STACKS

CS212: Data Structure

Stacks

- A stack is a container of objects that are inserted and removed according to the **last-in-first-out (LIFO)** principle.

- Objects can be inserted at any time, but only the last (the most-recently inserted) object can be removed.
- Inserting an item is known as “Pushing” onto the stack. “Popping” off the stack is synonymous with removing an item
- Used in Operating system to implement method calls, and in evaluating Expressions.

ADT Stack: Specification

Elements: The elements are of a generic type <Type>. (In a linked implementation an element is placed in a node)

Structure: the elements are linearly arranged, and ordered according to the **order of arrival**, most recently arrived element is called top.

Domain: the number of elements in the stack is bounded therefore the domain is finite. Type of elements: Stack

ADT Stack: Specification

Operations:

All operations operate on a stack S.

1. **Method** push (Type e)
   requires: Stack S is not full.
   input: Type e.
   results: Element e is added to the stack as its most recently added elements.
   output: none.
2. **Method** pop (Type e)
   requires: Stack S is not empty.
   input: none
   results: the most recently arrived element in S is removed and its value assigned to e.
   output: Type e.
3. **Method** empty (boolean flag)
   input: none
   results: If Stack S is empty then flag is true, otherwise false.
   output: flag.

ADT Stack: Specification

Operations:

4. **Method** Full (boolean flag).

   requires:

   input: none

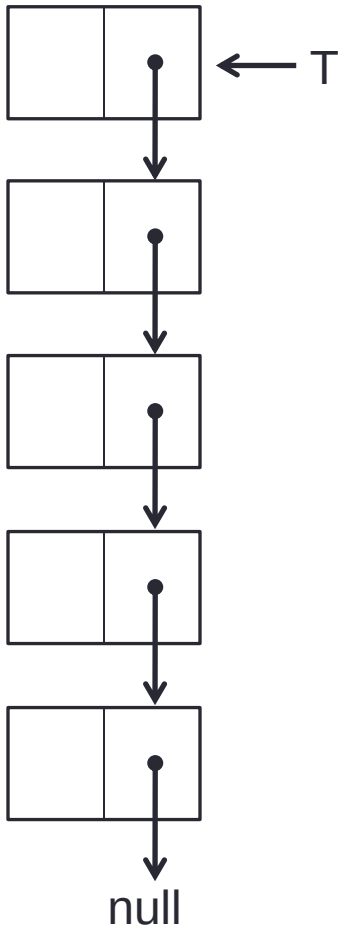
   results: If S is full then Full is true, otherwise Full is false.

   output: flag.

Stack Interface

```
public interface Stack<T>{
public T pop( );
public void push(T e);
public boolean empty( );
public boolean full( );
}
```

ADT Stack (Linked-List)



ADT Stack (Linked-List): Element

```
public class Node<T> {
  public T data;
  public Node<T> next;

  public Node () {
        data = null;
        next = null;
  }

  public Node (T val) {
        data = val;
        next = null;
  }

  // Setters/Getters?
}
```

ADT Stack (Linked-List): Implementation

```
public class LinkedStack<T> implements Stack<T> {
 private Node<T> top;

 /* Creates a new instance of LinkStack */
 public LinkedStack() {
      top = null;
 }
```

ADT Stack (Linked-List): Implementation

```
public class LinkedStack<T> implements Stack<L> {
 private Node<T> top;

 /* Creates a new instance of LinkStack */
 public LinkedStack() {
      top = null;
 }
```

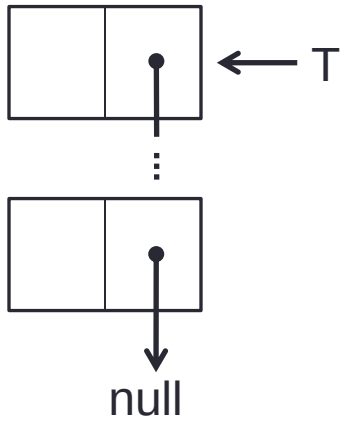
ADT Stack (Linked-List): Implementation

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

public boolean full(){
     return false;
}
```

ADT Stack (Linked-List): Implementation

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

public boolean full(){
     return false;
}
```

false



true

ADT Stack (Linked-List): Implementation

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

ADT Stack (Linked-List): Implementation

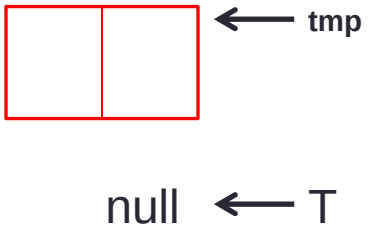
```
public void push(T e){
      Node<T> tmp = new Node<T>(e);
      tmp.next = top;
      top = tmp;
}
```

null ⟵ T

Example #1

ADT Stack (Linked-List): Implementation

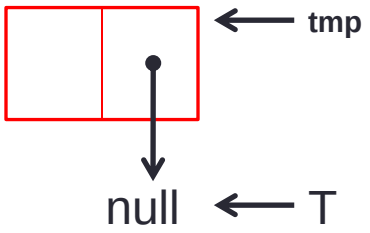
```
public void push(T e){
      Node<T> tmp = new Node<T>(e);
      tmp.next = top;
      top = tmp;
}
```



Example #1

ADT Stack (Linked-List): Implementation

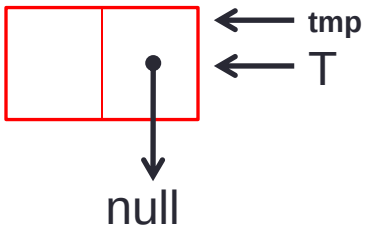
```
public void push(T e){
        Node<T> tmp = new Node<T>(e);
        tmp.next = top;
        top = tmp;
}
```



Example #1

ADT Stack (Linked-List): Implementation

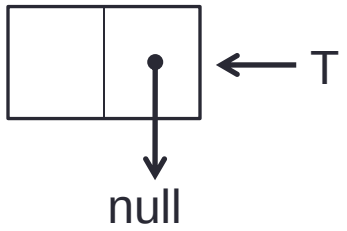
```
public void push(T e){
      Node<T> tmp = new Node<T>(e);
      tmp.next = top;
      top = tmp;
}
```



Example #1

ADT Stack (Linked-List): Implementation

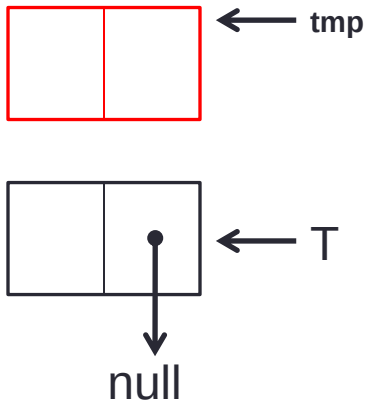
```
public void push(T e){
      Node<T> tmp = new Node<T>(e);
      tmp.next = top;
      top = tmp;
}
```



Example #2

ADT Stack (Linked-List): Implementation

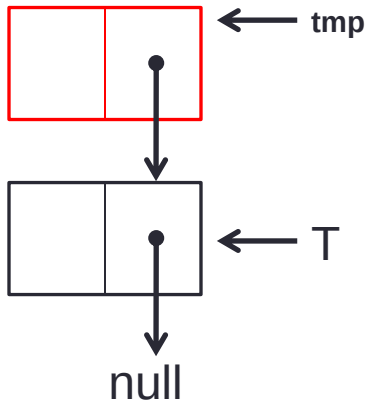
```
public void push(T e){
      Node<T> tmp = new Node<T>(e);
      tmp.next = top;
      top = tmp;
}
```



Example #2

ADT Stack (Linked-List): Implementation

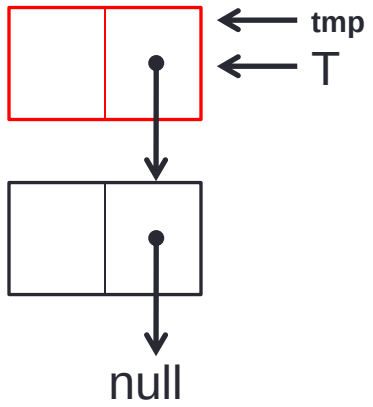
```
public void push(T e){
        Node<T> tmp = new Node<T>(e);
        tmp.next = top;
        top = tmp;
}
```



Example #2

ADT Stack (Linked-List): Implementation

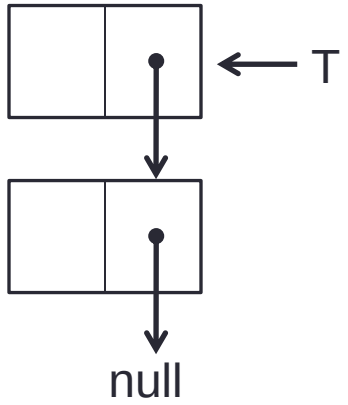
```
public void push(T e){
      Node<T> tmp = new Node<T>(e);
      tmp.next = top;
      top = tmp;
}
```



Example #2

ADT Stack (Linked-List): Implementation

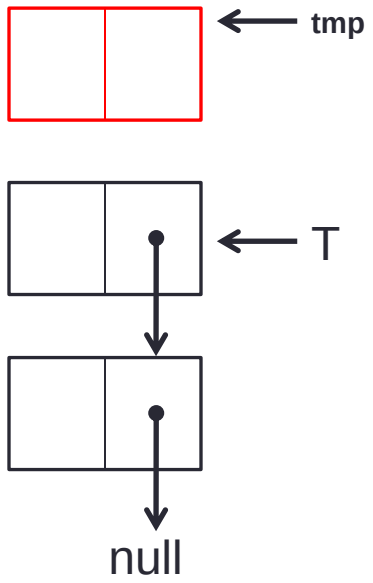
```
public void push(T e){
      Node<T> tmp = new Node<T>(e);
      tmp.next = top;
      top = tmp;
}
```



Example #3

ADT Stack (Linked-List): Implementation

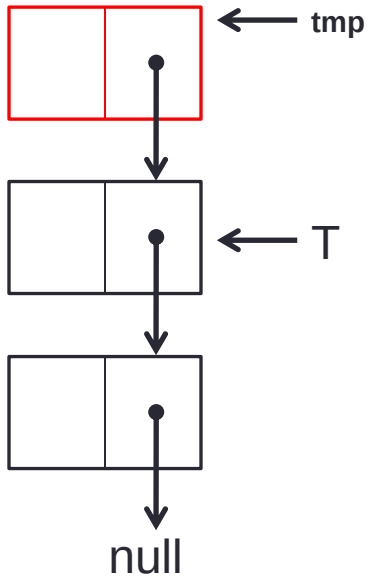
```
public void push(T e){
      Node<T> tmp = new Node<T>(e);
      tmp.next = top;
      top = tmp;
}
```



Example #3

ADT Stack (Linked-List): Implementation

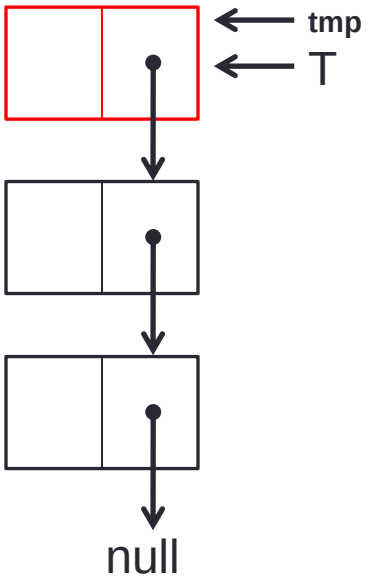
```
public void push(T e){
        Node<T> tmp = new Node<T>(e);
        tmp.next = top;
        top = tmp;
}
```



Example #3

ADT Stack (Linked-List): Implementation

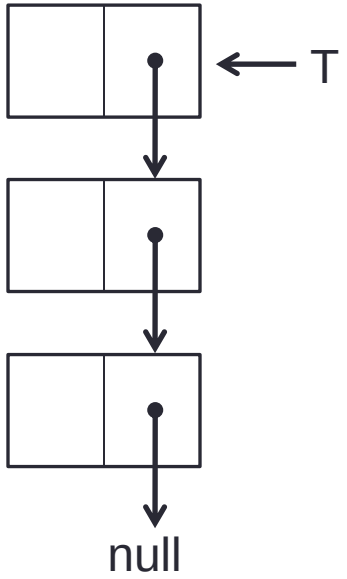
```
public void push(T e){
      Node<T> tmp = new Node<T>(e);
      tmp.next = top;
      top = tmp;
}
```



Example #3

ADT Stack (Linked-List): Implementation

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



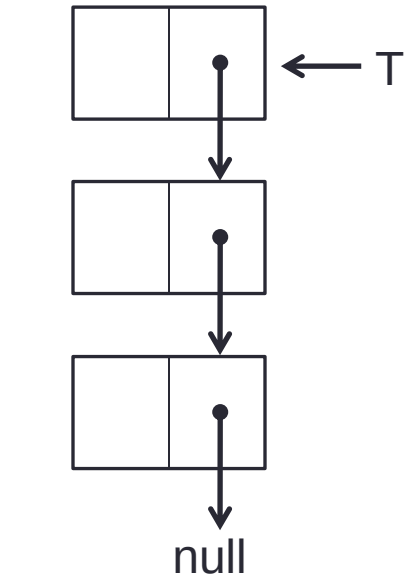
Example #3

ADT Stack (Linked-List): Implementation

```
 public T pop(){
      T e = top.data;
      top = top.next;
      return e;
 }
}
```

ADT Stack (Linked-List): Implementation

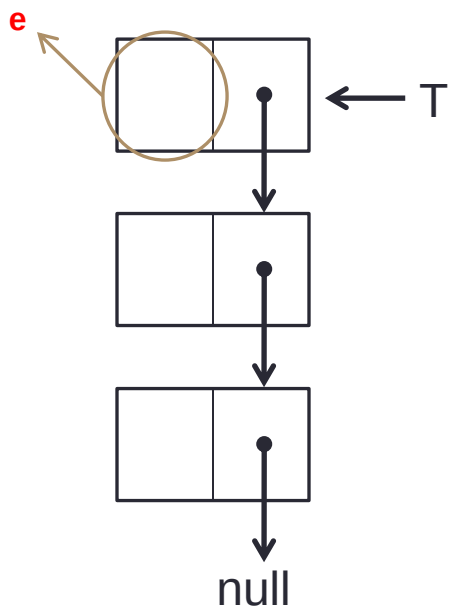
```
 public T pop(){
      T e = top.data;
      top = top.next;
      return e;
  }
}
```



Example #1

ADT Stack (Linked-List): Implementation

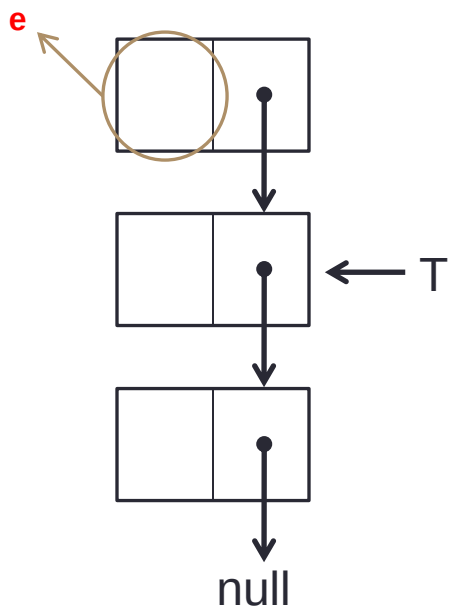
```
 public T pop(){
        T e = top.data;
        top = top.next;
        return e;
  }
}
```



Example #1

ADT Stack (Linked-List): Implementation

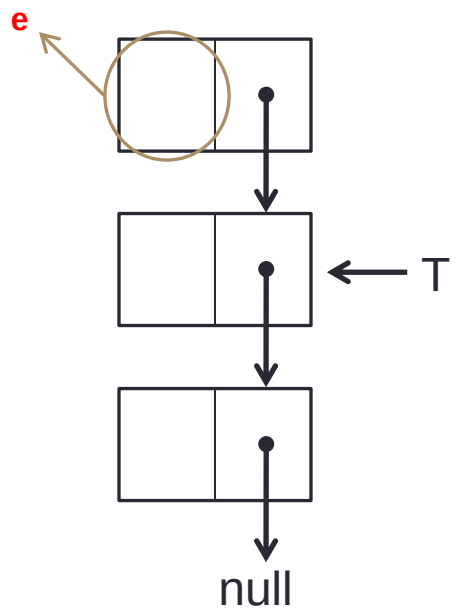
```
 public T pop(){
        T e = top.data;
        top = top.next;
        return e;
  }
}
```



Example #1

ADT Stack (Linked-List): Implementation

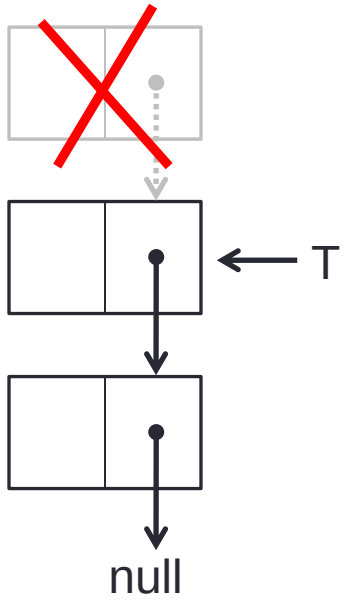
```
 public T pop(){
        T e = top.data;
        top = top.next;
        return e;
  }
}
```



Example #1

ADT Stack (Linked-List): Implementation

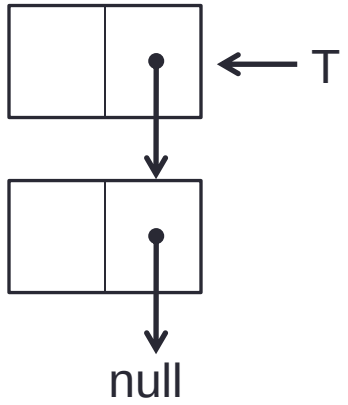
```
 public T pop(){
      T e = top.data;
      top = top.next;
      return e;
  }
}
```



Example #1

ADT Stack (Linked-List): Implementation

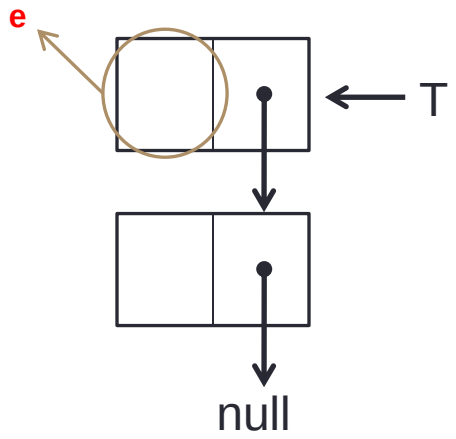
```
  public T pop(){
        T e = top.data;
        top = top.next;
        return e;
  }
}
```



Example #2

ADT Stack (Linked-List): Implementation

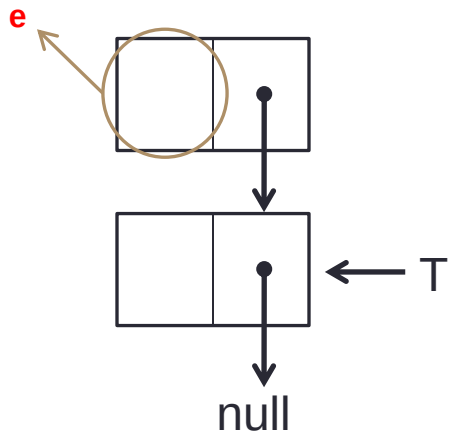
```
 public T pop(){
      T e = top.data;
      top = top.next;
      return e;
  }
}
```



Example #2

ADT Stack (Linked-List): Implementation

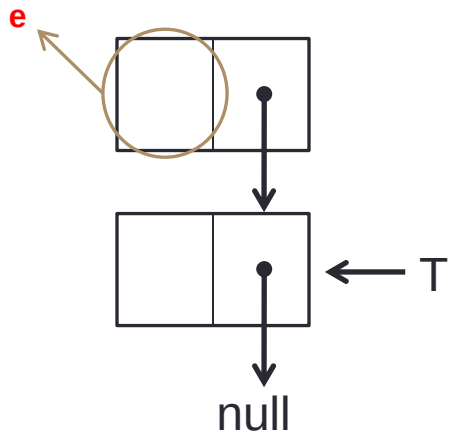
```
  public T pop(){
        T e = top.data;
        top = top.next;
        return e;
  }
}
```



Example #2

ADT Stack (Linked-List): Implementation

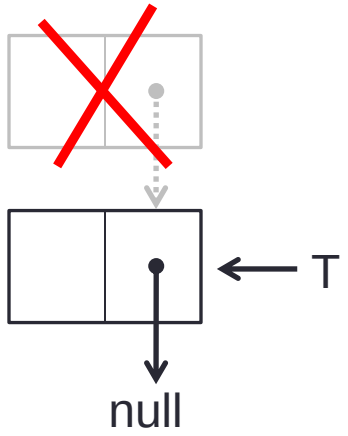
```
  public T pop(){
        T e = top.data;
        top = top.next;
        return e;
  }
}
```



Example #2

ADT Stack (Linked-List): Implementation

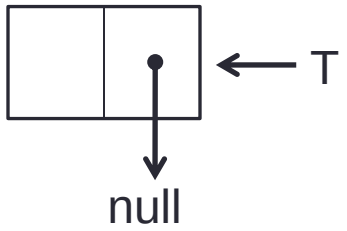
```
  public T pop(){
        T e = top.data;
        top = top.next;
        return e;
  }
}
```



Example #2

ADT Stack (Linked-List): Implementation

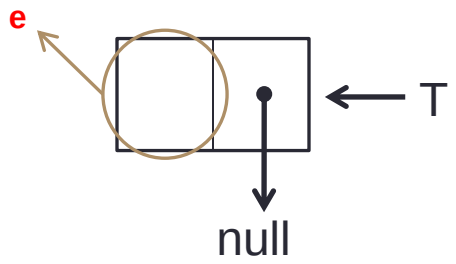
```
 public T pop(){
      T e = top.data;
      top = top.next;
      return e;
  }
}
```



Example #3

ADT Stack (Linked-List): Implementation

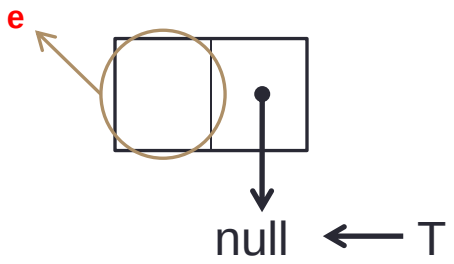
```
 public T pop(){
      T e = top.data;
      top = top.next;
      return e;
  }
}
```



Example #3

ADT Stack (Linked-List): Implementation

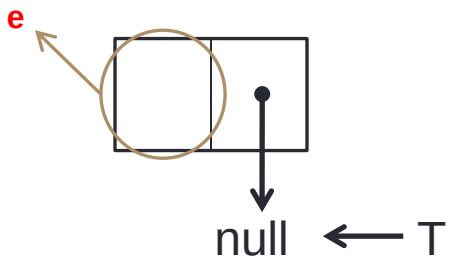
```
  public T pop(){
        T e = top.data;
        top = top.next;
        return e;
  }
}
```



Example #3

ADT Stack (Linked-List): Implementation

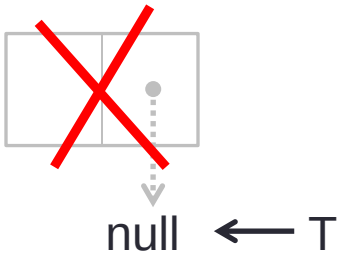
```
 public T pop(){
        T e = top.data;
        top = top.next;
        return e;
  }
}
```



Example #3

ADT Stack (Linked-List): Implementation

```
 public T pop(){
        T e = top.data;
        top = top.next;
        return e;
  }
}
```



Example #3

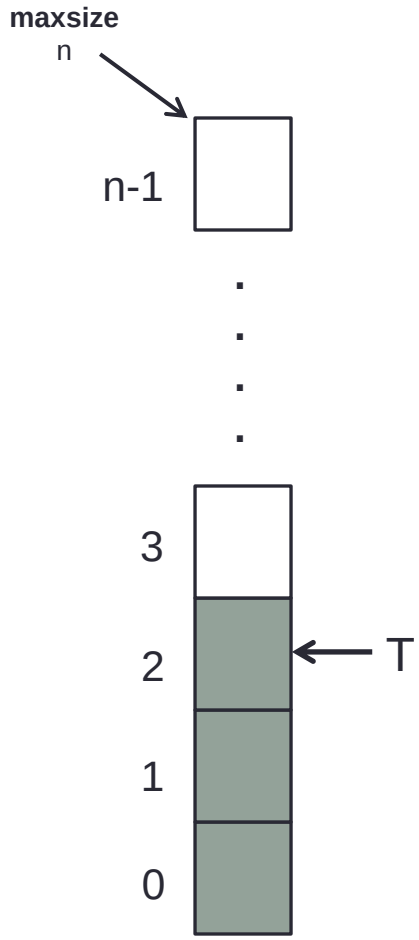
ADT Stack (Linked-List): Implementation

```
 public T pop(){
      T e = top.data;
      top = top.next;
      return e;
  }
}
```

null ⟵ T

Example #3

ADT Stack (Array)

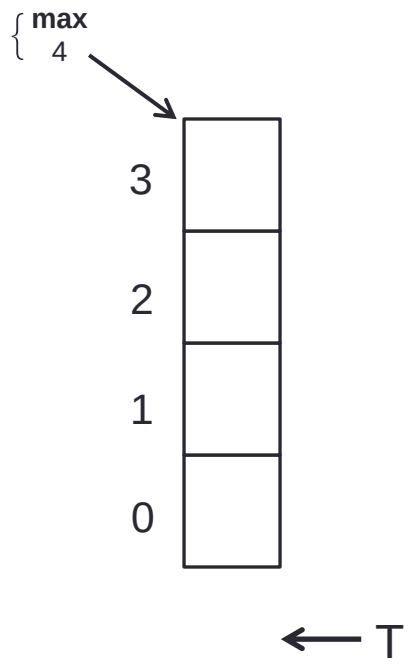


ADT Stack (Array): Representation

```
public class ArrayStack<T> implements Stack<L>  {
  private int maxsize;
  private int top;
  private T[] nodes;

  /** Creates a new instance of ArrayStack */
  public ArrayStack(int n) {
        maxsize = n;
        top = -1;
        nodes = (T[]) new Object[n];
  }
```

ADT Stack (Array): Representation

```
public class ArrayStack<T> implements Stack<L>  {
  private int maxsize;
  private int top;
  private T[] nodes;

  /** Creates a new instance of ArrayStack */
  public ArrayStack(int n) {
        maxsize = n;
        top = -1;
        nodes = (T[]) new Object[n];
  }
```

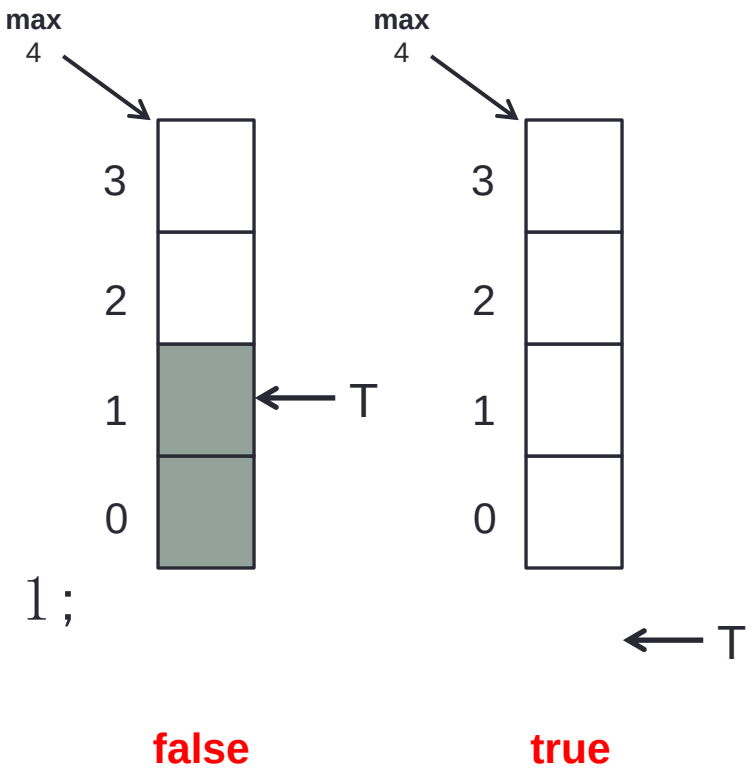


ADT Stack (Array): Implementation

```
public boolean empty(){
     return top == -1;
}

public boolean full(){
     return top == maxsize - 1;
}
```

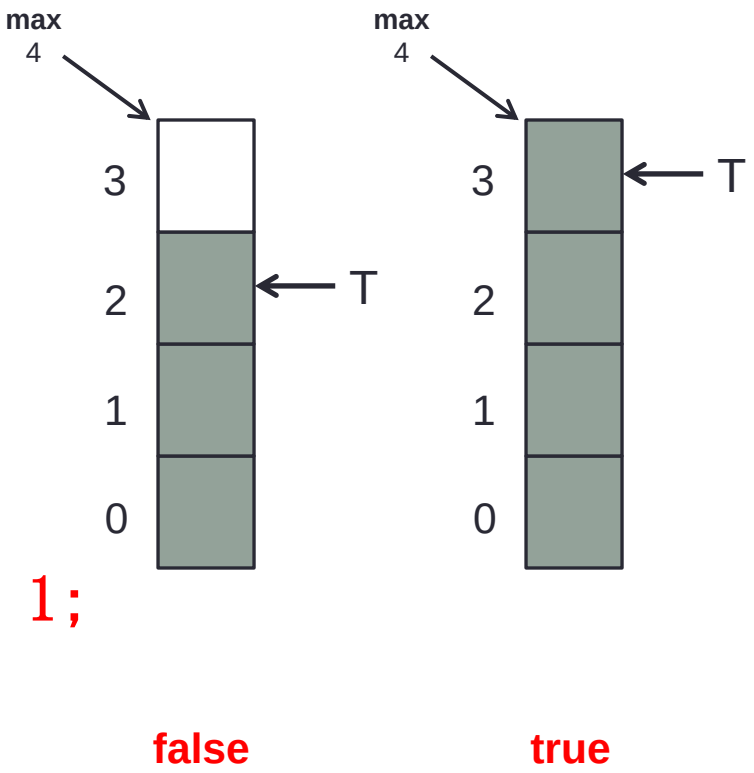
ADT Stack (Array): Implementation

```
public boolean empty(){
      return top == -1;
}

public boolean full(){
      return top == maxsize - 1;
}
```



ADT Stack (Array): Implementation

```
public boolean empty(){
      return top == -1;
}

public boolean full(){
      return top == maxsize - 1;
}
```

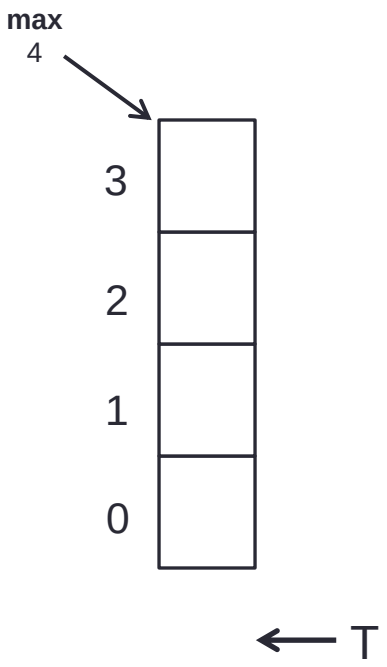


ADT Stack (Array): Implementation

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

  public T pop(){
        return nodes[top--];
  }
}
```

ADT Stack (Array): Implementation

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

  public T pop(){
        return nodes[top--];
  }
}
```

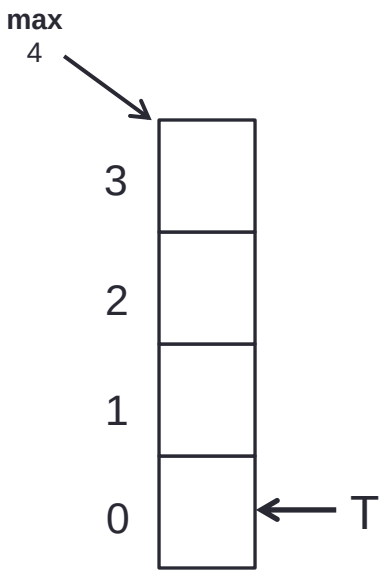


Example #1

ADT Stack (Array): Implementation

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

public T pop(){
        return nodes[top--];
}
}
```

S1 ← ++top
nodes[S1] = e

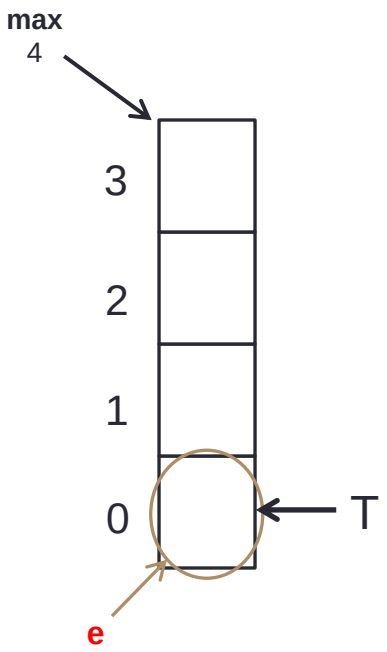


Example #1

ADT Stack (Array): Implementation

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

  public T pop(){
        return nodes[top--];
  }
}
```

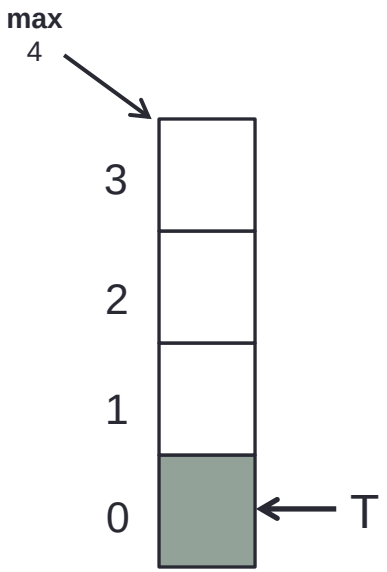
S1 ← ++top
nodes[S1] = e



Example #1

ADT Stack (Array): Implementation

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

  public T pop(){
        return nodes[top--];
  }
}
```

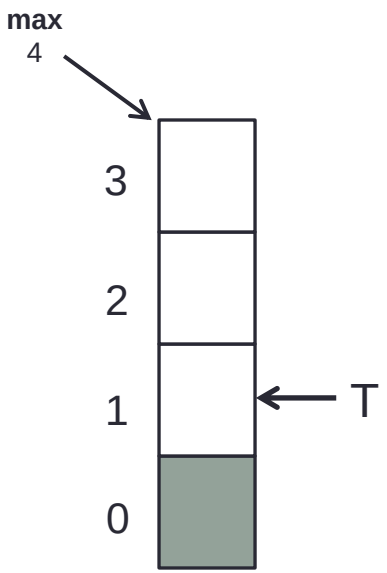


Example #2

ADT Stack (Array): Implementation

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

public T pop(){
        return nodes[top--];
}
}
```

S1 ← ++top
nodes[S1] = e

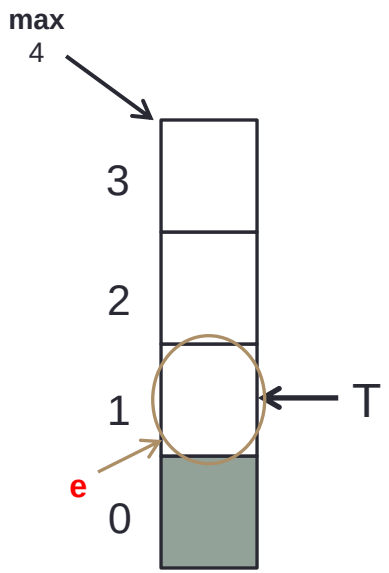


Example #2

ADT Stack (Array): Implementation

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

  public T pop(){
        return nodes[top--];
  }
}
```

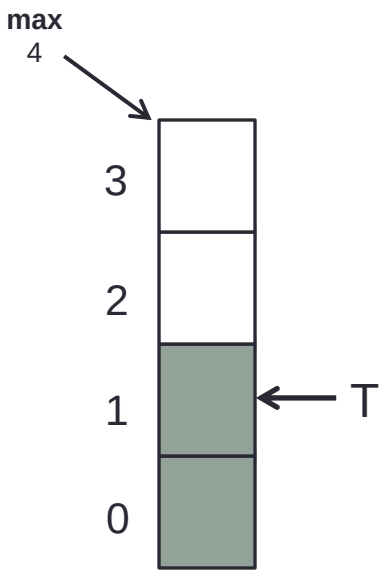
S1 ← ++top
nodes[S1] = e



Example #2

ADT Stack (Array): Implementation

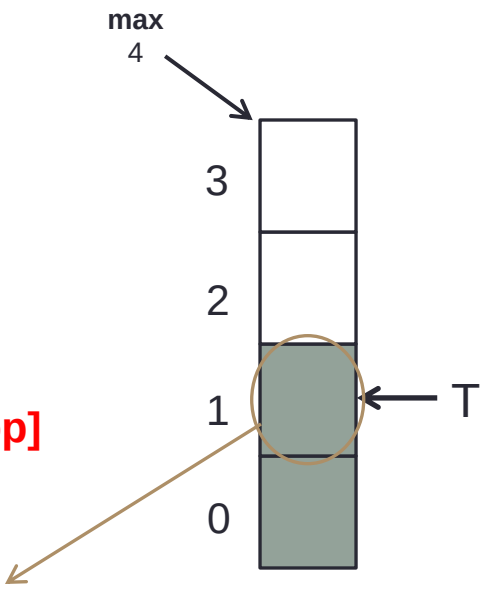
```
  public void push(T e){
        nodes[++top] = e;
  }

  public T pop(){
        return nodes[top--];
  }
}
```



Example #3

ADT Stack (Array): Implementation

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

  public T pop(){
        return nodes[top--];
  }
}
```

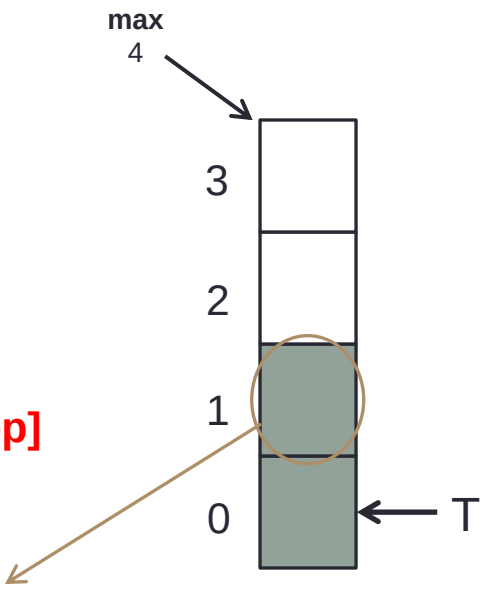


S1 ← nodes[top]
top--
return S1

Example #3

ADT Stack (Array): Implementation

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

  public T pop(){
        return nodes[top--];
  }
}
```

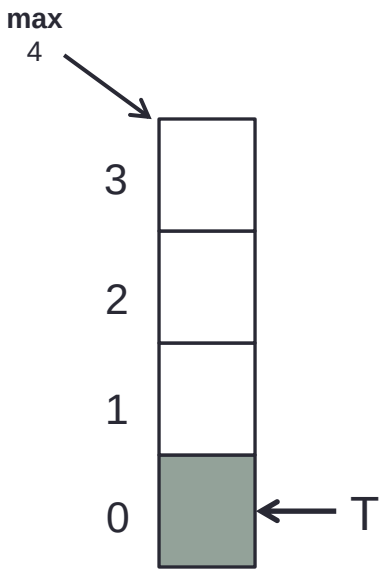


S1 ← nodes[top]
top--
return S1

Example #3

ADT Stack (Array): Implementation

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

  public T pop(){
        return nodes[top--];
  }
}
```

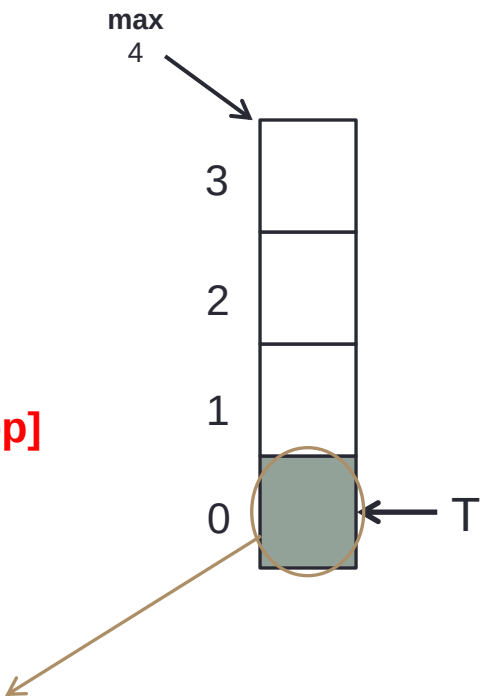


Example #4

ADT Stack (Array): Implementation

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

  public T pop(){
        return nodes[top--];
  }
}
```

S1 ← nodes[top]
top--
return S1

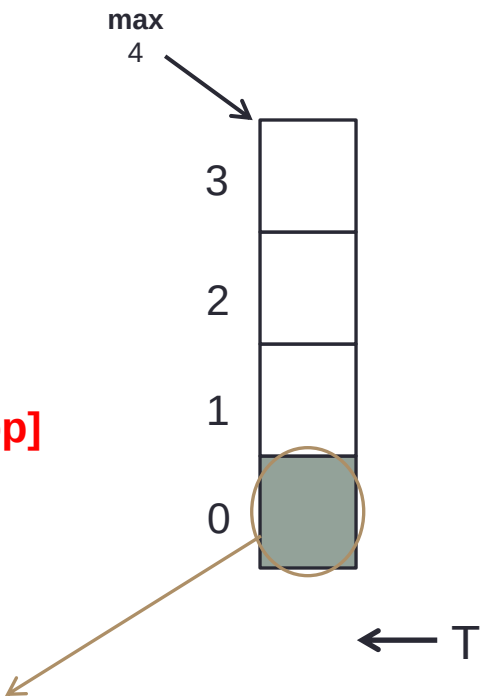


Example #4

ADT Stack (Array): Implementation

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

public T pop(){
        return nodes[top--];
}
}
```

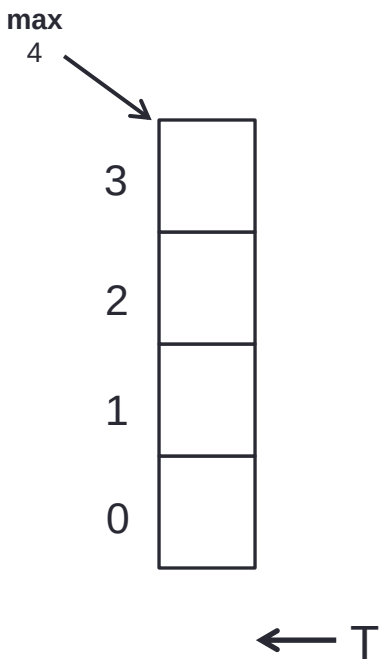
S1 ← nodes[top]
top--
return S1



Example #4

ADT Stack (Array): Implementation

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

  public T pop(){
        return nodes[top--];
  }
}
```



Example #4

Applications of Stacks

- Some applications of stacks are:
  - Balancing symbols.
  - Computing or evaluating postfix expressions.
  - Converting expressions from infix to postfix.

1. Balancing Symbols

- Expressions: mathematical (a + ((b-c)*d)) or programs have **delimiters**.

```
begin                          {
        S1                             S1
        S2                             {
        begin                                  S2
                S3                             S3
                begin                  }
                ....                   S4
                end            }
        end
end
```

1. Balancing Symbols

- Delimiters must be balanced.
- One of the common use of the stacks is to parse certain kinds of expressions or string text.
- Write a program that verifies the delimiters in a line of text or expression typed by the user.
  - a*(b+c) //This expression is right
  - b/[a*(b+c)] //This expression is right
  - {a*(b+c]} //This expression is wrong

1. Balancing Symbols

- Read characters from the start of the expression to the end.
  - If the token is a starting delimiter, then push on to the stack.
  - If the token is a closing delimiter, then pop from the stack.
    - If symbol from this pop operation matches the closing delimiter, then we carry on.
    - If not, or the stack was empty, then we have unbalanced symbols (report an error).
- If stack is empty at the end of expression, we have balanced symbols.
- If not (stack is not empty), then we have unbalanced symbols (report an error).

1. Balancing Symbols

- **Input** : expression
- **Output**: True if and only if delimiters are balanced
- Let S be empty Stack
- Let n be number of characters
- for i=0→ n-1
  - If expression[i] is a **Opening delimiter**, then
    - S.push(expression[i]).
  - else If expression[i] is a **closing delimiter,** then
    - **If** the S is empty
      - return false unbalanced symbols
    - symbol=S.pop().
    - If symbol does not matches the closing delimiter
      - return false unbalanced symbols.
- If S is empty
  - return true balanced symbols.
- else
  - return false unbalanced symbols

2. Postfix Expressions

- Evaluating Postfix Expressions:
  - Infix expression: 4.99*1.06+5.99+6.99*1.06
  - Value 18.69 correct← parenthesis used.
  - Value 19.37 incorrect ← no parenthesis used.
  - In postfix form, above expression becomes:

    4.99 1.06 * 5.99 + 6.99 1.06*+

  →Advantage: no brackets are needed and a stack can be used to compute the expression.

2. Postfix Expressions

- Example:
  - infix: 6*(5+((2+3)*8)+3)
  - postfix: 6 5 2 3 + 8 * + 3 + *.
- Algorithm to compute postfix expression:
  - Read the postfix expression left to right.
  - When a number is read push it on the stack.
  - When an operator is read:
    - pop two numbers from the stack
    - carry out the operation on them
    - push the result back on the stack.

2. Postfix Expressions

- Example:
  - infix: 6*(5+((2+3)*8)+3)
  - postfix: 6 5 2 3 + 8 * + 3 + *.
- Algorithm to compute postfix expression:
  - Read the postfix expression left to right.
  - When a number is read push it on the stack.
  - When an operator is read:
    - pop two numbers from the stack
    - carry out the operation on them
    - push the result back on the stack.

2. Postfix Expressions

- Example:
  - infix: 6*(5+((2+3)*8)+3)
  - postfix: 6 5 2 3 + 8 * + 3 + *.
- Algorithm to compute postfix expression:
  - Read the postfix expression left to right.
  - When a number is read push it on the stack.
  - When an operator is read:
    - pop two numbers from the stack
    - carry out the operation on them
    - push the result back on the stack.

2. Postfix Expressions

- Example:
  - infix: 6*(5+((2+3)*8)+3)
  - postfix: 6 5 2 3 + 8 * + 3 + *.
- Algorithm to compute postfix expression:
  - Read the postfix expression left to right.
  - When a number is read push it on the stack.
  - When an operator is read:
    - pop two numbers from the stack
    - carry out the operation on them
    - push the result back on the stack.

2. Postfix Expressions

- Example:
  - infix: 6*(5+((2+3)*8)+3)
  - postfix: 6 5 2 3 + 8 * + 3 + *.
- Algorithm to compute postfix expression:
  - Read the postfix expression left to right.
  - When a number is read push it on the stack.
  - When an operator is read:
    - pop two numbers from the stack
    - carry out the operation on them
    - push the result back on the stack.

2. Postfix Expressions

- Example:
  - infix: 6*(5+((2+3)*8)+3)
  - postfix: 6 5 2 3 + 8 * + 3 + *.
- Algorithm to compute postfix expression:
  - Read the postfix expression left to right.
  - When a number is read push it on the stack.
  - When an operator is read:
    - pop two numbers from the stack
    - carry out the operation on them
    - push the result back on the stack.

2. Postfix Expressions

- Example:
  - infix: 6*(5+((2+3)*8)+3)
  - postfix: 6 5 2 3 + 8 * + 3 + *.
- Algorithm to compute postfix expression:
  - Read the postfix expression left to right.
  - When a number is read push it on the stack.
  - When an operator is read:
    - pop two numbers from the stack
    - carry out the operation on them
    - push the result back on the stack.

2. Postfix Expressions

- Example:
  - infix: 6*(5+((2+3)*8)+3)
  - postfix: 6 5 2 3 + 8 * + 3 + *.
- Algorithm to compute postfix expression:
  - Read the postfix expression left to right.
  - When a number is read push it on the stack.
  - When an operator is read:
    - pop two numbers from the stack
    - carry out the operation on them
    - push the result back on the stack.

2 + 3 = 5

2. Postfix Expressions

- Example:
  - infix: 6*(5+((2+3)*8)+3)
  - postfix: 6 5 2 3 + 8 * + 3 + *.
- Algorithm to compute postfix expression:
  - Read the postfix expression left to right.
  - When a number is read push it on the stack.
  - When an operator is read:
    - pop two numbers from the stack
    - carry out the operation on them
    - push the result back on the stack.

2 + 3 = 5

2. Postfix Expressions

- Example:
  - infix: 6*(5+((2+3)*8)+3)
  - postfix: 6 5 2 3 + 8 * + 3 + *.
- Algorithm to compute postfix expression:
  - Read the postfix expression left to right.
  - When a number is read push it on the stack.
  - When an operator is read:
    - pop two numbers from the stack
    - carry out the operation on them
    - push the result back on the stack.

2. Postfix Expressions

- Example:
  - infix: 6*(5+((2+3)*8)+3)
  - postfix: 6 5 2 3 + 8 * + 3 + *.
- Algorithm to compute postfix expression:
  - Read the postfix expression left to right.
  - When a number is read push it on the stack.
  - When an operator is read:
    - pop two numbers from the stack
    - carry out the operation on them
    - push the result back on the stack.

2. Postfix Expressions

- Example:
  - infix: 6*(5+((2+3)*8)+3)
  - postfix: 6 5 2 3 + 8 * + 3 + *.
- Algorithm to compute postfix expression:
  - Read the postfix expression left to right.
  - When a number is read push it on the stack.
  - When an operator is read:
    - pop two numbers from the stack
    - carry out the operation on them
    - push the result back on the stack.

5 * 8 = 40

2. Postfix Expressions

- Example:
  - infix: 6*(5+((2+3)*8)+3)
  - postfix: 6 5 2 3 + 8 * + 3 + *.
- Algorithm to compute postfix expression:
  - Read the postfix expression left to right.
  - When a number is read push it on the stack.
  - When an operator is read:
    - pop two numbers from the stack
    - carry out the operation on them
    - push the result back on the stack.

5 * 8 = 40

2. Postfix Expressions

- Example:
  - infix: 6*(5+((2+3)*8)+3)
  - postfix: 6 5 2 3 + 8 * + 3 + *.
- Algorithm to compute postfix expression:
  - Read the postfix expression left to right.
  - When a number is read push it on the stack.
  - When an operator is read:
    - pop two numbers from the stack
    - carry out the operation on them
    - push the result back on the stack.

2. Postfix Expressions

- Example:
  - infix: 6*(5+((2+3)*8)+3)
  - postfix: 6 5 2 3 + 8 * + 3 + *.
- Algorithm to compute postfix expression:
  - Read the postfix expression left to right.
  - When a number is read push it on the stack.
  - When an operator is read:
    - pop two numbers from the stack
    - carry out the operation on them
    - push the result back on the stack.

5 + 40 = 45

2. Postfix Expressions

- Example:
  - infix: 6*(5+((2+3)*8)+3)
  - postfix: 6 5 2 3 + 8 * + 3 + *.
- Algorithm to compute postfix expression:
  - Read the postfix expression left to right.
  - When a number is read push it on the stack.
  - When an operator is read:
    - pop two numbers from the stack
    - carry out the operation on them
    - push the result back on the stack.

5 + 40 = 45

2. Postfix Expressions

- Example:
  - infix: 6*(5+((2+3)*8)+3)
  - postfix: 6 5 2 3 + 8 * + 3 + *.
- Algorithm to compute postfix expression:
  - Read the postfix expression left to right.
  - When a number is read push it on the stack.
  - When an operator is read:
    - pop two numbers from the stack
    - carry out the operation on them
    - push the result back on the stack.

2. Postfix Expressions

- Example:
  - infix: 6*(5+((2+3)*8)+3)
  - postfix: 6 5 2 3 + 8 * + 3 + *.
- Algorithm to compute postfix expression:
  - Read the postfix expression left to right.
  - When a number is read push it on the stack.
  - When an operator is read:
    - pop two numbers from the stack
    - carry out the operation on them
    - push the result back on the stack.

2. Postfix Expressions

- Example:
  - infix: 6*(5+((2+3)*8)+3)
  - postfix: 6 5 2 3 + 8 * + 3 + *.
- Algorithm to compute postfix expression:
  - Read the postfix expression left to right.
  - When a number is read push it on the stack.
  - When an operator is read:
    - pop two numbers from the stack
    - carry out the operation on them
    - push the result back on the stack.

45 + 3 = 48

2. Postfix Expressions

- Example:
  - infix: 6*(5+((2+3)*8)+3)
  - postfix: 6 5 2 3 + 8 * + 3 + *.
- Algorithm to compute postfix expression:
  - Read the postfix expression left to right.
  - When a number is read push it on the stack.
  - When an operator is read:
    - pop two numbers from the stack
    - carry out the operation on them
    - push the result back on the stack.

45 + 3 = 48

2. Postfix Expressions

- Example:
  - infix: 6*(5+((2+3)*8)+3)
  - postfix: 6 5 2 3 + 8 * + 3 + *.
- Algorithm to compute postfix expression:
  - Read the postfix expression left to right.
  - When a number is read push it on the stack.
  - When an operator is read:
    - pop two numbers from the stack
    - carry out the operation on them
    - push the result back on the stack.

2. Postfix Expressions

- Example:
  - infix: 6*(5+((2+3)*8)+3)
  - postfix: 6 5 2 3 + 8 * + 3 + *.
- Algorithm to compute postfix expression:
  - Read the postfix expression left to right.
  - When a number is read push it on the stack.
  - When an operator is read:
    - pop two numbers from the stack
    - carry out the operation on them
    - push the result back on the stack.

6 * 48 = 288

2. Postfix Expressions

- Example:
  - infix: 6*(5+((2+3)*8)+3)
  - postfix: 6 5 2 3 + 8 * + 3 + *.
- Algorithm to compute postfix expression:
  - Read the postfix expression left to right.
  - When a number is read push it on the stack.
  - When an operator is read:
    - pop two numbers from the stack
    - carry out the operation on them
    - push the result back on the stack.

6 * 48 = 288

2. Postfix Expressions

- Example:
  - infix: 6*(5+((2+3)*8)+3)
  - postfix: 6 5 2 3 + 8 * + 3 + *.
- Algorithm to compute postfix expression:
  - Read the postfix expression left to right.
  - When a number is read push it on the stack.
  - When an operator is read:
    - pop two numbers from the stack
    - carry out the operation on them
    - push the result back on the stack.

End!

result

3. Converting from infix to postfix

Assume the infix expression is a string of tokens delimited by spaces. The operator tokens are *, /, +, and -, along with the left and right parentheses, (with). The operand tokens are the single-character identifiers A, B, C, and so on.
The following steps will produce a string of tokens in postfix order.

1. Create an **empty stack** called opstack for keeping operators. Create an **empty list** for output.
2. Scan the token list from left to right.
   - If the token is an operand, **append** it to the end of the **output list**.
   - If the token is a left parenthesis, **push** it on the **opstack**.
   - If the token is a right parenthesis, **pop** the **opstack** until the corresponding left parenthesis is removed. Append each operator to the end of the output list.
   - If the token is an operator, *, /, +, or -, push it on the opstack. However, first remove any operators already on the opstack that have higher or equal precedence and **append** them to the **output list**.
3. When the input expression has been completely processed, check the opstack. Any operators still on the stack can be removed and appended to the end of the output list.

STACK OPERATIONS

CS212: Data Structure

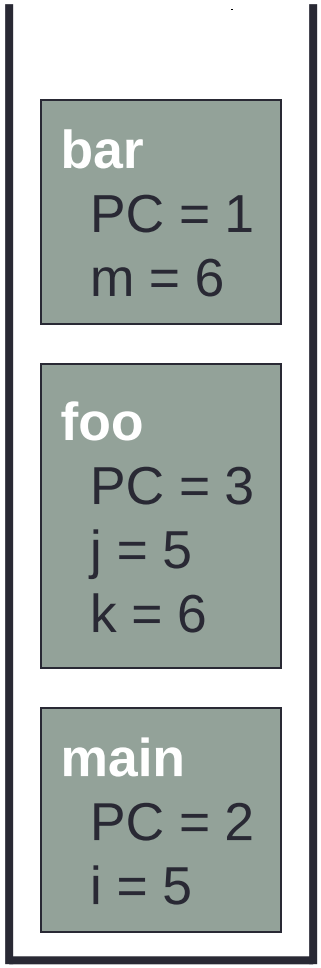
Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Method Stack in the JVM

- The Java Virtual Machine (JVM) keeps track of the chain of active methods with a stack
- When a method is called, the JVM pushes on the stack a frame containing
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When a method ends, its frame is popped from the stack and control is passed to the method on top of the stack
- Allows for recursion

```
main() {
    int i = 5;
    foo(i);
}

foo(int j) {
    int k;
    k = j+1;
    bar(k);
}

bar(int m) {
    …
}
```



Reverse a List using Stack

```
public class Tester {


   // … other methods here

   public void intReverse(List<Integer> l) {
        Stack<Integer> s = new Stack<Integer>();

        l.findFirst();
        while(!l.empty()) {
             s.push(l.retrieve());
             l.remove();
        }

        while(!s.empty())
             l.insert(s.pop());
    }
}
```

Parentheses Matching

- Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “]”
  - correct: ( )(( )){([( )])}
  - correct: ((( )(( )))){([( )])}
  - incorrect:)(( )){([( )])}
  - incorrect: ({[ ])}
  - incorrect: (

Parentheses Matching Algorithm

```
Algorithm ParenMatch(X, n):
    Input: An array X of n tokens, each of which is either a grouping symbol,
    a variable, an arithmetic operator, or a number
    Output: true if and only if all the grouping symbols in X match
    Let S be an empty stack
    for i=0 to n-1 do
        if X[i] is an opening grouping symbol then
            S.push(X[i])
        else if X[i] is a closing grouping symbol then
            if S.isEmpty() then
                return false {nothing to match with}
            if S.pop() does not match the type of X[i] then
                return false {wrong type}
    if S.isEmpty() then
        return true {every symbol matched}
    else
        return false {some symbols were never matched}
```

HTML Tag Matching

For fully-correct HTML, each <name> should pair with a matching </name>

```
<body>
  <center>
        <h1> The Little Boat </h1>
  </center>
  <p> The storm tossed the little
  boat like a cheap sneaker in an
  old washing machine. The three
  drunken fishermen were used to
  such treatment, of course, but
  not the tree salesman, who even as
  a stowaway now felt that he
  had overpaid for the voyage.</p>
  <ol>
        <li> Will the salesman die? </li>
        <li> What color is the boat? </li>
        <li> And what about Naomi? </li>
  </ol>
</body>
```

The Little Boat

The storm tossed the little boat like a cheap sneaker in an old washing machine. The three drunken fishermen were used to such treatment, of course, but not the tree salesman, who even as a stowaway now felt that he had overpaid for the voyage.

1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

Evaluating Arithmetic Expressions

14 – 3 * 2 + 7 = (14 – (3 * 2)) + 7

Operator precedence

* has precedence over +/–

Associativity

operators of the same precedence group evaluated from left to right

Example: (x – y) + z rather than x – (y + z)

Idea: push each operator on the stack, but first pop and perform higher and *equal* precedence operations.

Algorithm for Evaluating Expressions

Two stacks:

- opStk holds operators
- valStk holds values
- Use $ as special "end of input" token with lowest precedence

Algorithm doOp()

```
x ← valStk.pop();
y ← valStk.pop();
op ← opStk.pop();
valStk.push( y op x )
```

Algorithm repeatOps(refOp):

```
while ( valStk.size() > 1 ∧
        prec(refOp) ≤ prec(opStk.top() )
            doOp()
```

Algorithm EvalExp()

Input: a stream of tokens representing an arithmetic expression (with numbers)

Output: the value of the expression

```
while there's another token z
        if isNumber(z) then
                valStk.push(z)
        else
                repeatOps(z);
                opStk.push(z)
repeatOps($);
return valStk.top()
```

Algorithm on an Example Expression

14 ≤ 4 – 3 * 2 + 7

Operator ≤ has lower precedence than +/–