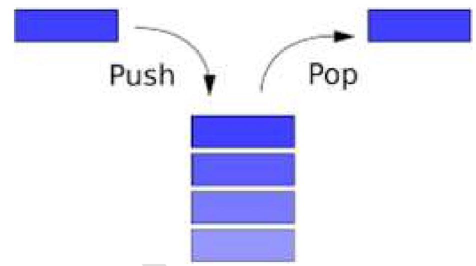
Stacks

Stack is an abstract data type that serves as a collection of elements, with two principal operations:

- **push** adds an element to the collection;
- **pop** removes the last element that was added.



- **Last In, First Out → LIFO**

ABSTRACT DATA TYPE: STACK		
DATA		
• A collection of objects in reverse chronological order and having the same data type		
OPERATIONS		
PSEUDOCODE	**UML**	**DESCRIPTION**
`push(newEntry)`	`+push(newEntry: T): void`	Task: Adds a new entry to the top of the stack. Input: `newEntry` is the new entry. Output: None.
`pop()`	`+pop(): T`	Task: Removes and returns the stack's top entry. Input: None. Output: Returns the stack's top entry. Throws an exception if the stack is empty before the operation.
`peek()`	`+peek(): T`	Task: Retrieves the stack's top entry without changing the stack in any way. Input: None. Output: Returns the stack's top entry. Throws an exception if the stack is empty.
`isEmpty()`	`+isEmpty(): boolean`	Task: Detects whether the stack is empty. Input: None. Output: Returns true if the stack is empty.
`clear()`	`+clear(): void`	Task: Removes all entries from the stack. Input: None. Output: None.

Stack Applications:

- Parsing in a compiler
- JVM
- Undo in a word processor
- Back button in a web browser
- Implementing function calls in a compiler
- :

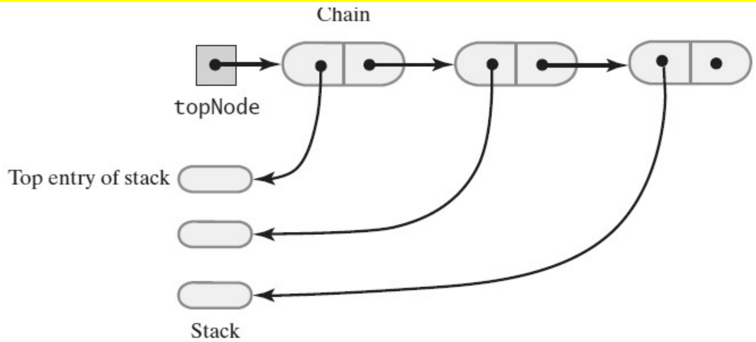
Single Linked List Implementation:

Each of the following operation involves top of stack

- **push**
- **pop**
- **peek**

Head or Tail for topNode??

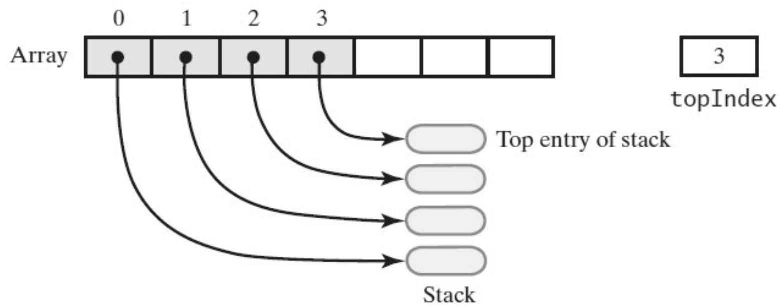
Head of linked list easiest, fastest to access O(c) ➔ Let this be the top of the stack




```java
public class LinkedStack<T extends Comparable<T>> {
   private Node<T> topNode;

   public void push(T data) {
      Node<T> newNode = new Node<T>(data);
      newNode.setNext(topNode);
      topNode = newNode;
   }

   public Node<T> pop() {
      Node<T> toDel = topNode;
      if(topNode != null)
         topNode = topNode.getNext();
      return toDel;
   }

   public Node<T> peek() {   return topNode; }

   public int length() {
      int length = 0;
      Node<T> curr = topNode;
      while (curr != null) {
         length++;
         curr = curr.getNext();
      }
      return length;
   }

   public boolean isEmpty() {    return (topNode == null);    }

   public  void clear() {  topNode = null;  }
}
```

Array-Based Implementation:

- End of the array easiest to access
    - Let this be top of stack
    - Let first entry be bottom of stack




```java
public class ArrayStack <T> {
   private Object[] s;
   private int n=-1;

   public ArrayStack(int capacity){
      s = new Object[capacity];
   }

   public boolean isEmpty(){ return n ==-1;}
   public int getN(){ return n;}

   public void push(T data){
      s[++n] = data;
   }

   public Object pop(){
      if(!isEmpty())
         return s[n--];
      return null;
   }

   public String toString() {
      String res = "Top-->";
      for(int i=n; i>=0;i--)
         res+="["+s[i]+"]-->";
      return res+"Null";
   }
}
```

- **Overflow**: use resizing array HW: To Do resize to double original size **(repeated doubling)**
- How to **shrink** array?? Half size of array when array is **¼ full**.

- Stack implementations: resizing array vs. linked list. **Which one is better??**
    - **Linked-list:** each operation takes O(c) in the worst case. However, uses extra time and space to deal with the links.
    - **Resizing array:** every operation takes O(c) **amortized**[1] also less wasted times.

[1] The basic idea is that an expensive operation can alter the state so that the worst case cannot occur again for a long **time**, thus **amortizing** its cost.