O(1)$ except destructor, which takes $O(n)$ time.
  - For applications in which the maximum stack size is known ahead of time, an array is suitable.
  - If the maximum stack size is not known beforehand, we can use a linked list.
Application of Stack ADT

Infix Expressions
- An expression in which every binary operation appears between its operands
  Example: (i) $a+b$ – “+” is a binary operation and $a$ and $b$ are its operands
  (ii) $(a+b)*c$

Prefix Expressions
- An expression in which operator comes before its operands
  Example: (i) $a+b$ ⇒ +ab
  (ii) $(a+b)*c$ ⇒ *+abc
  (iii) $a+(b*c)$ ⇒ +a*bc

Postfix Expressions
- An expression in which operator comes after its operands
  Example: (i) $a+b$ ⇒ ab+
  (ii) $(a+b)*c$ ⇒ ab+c*
  (iii) $a+(b*c)$ ⇒ abc*+
NOTE

- Some calculators require that the input expression be in postfix form.
- Some computers accept expressions in infix form but convert them to postfix form before.

**Conversion of Infix form to Postfix Form using stack**

- Convert the infix expression a-((b+c)*d) into postfix form.
- It involves the following steps with a stack s:
  - write a → a
  - push(s, -)
  - push(s, ()
  - write b → ab
  - push(s, +)
  - write b → abc
  - After finding right parenthesis, pop(s, +) write it → abc+.
• pop(s, ()
• push(s, *)
• write d → abc+d
• pop(s, *) write it → abc+d*
• pop(s, -) write it → abc+d* -

**Exercise**

Write a function that converts an infix expression to its postfix form using a stack s and its operations given in the specification of stack ADT.