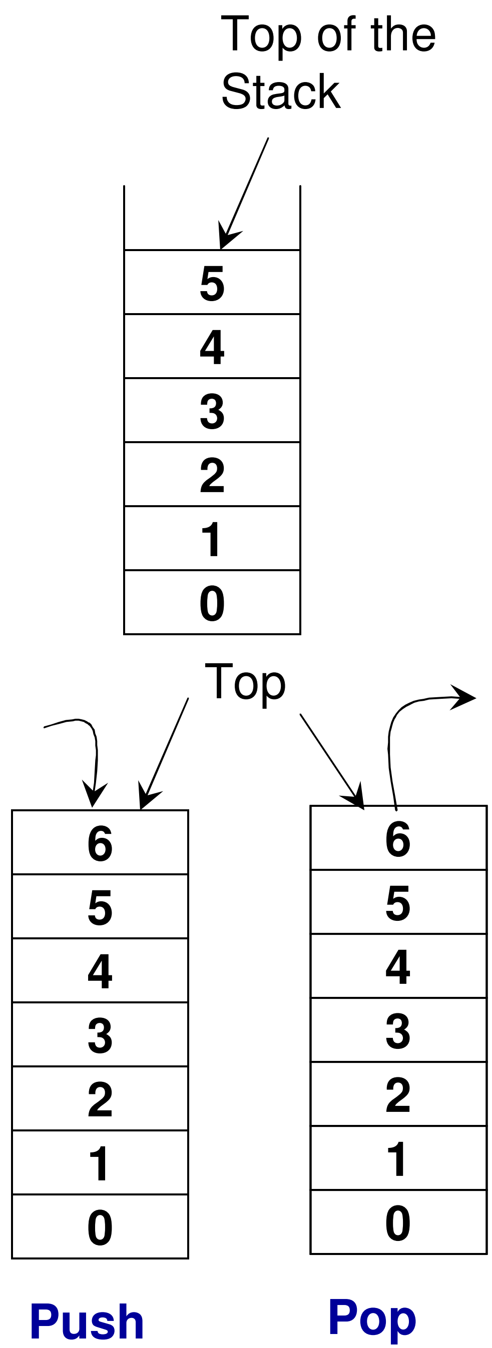
Module 3:
Stack ADT

Dr. Natarajan Meghanathan
Professor of Computer Science
Jackson State University
Jackson, MS 39217
E-mail: natarajan.meghanathan@jsums.edu

Stack ADT

- Features (Logical View)
  - A List that operates in a Last In First Out (LIFO) fashion
  - Insertion and deletion can be performed only from one end (i.e., the top of the stack)
    - The last added item has to be removed first
  - Operations:
    - Push( ) – adding an item to the top of the stack
    - Pop( ) – delete the item from the top of
    - Peek( ) – read the item in the top of the stack
    - IsEmpty( ) – whether there is any element in the top of the stack
  - All the above operations should be preferably implemented in O(1) time.



Dynamic Array-based Implementation of Stack ADT

- **List ADT**
- **Member variables**
  - int *array
  - int maxSize
  - int endOfArray
- **Constructor**
  - List(int size)
- **Member functions**

bool isEmpty()
void resize(int s)
void insert(int data)
~~void insertAtIndex(int insertIndex, int data)~~
~~int read(int index)~~
~~void modifyElement(int index, int data)~~
~~void deleteElement(int deleteIndex)~~

- **Stack ADT**
- **Member variables**
  - int *array
  - int maxSize
  - int topOfStack
- **Constructor**
  - Stack(int size)
- **Member functions**

bool isEmpty()
void resize(int s)
void push(int data)
int peek()
int pop()

Code 3.1: Dynamic Array-based Implementation of Stack ADT

C++

```
private:
    int *array;
    int maxSize;
    int topOfStack; // same as endOfArray

public:
    Stack(int size){
            array = new int[size];
            maxSize = size;
            topOfStack = -1;
    }

    bool isEmpty(){

            if (topOfStack == -1)
                    return true;

            return false;
    }
```

Java

```
private int array[];
private int maxSize;
private int topOfStack;

public Stack(int size){
        array = new int[size];
        maxSize = size;
        topOfStack = -1;
}

public boolean isEmpty(){

        if (topOfStack == -1)
                return true;

        return false;
}
```

Code 3.1 (C++): Dynamic Array-based Implementation of Stack ADT

```
void resize(int s){
    int *tempArray = array;

    array = new int[s];

    for (int index = 0; index < min(s, topOfStack+1); index++){
            array[index] = tempArray[index];
    }

    maxSize = s;
}
```

```
void push(int data){  // same as insert 'at the end'

    if (topOfStack == maxSize-1)
            resize(2*maxSize);

    array[++topOfStack] = data;

}
```

Code 3.1 (Java): Dynamic Array-based Implementation of Stack ADT

```
public void resize(int s){
    int tempArray[] = array;

    array = new int[s];

    for (int index = 0; index < Math.min(s, topOfStack+1); index++){
            array[index] = tempArray[index];
    }
    maxSize = s;
}
```

```
public void push(int data){  // same as insert 'at the end'
        if (topOfStack == maxSize-1)
                resize(2*maxSize);

        array[++topOfStack] = data;

}
```

Code 3.1 (C++): Dynamic Array-based Implementation of Stack ADT

```
int peek(){

      if (topOfStack >= 0)
         return array[topOfStack];
      else
         return -1000000;
          // an invalid value to indicate empty stack
}
```

```
int pop(){

    if (topOfStack >= 0){
          return array[topOfStack--];
          // the topOfStack is decreased by one after
          // the value is retrieved
    }
    else
          return -1000000;
          // an invalid value to indicate empty stack
}
```

Code 3.1 (Java): Dynamic Array-based Implementation of Stack ADT

```
public int peek(){

        if (topOfStack >= 0)
                return array[topOfStack];
        else
                return -1000000;// an invalid value indicating
                                // stack is empty
}
```

```
public int pop(){

        if (topOfStack >= 0){
                return array[topOfStack--];
                // the topOfStack is decreased by one
        }
        else
                return -1000000; // an invalid value indicating
                                 // stack is empty
}
```

Implementation of Stack

Dynamic Array vs. Singly/Doubly Linked List

- Push
  - Array: O(n) time, due to need for resizing when the stack gets full
  - Singly Linked List: $\Theta(n)$ time
  - Doubly Linked List: O(1) time
- Pop
  - Array: O(1) time
  - Singly Linked List: $\Theta(n)$ time
  - Doubly Linked List: O(1) time
- Peek
  - Array: O(1) time
  - Singly Linked List: $\Theta(n)$ time
  - Doubly Linked List: O(1) time

- A singly linked list-based implementation would be the most time consuming, as we would need to traverse the entire list for every push, pop and peek operation.

Code 3.2: Doubly Linked List-based Implementation of Stack

Class Node (C++) Overview

```
private:
    int data;
    Node* nextNodePtr;
    Node* prevNodePtr;
public:
    Node( )
    void setData(int)
    int getData()
    void setNextNodePtr(Node* )
    Node* getNextNodePtr( )
    void setPrevNodePtr(Node* )
    Node* getPrevNodePtr( )
```

Class Stack (C++)

```
private:
        Node* headPtr;
        Node* tailPtr;
public:
        Stack(){
                headPtr = new Node();
                tailPtr = new Node();
                headPtr->setNextNodePtr(0);
                tailPtr->setPrevNodePtr(0);
        }

        Node* getHeadPtr(){
                return headPtr;
        }

        Node* getTailPtr(){
                return tailPtr;
        }

        bool isEmpty(){

                if (headPtr->getNextNodePtr() == 0)
                        return true;

                return false;
        }
```

Code 3.2: Doubly Linked List-based Implementation of Stack

Class Node (Java) Overview

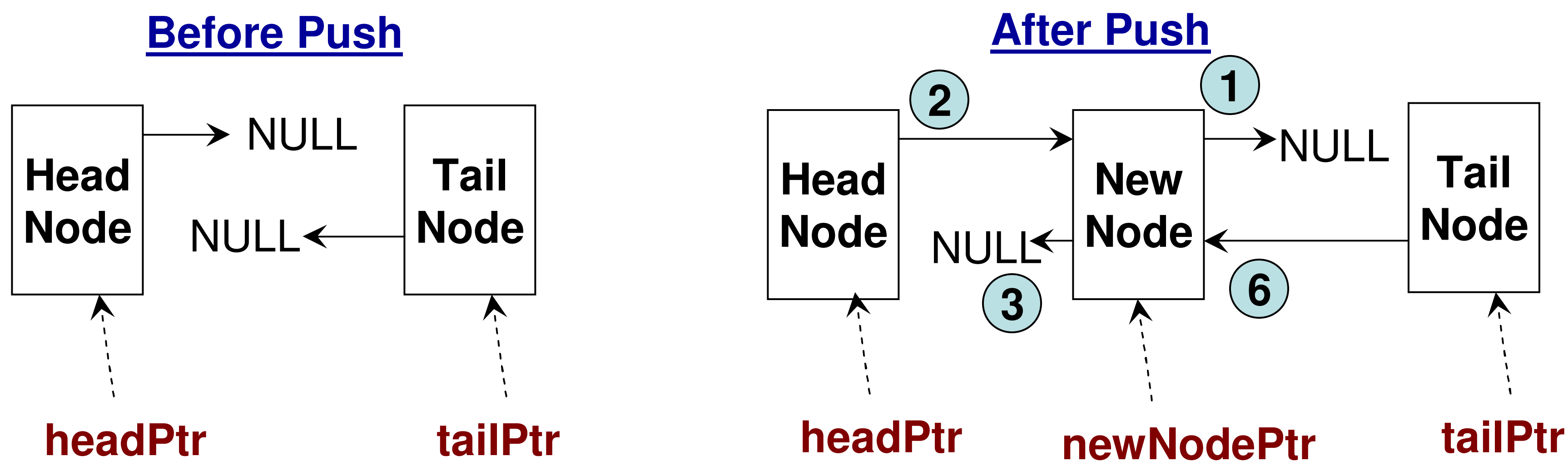
```
private  int data;
private  Node nextNodePtr;
private  Node prevNodePtr;

public  Node( )
public  void setData(int)
public  int getData()
public  void setNextNodePtr(Node)
public  Node getNextNodePtr( )
public  void setPrevNodePtr(Node)
public  Node getPrevNodePtr( )
```

Class Stack (Java)

```
class Stack{

        private Node headPtr;
        private Node tailPtr;

        public Stack(){
                headPtr = new Node();
                tailPtr = new Node();
                headPtr.setNextNodePtr(null);
                tailPtr.setPrevNodePtr(null);
        }

        public Node getHeadPtr(){
                return headPtr;
        }

        public Node getTailPtr(){
                return tailPtr;
        }

        public boolean isEmpty(){

                if (headPtr.getNextNodePtr() == null)
                        return true;

                return false;
        }
```

Push Operation

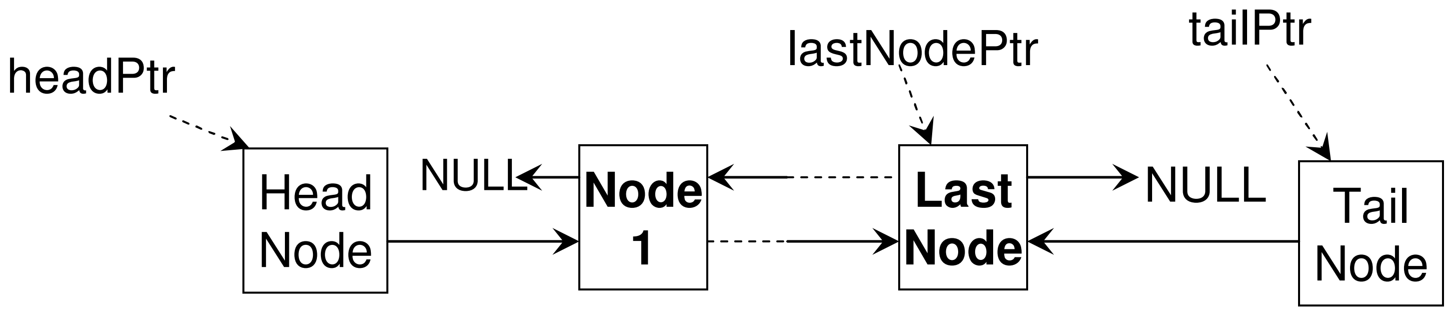
Scenario 1: There is no node currently in the stack



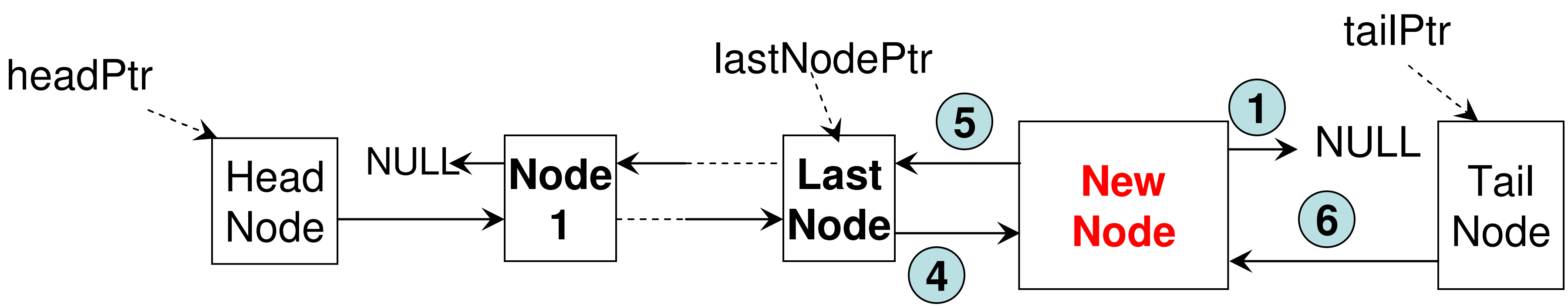
Push Operation

Scenario 2: There is at least one node already in the stack

// Before the new node is pushed, the prevNodePtr for the "tail node"
// would be pointing to the last node in the stack and the nextNodePtr
// for that last node would be pointing to NULL.



Before Push



After Push

Code 3.2 (C++)

```
void push(int data){

    Node* newNodePtr = new Node();
    newNodePtr->setData(data);
    newNodePtr->setNextNodePtr(0); (1)

    Node* lastNodePtr = tailPtr->getPrevNodePtr();

    if (lastNodePtr == 0){   // There is no other node in the Stack (Scenario 1)

        headPtr->setNextNodePtr(newNodePtr); (2)
        newNodePtr->setPrevNodePtr(0); (3)

    }
    else{  // There is at least one node already in the Stack (Scenario 2)

        lastNodePtr->setNextNodePtr(newNodePtr); (4)
        newNodePtr->setPrevNodePtr(lastNodePtr); (5)

    }

    tailPtr->setPrevNodePtr(newNodePtr); (6)

}
```

Whatever be the case, the prevNodePtr for the tail node will point to the newly pushed node

Code 3.2 (Java)

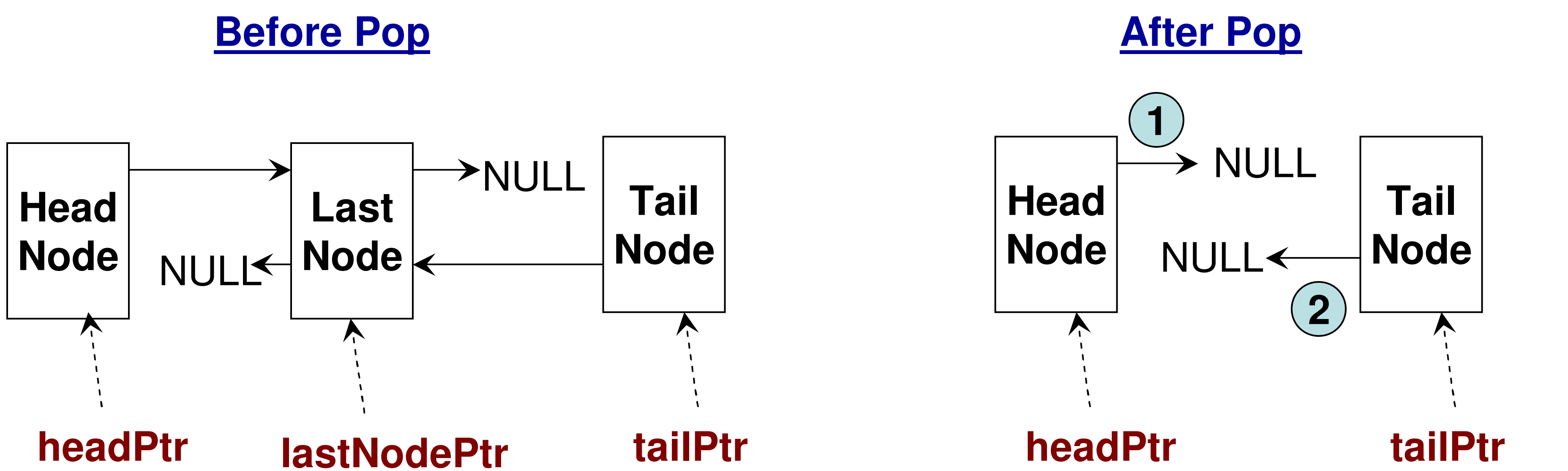
```
public void push(int data){

    Node newNodePtr = new Node();
    newNodePtr.setData(data);
    newNodePtr.setNextNodePtr(null); (1)

    Node lastNodePtr = tailPtr.getPrevNodePtr();

    if (lastNodePtr == null){ // There is no other node in the Stack (Scenario 1)
        headPtr.setNextNodePtr(newNodePtr); (2)
        newNodePtr.setPrevNodePtr(null); (3)
    }
    else{ // There is at least one node already in the Stack (Scenario 2)
        lastNodePtr.setNextNodePtr(newNodePtr); (4)
        newNodePtr.setPrevNodePtr(lastNodePtr); (5)
    }

    tailPtr.setPrevNodePtr(newNodePtr); (6)
}
```

Whatever be the case, the prevNodePtr for the tail node will point to the newly pushed node

Pop Operation

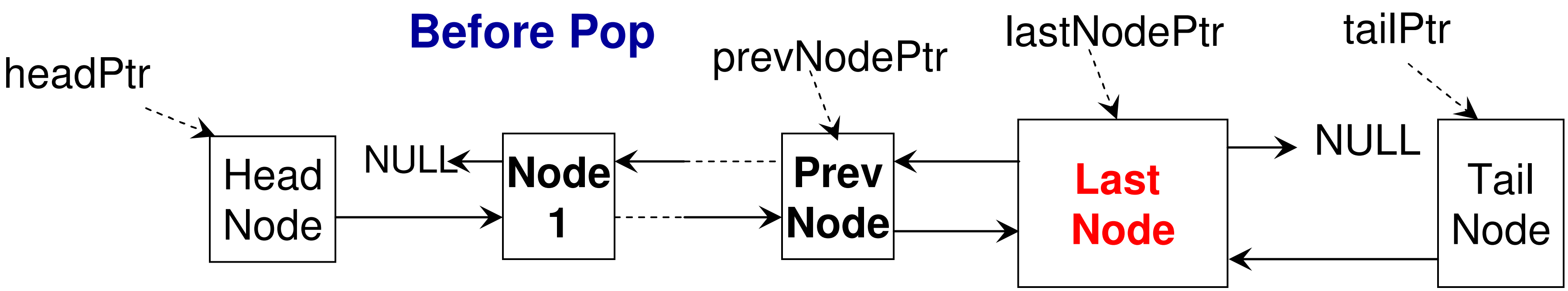
Scenario 1: There will be no node in the Stack after the Pop (i.e., there is just one node in the Stack before the Pop)

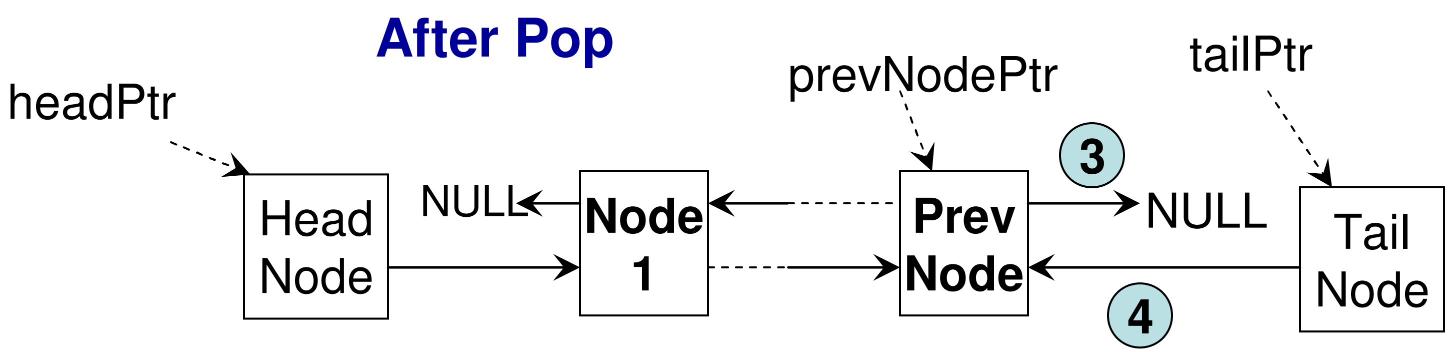
// Before Pop: The Head Node's nextNodePtr and the Tail Node's prevNodePtr are both pointing to the only node in the stack.
// After Pop: Both the Head Node's nextNodePtr and the Tail Node's prevNodePtr are set to NULL



Pop Operation

Scenario 2: There will be at least one node in the stack after the Pop operation is executed





Code 3.2 (C++)

```
int pop(){

        Node* lastNodePtr = tailPtr->getPrevNodePtr();
        Node* prevNodePtr = 0;

        int poppedData = -100000; //empty stack

        if (lastNodePtr != 0){  // If there is at least one node in the Stack before Pop
                prevNodePtr = lastNodePtr->getPrevNodePtr();
                poppedData = lastNodePtr->getData();
        }
        else  // If the Stack is empty before pop, return an invalid value
                return poppedData;

        if (prevNodePtr != 0){  // If there is going to be at least one node in the
                                //                           Stack after the pop
                                //                                  (Scenario 2)
                prevNodePtr->setNextNodePtr(0); 3
                tailPtr->setPrevNodePtr(prevNodePtr); 4
        }
        else{   // If there is going to be no node in the Stack after the pop
                //                                  (Scenario 1)
                headPtr->setNextNodePtr(0); 1
                tailPtr->setPrevNodePtr(0); 2
        }

        return poppedData;
}
```

Code 3.2 (Java)

```
public int pop(){

        Node lastNodePtr = tailPtr.getPrevNodePtr();
        Node prevNodePtr = null;

        int poppedData = -100000; //empty stack

        if (lastNodePtr != null){  // If there is at least one node in the Stack before Pop
                prevNodePtr = lastNodePtr.getPrevNodePtr();
                poppedData = lastNodePtr.getData();
        }
        else  // If the Stack is empty before Pop, return an invalid value
                return poppedData;

        if (prevNodePtr != null){  // If there is going to be at least one node in the
                prevNodePtr.setNextNodePtr(null); (3)             Stack after the pop
                tailPtr.setPrevNodePtr(prevNodePtr); (4)               (Scenario 2)
        }
        else{   // If there is going to be no node in the Stack after the pop
                headPtr.setNextNodePtr(null); (1)                  (Scenario 1)
                tailPtr.setPrevNodePtr(null); (2)
        }

        return poppedData;
}
```

Code 3.2: Peek Operation

C++

```
int peek(){

        Node* lastNodePtr = tailPtr->getPrevNodePtr();

        if (lastNodePtr != 0)
                return lastNodePtr->getData();
        else
                return -100000; //  empty stack
}
```

Java

```
public int peek(){

        Node lastNodePtr = tailPtr.getPrevNodePtr();

        if (lastNodePtr != null)
                return lastNodePtr.getData();
        else
                return -100000; //  empty stack
}
```

Code 3.3 (C++): String Processing Example

This code reads a string (of possibly more than one Word) from the user and prints a new string that has the uppercase characters of the original string as well as reverses the string

```
#include <string>
#include <cstring>// to get the character array of a string
#include <iostream>
#include <algorithm> // reverse
using namespace std;

int main(){

        string originalString;
        cout << "Enter a string: ";
        getline(cin, originalString);
```

To read more than word (a line) as string

```
        string upperCaseString("");
```

Initialize a new string as an empty string

```
        for (int index = 0; index < originalString.size(); index++){

                char c = originalString[index];
                upperCaseString += toupper(c);

        }

        cout << upperCaseString << endl;

        reverse(originalString.begin(), originalString.end());

        cout << "reversed string: " << originalString << endl;

return 0;
}
```

Getting the character at a specific index

To get the uppercase version of a character

Reverse the string from its end to its beginning

Code 3.3 (Java): String Processing Example

This code reads a string (of possibly more than one Word) from the user and prints a new string that has the uppercase characters of the original string as well as reverses the string

```
import java.util.*;

class StringProcessing{

        public static void main(String[] args){

        Scanner input = new Scanner(System.in);

        String originalString;
        System.out.print("Enter a string: ");
        originalString = input.nextLine();

        String upperCaseString = "";

        for (int index = 0; index < originalString.length(); index++){

            char c = originalString.charAt(index);
            upperCaseString += Character.toUpperCase(c);

        }

        System.out.println(upperCaseString);

        String reverseString = "";

        for (int index = originalString.length()-1; index >= 0; index--)
                reverseString += originalString.charAt(index);

        System.out.println("reverse string: "+reverseString);

        }

}
```

To read more than word (a line) as string

Initialize a new string as an empty string

Getting the character at a specific index

To get the uppercase version of a character

Note: String objects are not mutable in Java. Hence, we have to create a new String object that is the reverse of the original string.

Reverse the string from its end to its beginning

Parentheses Balance

- By parenthesis, we refer to the following symbols

  ( ), { }, [ ]
- The problem is about checking whether corresponding to each opening parenthesis there is a corresponding closing parenthesis in correct order.
- Examples for balanced parentheses
  - { [ ] (( )) }
  - [({ }) [ ]]
  - ({ } [( )])
- Examples for unbalanced parentheses
  - [[ )]
  - { ( ) [}

Parentheses Balance (Program Logic)

- Logic to determine whether the parentheses in an expression are balanced or not. (We could use a Linked List or Dynamic Array-based Stack).
  - Input the expression as a string and read it one character at a time.
  - If the character read is a opening parenthesis, then push it into the stack
  - If the character read is a closing parenthesis, then pop the stack and check if the popped symbol is a matching opening parenthesis.
    - If so, continue.
    - Otherwise, stop and say, parenthesis is not balanced.
  - If the character read does not match with any of the above six symbols, then stop the program and say there is an invalid symbol in the input expression.

Code 3.4 (C++): Parentheses Balancing

Note: We will use the implementation of stack using doubly linked list. We will replace all the ‘int’ in the doubly linked list – based stack code to ‘char’ as appropriate

```
int main(){
    Stack stack;

    string expression;
    cout << "Enter an expression: ";
    cin >> expression;

    int index = 0;

     while (index < expression.size()){

                    char symbol = expression[index];

                    if (symbol == '{' || symbol == '(' || symbol == '['){
                              stack.push(symbol);
                              index++;
                              continue;
                    }
```

Code 3.4 (C++): Parentheses Balancing

```
        else if (symbol == '}' || symbol == ')' || symbol == ']'){

            char topSymbol = stack.pop();
            if ( (topSymbol == '{' && symbol == '}') ||
                (topSymbol == '(' && symbol == ')') ||
                    (topSymbol == '[' && symbol == ']') ){

                        index++;
                        continue;

            }
            else{

                cout << "parenthesis not balanced!!" << endl;
                return 0;

            }
        }
        else{

          cout << "Invalid symbol "<< symbol << " in the expression!!" << endl;
          return 0;
        }

    }

    cout << expression << " is balanced!!" << endl;

return 0;
}
```

Code 3.4 (Java): Parentheses Balancing

Note: We will use the implementation of stack using doubly linked list. We will replace all the ‘int’ in the single linked list – based stack code to ‘char’ as appropriate

```
class DoublyLinkedList{

 public static void main(String[] args){

        String expression;

        Scanner input = new Scanner(System.in);
        System.out.print("Enter an expression: ");
        expression = input.nextLine();

        Stack stack = new Stack();

        int index = 0;

        while (index < expression.length()){

                char symbol = expression.charAt(index);

                if (symbol == '{' || symbol == '(' || symbol == '['){
                        stack.push(symbol);
                        index++;
                        continue;
                }
```

Code 3.4 (Java): Parentheses Balancing

```
else if (symbol == '}' || symbol == ')' || symbol == ']'){

        char topSymbol = stack.pop();
        if ( (topSymbol == '{' && symbol == '}') ||
            (topSymbol == '(' && symbol == ')') ||
                (topSymbol == '[' && symbol == ']') ){

                index++;
                continue;

        }
        else{
            System.out.println("parenthesis not balanced!!");
            return;
        }
    }
    else{

     System.out.println("Invalid symbol "+ symbol + " in the expression!!");
    return;
    }

}

System.out.println(expression + " is balanced!!");

}

}
```

Example (C++) for String Tokenization

(breaking a string into tokens based on delimiters)

Code 3.5

```
#include <iostream>
#include <string>
#include <cstring>
using namespace std;

int main(){

        string sample;

        cout << "Enter an expression: ";
        getline(cin, sample);

        char* sampleArray = new char[sample.length()+1];
        strcpy(sampleArray, sample.c_str());

        char* cptr = strtok(sampleArray, ", ");

        int numSymbols = 0;
        int sumIntegers = 0;
```

// for C-style string processing as character array

In this example program, we will count the number of Symbols and the sum of the integers that appear in an input string 'sample'

Get a line of words as a string, sample

Create a character array of size one more than the length of the string and copy the elements from the string 'sample' to the Character array 'sampleArray'

Set up a tokenizer for the character Array with , and blank space as Delimiters. The tokenizer will return Tokens as character arrays (strings)

Code 3.5 (C++)

```
while (cptr != 0){
        string token(cptr);
        if ( (token.compare("@") == 0) ||
            (token.compare("!") == 0) ||
                (token.compare("#") == 0) ||
                (token.compare("$") == 0) ||
                (token.compare("%") == 0) ){
                numSymbols++;
        }
        else{
                int value = stoi(token);
                sumIntegers += value;
        }

        cptr = strtok(NULL, ", ");
}

cout << "number of operators: " << numSymbols << endl;
cout << "sum of the integers: " << sumIntegers << endl;

return 0;
}
```

Run the while loop unless the pointer Corresponding to a token (character array) is NULL

Generate a string ‘token’ corresponding to the Character array

The ‘compare’ function returns 0 if the two strings are equal

Keep track of the number of symbols

The ‘stoi’ function converts a string to the integer representing it. For example, if ’13’ is the string token, it is now transformed to an integer ‘value’

Syntax of the strtok function to read the next token in the original string

Example (Java) for String Tokenization

(breaking a string into tokens based on delimiters)

Code 3.5

```
import java.util.*;    // to use the StringTokenizer and Scanner class

class stringTokenizing{

    public static void main(String[] args){

    String sample;
    Scanner input = new Scanner(System.in);

    System.out.print("Enter an expression: ");
    sample = input.nextLine();

    int numSymbols = 0;
    int sumIntegers = 0;

    StringTokenizer stk = new StringTokenizer(sample, ", ");
```

In this example program, we will count the number of Symbols and the sum of the integers that appear in an input string 'sample'

Get a line of words as a string, sample

Set up a tokenizer for the character Array with , and blank space as Delimiters. The tokenizer will return Tokens as strings

Code 3.5 (Java)

```
while (stk.hasMoreTokens()){
        String token = stk.nextToken();

        if ( (token.equals("@") ) ||
            (token.equals("!") ) ||
                (token.equals("#") ) ||
                (token.equals("$") ) ||
                (token.equals("%") ) ){
              numSymbols++;
        }
        else{
              int value = Integer.parseInt(token);
              sumIntegers += value;
        }

}

System.out.println("number of operators: " + numSymbols);
System.out.println("sum of the integers: " + sumIntegers);

}

}
```

Run the while loop until there are tokens remaining

Retrieve the next token for the Original string ‘sample’

The ‘equals’ function returns true if the two strings are equal

Keep track of the number of symbols

The Integer.parseInt function returns the integer embedded inside a string

Order of Operation (Operator Precedence)

1) Parenthesis: ( ), { }, [ ]
2) Exponent: In case of a tie, we evaluate from right to left.
   Example: 3^2^4 = 3^16 = 43046721
3) Multiplication and Division: Break the tie, by evaluating from left to right.
4) Addition and Subtraction: Break the tie, by evaluating from left to right.

Example:

1) 5 + 8 / 4 = 5 + 2 = 7
2) 12 / 6 * 3 = 2 * 3 = 6
3) 4 * 5 / 2 – 7 + 3
   = 20 / 2 – 7 + 3
   = 10 – 7 + 3
   = 3 + 3 = 6
4) 4 * {5 / (2 – 7) + 3}
   = 4 * {5 / (-5) + 3}
   = 4 * {-1 + 3} = 8

Infix, Prefix and Postfix

- Infix: LeftOperand **<Operator>** RightOperand
  - Example: 2 + 3
- Prefix: **<Operator>** LeftOperand RightOperand
  - Example: + 2 3
- Postfix: LeftOperand RightOperand **<Operator>**
  - Example: 2 3 +
- Infix expressions use the order of operation to break the ties.
- Prefix and Postfix expressions do not require the order of operation.
  - In both prefix and postfix expressions, each operand will be associated only with one operator and hence no need to use rules of operator precedence.
  - For example: consider a + b * c: this expression (infix notation) needs to use operator precedence for evaluation
  - + a * b c is the prefix notation and abc*+ is the postfix notation

Evaluation of Postfix Expression

Consider an infix expression: A * B + C * D – E

If evaluated in infix, the expression needs to be evaluated as follows:

(A * B) + (C * D) – E

{ (A * B) + (C * D)} – E

Converting this to postfix

(AB*) + (CD*) – E

(AB*) (CD*) + – E

(AB*) (CD*) + E –

Removing the parenthesis, the final postfix expression is: AB*CD*+E–

Evaluation Logic:

Scan the expression from left to right.

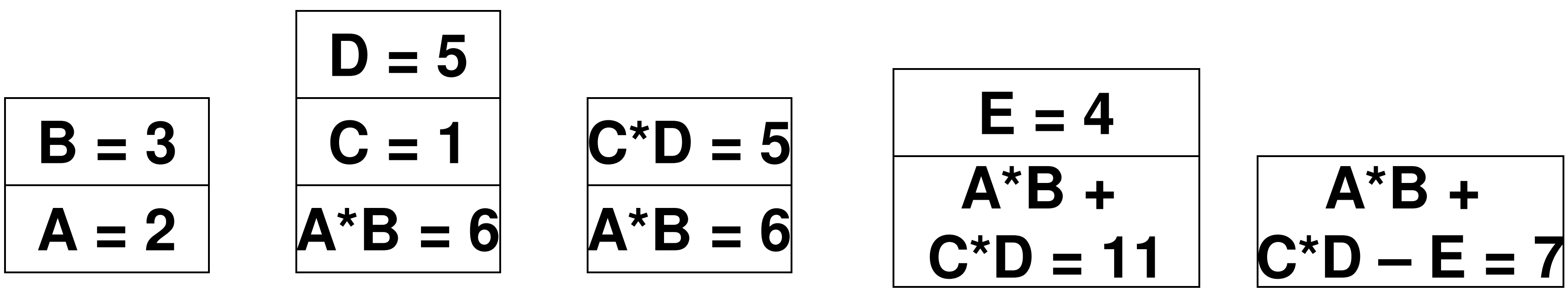
If we see an operand in the expression, push it into the stack.

If we see an operator, we pop the last two items from the stack, apply the operator on the two popped items (**the first popped item will be the right operand and the second popped item will the left operand**) and push the result of the operation to the stack.

The only item in the stack after reading the entire expression is the value of the expression.

Evaluation of Post-Fix Expression

- Consider the post-fix expression
- AB*CD*+E –
- Let A = 2, B = 3, C = 1, D = 5, E = 4



Note: During a scan of a post-fix expression, the left operand of an operator goes first into the stack followed by the right operand. Hence, during a pop, the right operand comes first out of the stack, followed by the left operand

C++ Code for Postfix Evaluation

Code 3.6

We will use the integer-based doubly linked list implementation of stack.

```
Stack stack;

string expression;

cout << "Enter the expression to evaluate: ";
getline(cin, expression);
char* expressionArray = new char[expression.length()+1];
strcpy(expressionArray, expression.c_str());

char* cptr = strtok(expressionArray, ", ");

while (cptr != 0){

    string token(cptr);

    bool isOperator = false;

    if ( (token.compare("*") == 0) || (token.compare("/") == 0) ||
         (token.compare("+") == 0) || (token.compare("-") == 0) )
         isOperator = true;

    if (!isOperator){
         int val = stoi(token);
         stack.push(val);
    }
```

Check if the token is one of the four operators *, /, +, -; if so, set the 'isOperator' boolean to true

If the token is not an operator, we assume It must be an integer, and push it into the Stack.

Code 3.6 (C++) continued

```
        if (isOperator){

            int rightOperand = stack.pop();
            int leftOperand = stack.pop();

            if (token.compare("*") == 0){
                int result = leftOperand * rightOperand;
                cout << "intermediate result: " << result << endl;
                stack.push(result);
            }
            else if (token.compare("/") == 0){
                int result = leftOperand / rightOperand;
                cout << "intermediate result: " << result << endl;
                stack.push(result);
            }
            else if (token.compare("+") == 0){
                int result = leftOperand + rightOperand;
                cout << "intermediate result: " << result << endl;
                stack.push(result);
            }
            else if (token.compare("-") == 0){
                int result = leftOperand - rightOperand;
                cout << "intermediate result: " << result << endl;
                stack.push(result);
            }

        } //end if

        cptr = strtok(NULL, ", ");
    } // end while

    cout << "final result: " << stack.pop() << endl;

  return 0;
  }
```

The right operand is popped first followed by the Left operand

If 'isOperator' is true, then pop the top two integers from the Stack, perform the operation and Push the resulting value to the stack

Set up the next iteration of the while loop by retrieving the next token

The final value of the expression will be the only value in the stack when we exit the while loop.

Java Code for Postfix Evaluation

Code 3.6

We will use the integer-based doubly linked list implementation of stack.

```
Stack stack = new Stack();

String expression;

Scanner input = new Scanner(System.in);

System.out.print("Enter the expression to evaluate: ");
expression = input.nextLine();

StringTokenizer stk = new StringTokenizer(expression, ", ");

while (stk.hasMoreTokens()){

        String token = stk.nextToken();

        boolean isOperator = false;

        if ( (token.equals("*")) || (token.equals("/")) || (token.equals("+")) ||
        (token.equals("-")) )
                isOperator = true;

        if (!isOperator){
                int val = Integer.parseInt(token);
                stack.push(val);
        }
```

Retrieve the next token

Check if the token is one of the four operators *, /, +, -; if so, set the ‘isOperator’ boolean to true

If the token is not an operator, we assume it must be an integer, and push it into the Stack.

Code 3.6 (Java) continued

The right operand is popped first followed by the Left operand

If 'isOperator' is true, then pop the top two integers from the Stack, perform the operation and Push the resulting value to the stack

```
if (isOperator){

        int rightOperand = stack.pop();
        int leftOperand = stack.pop();

        if (token.equals("*")){
                int result = leftOperand * rightOperand;
                System.out.println("intermediate result: " + result);
                stack.push(result);
        }
        else if (token.equals("/")){
                int result = leftOperand / rightOperand;
                System.out.println("intermediate result: " + result);
                stack.push(result);
        }
        else if (token.equals("+")){
                int result = leftOperand + rightOperand;
                System.out.println("intermediate result: " + result);
                stack.push(result);
        }
        else if (token.equals("-")){
                int result = leftOperand - rightOperand;
                System.out.println("intermediate result: " + result);
                stack.push(result);
        }

} //end if

} // end while

System.out.println("final result: " + stack.pop() );
```

The final value of the expression will be the only value in the stack when we exit the while loop.

Evaluation of Prefix Expression

Consider an infix expression: A * B + C * D – E

If evaluated in infix, the expression needs to be evaluated as follows:

(A * B) + (C * D) – E

{ (A * B) + (C * D)} – E

Converting this to prefix

(*AB) + (*CD) – E

+ (*AB) (*CD) – E

– + (*AB) (*CD) E

Removing the parenthesis, the final prefix expression is: – + *AB*CDE

Evaluation Logic:

Scan the expression from right to left (or reverse the expression and scan from left to right).

If we see an operand in the expression, push it into the stack.

If we see an operator, we pop the last two items from the stack, apply the operator on the two popped items (**the first popped item will be the left operand and the second popped item will the right operand**) and push the result of the operation to the stack.

The only item in the stack after reading the entire expression is the value of the expression.

Evaluation of Pre-Fix Expression

- Consider the pre-fix expression
- – + *AB*CDE

  Read this expression from right to left
- Let A = 2, B = 3, C = 1, D = 5, E = 4

	A = 2			
C = 1	B = 3	A*B = 6	A*B + C*D = 11	A*B + C*D – E = 7
D = 5	C*D = 5	C*D = 5		
E = 4	E = 4	E = 4	E = 4	

Note: During a scan of a pre-fix expression, the right operand of an operator goes first into the stack followed by the left operand. Hence, during a pop, the left operand comes first out of the stack, followed by the right operand