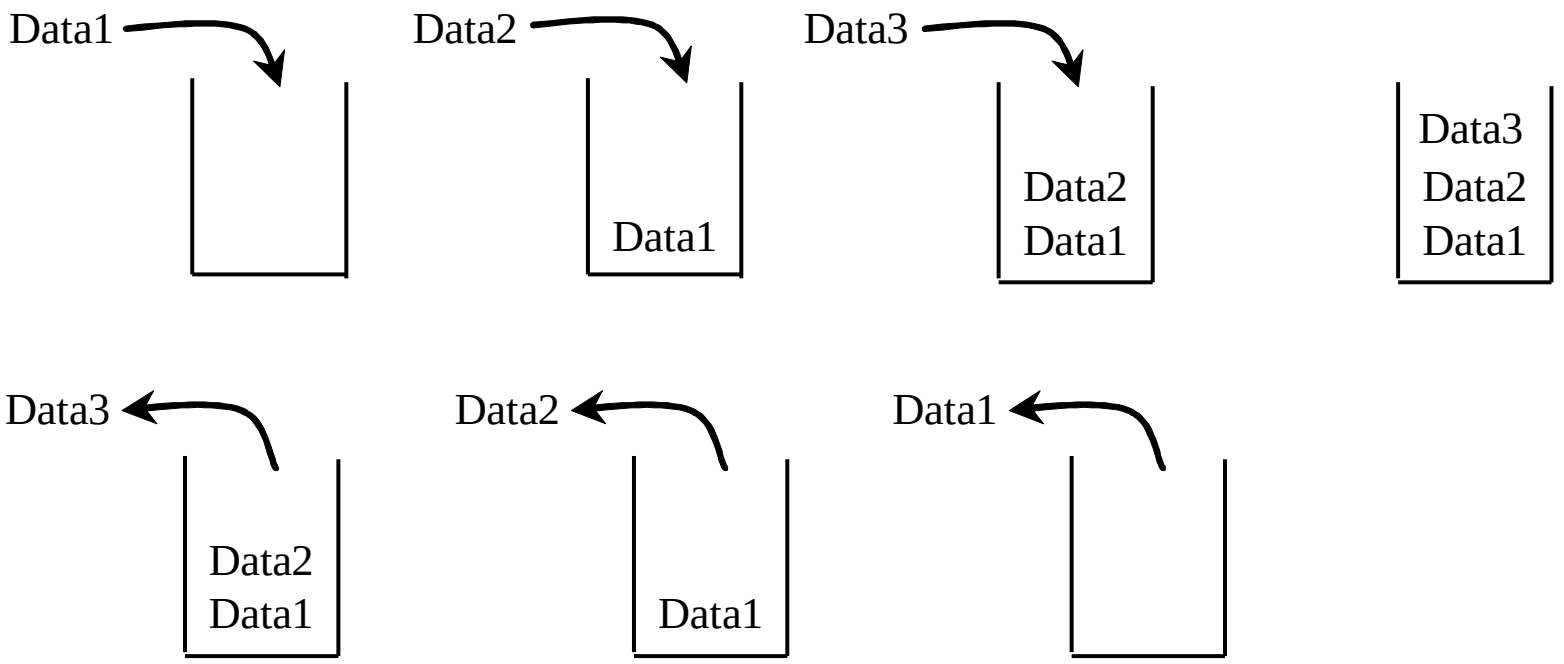
Chapter 20 - Part 3

Stacks

Stacks

A stack is a special type of list, where the elements are accessed, inserted, and deleted only from the end, called the top of the stack. Stack behavior is knows as LIFO.



Stack ADT Specifications

```
public StackInterface {
public void push(Object item) throws StackException;
    //Precondition: item is the new item to be added.
    //Postcondition: If insertion is successful, item is on the top of the stack.
    //Postcondition: Throw StackException if the insertion is not successful.
public void pop( ) throws StackException;
    //Precondition: none.
    //Postcondition: If stack is not empty, the item on the top is removed from the stack.
    //Postcondition: Throws StackException if the stack is empty.
public Object top( ) throws StackException;
    //Precondition: none.
    //Postcondition: If stack is not empty, the item on the top is returned. The stack is left unchanged.
    //Postcondition: Throws StackException if the stack is empty
public int size( );
    //Precondition: none.
    //Postcondition: stack size is returned.
public boolean isEmpty();
    //Precondition: none.
    //Postcondition: Returns true if the stack is empty, otherwise returns false.
public boolean isFull();
    //Precondition: none.
    //Postcondition: Returns true if the stack is full, otherwise returns true. }
```

Stack Operations

Basic stack operations:

```
Push(e);      // add element e to the top of stack
Pop();        // return and remove the top element from the stack
Top();        // return (don't remove) the top element from the stack
Size();       // return number of elements in the stack
isEmpty();    // return true if stack is empty
isFull();     // return true if stack is full
```

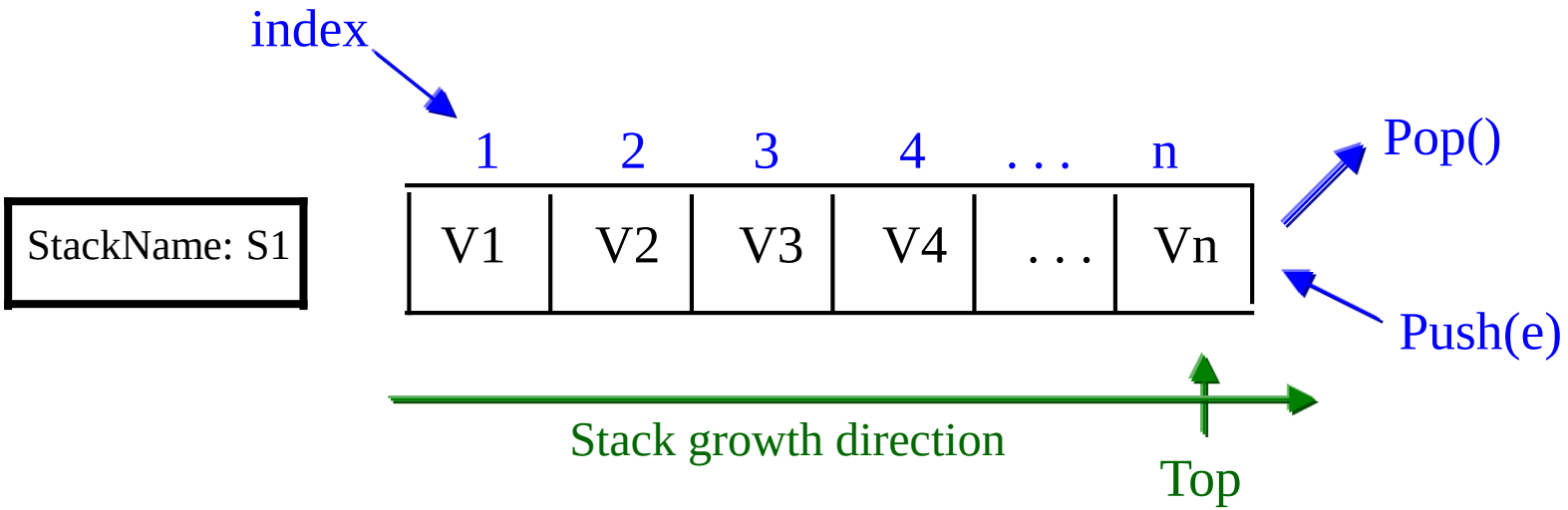
Basic Variables:

```
Size;         // hold current size of the stack  (-1 if empty)
MAX_SIZE;     // hold max size of the stack
```

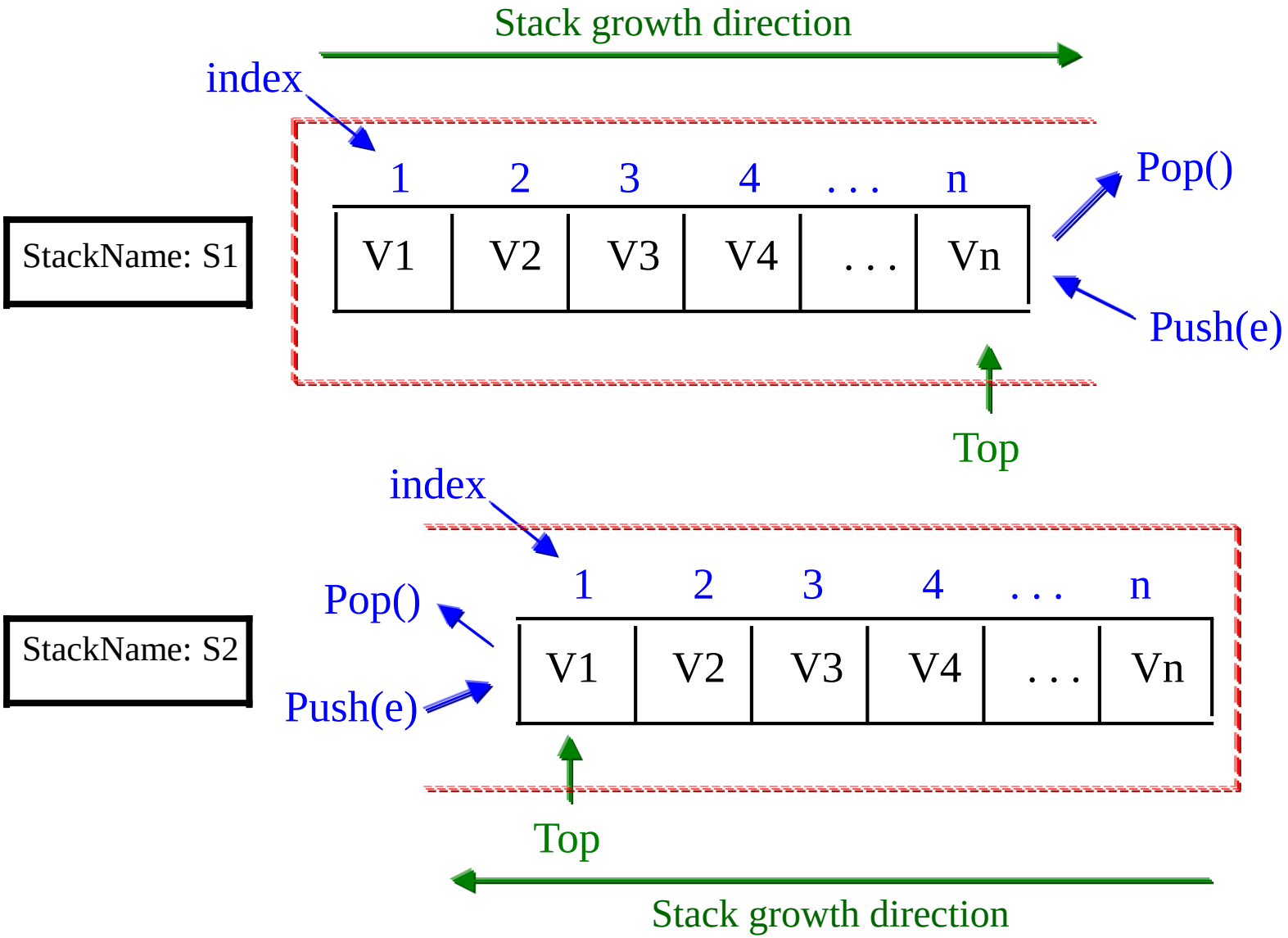
Stack Implementation Using Array

Using an array list to implement Stack

Since the insertion and deletion operations on a stack are made only at the end of the stack, using an array to implement a stack is more efficient than a linked list.



Stack Implementation Using Array



Stack Implementation Using Array

```
//Pseudo code for stack S1
Push(E e)
{   if !isFull()
    { Top = Top + 1;
      S(Top) = e;
    }
    else Stack-Full-Error
}

Pop()
{   if !isEmpty()
       Top = Top - 1;
    else Stack-Empty-Error
}

Top()
{   if !isEmpty()
       Return S(Top);
    else Stack-Empty-Error
}
```

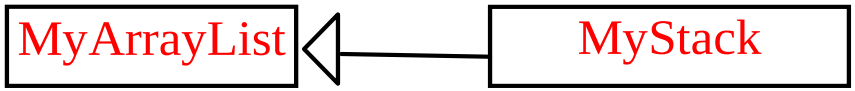
```
//Pseudo code for stack S1
Size()
{ Return size; }

isFull()
{   if (Top >= MAX_SIZE)
        Return True;
    else Return False;
}

isEmpty()
{   if (size < 0)
       Return True;
    else Return False;
}
```

MyStack in Java Using Array - Textbook

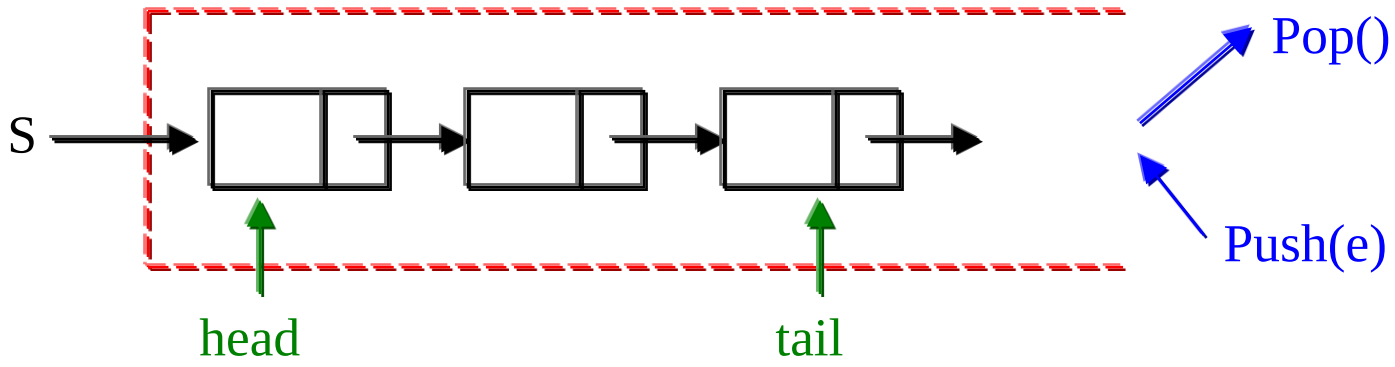
MyStack	
-list: MyArrayList	
+isEmpty(): boolean	Returns true if this stack is empty.
+getSize(): int	Returns the number of elements in this stack.
+peek(): Object	Returns the top element in this stack.
+pop(): Object	Returns and removes the top element in this stack.
+push(o: Object): Object	Adds a new element to the top of this stack.
+search(o: Object): int	Returns the position of the specified element in this stack.

Note: using inheritance:

Declare class Stack by extending class Array List (java specific) so that specific array list operations can be used to emulate the stack behavior (add and delete from the top of the stack).



Stack Implementation as Linked List




```
Push(e);        // add element e to the top of stack
Pop();          // remove the top element from the stack
Top();// return (don't remove) the top element from the stack
Size();         // return number of elements in the stack
isEmpty();      // return true if stack is empty
Size;           // variable to hold current size of the stack
```

Method Push(e)

```
//Pseudo code - Method Push(e). Same as method AddLastNode(e)
Method Push(E e)
{
  // 1. create and initialize a new bode
  Node<E> newNode = new Node<E>(e);

  // 2. check if empty stack
  if (Head == NULL)
       Head = newNode;
       Tail = newNode;

  //3. Generic case, add to end of list
  Tail.next = newNode;
  Tail = Tail.next;
}
```

Method Pop()

```
//Pseudo code - Method Pop(). Same as method RemoveLastNode()
Method Pop()
{
// 1. check if empty stack
   if (size == 0) return null;
// 2. check if one node stack
   if (size == 1)
   {  Node<E> temp = head;
      head = tail = null;
      size = 0; }
// 3. Generic case 2 or more nodes stack
    Node<E> current = head; //declare current and set to head
    for (int i = 1; i < size - 1; i++)
        current = current.next; //move current to second last node
    current.next = null; //disconnect last node
    tail = current; //set tail to current
    size--; //decrease list size by one element
}
```

Methods Top(), Size(), and isEmpty()

```
//Pseudo code - Method Top()
Method Top()
{
   Return tail.data;
}

//Pseudo code - Method Size()
Method size()
{
   Return size;
}

//Pseudo code - Method isEmpty()
Method isEmpty()
{
   Return (size == 0);
}
```

Stack Applications

Stacks have wide range of applications including

1. Reversing data (files or strings)
2. Checking for palindrome strings
3. Prefix/Postfix expression evaluation (Polish Notation)
4. Expression translation to prefix/postfix notation.
5. Backtracking (such as find the way through a maze, finding a path from one point in a graph (roadmap) to another point, and play a game in which there are moves to be made (checkers, chess).
6. Memory management during run-time. Each program has its own memory allocations containing activating records.
7. Others…

Reversing Data

Algorithm:

```
// reverse the content of a data file.
Step 1: read the first word from the input file
        data = readFromFile();
Step 2: loop until end of file, push all data onto the stack
        while (not EOF and stackName.isFull() != true)
        {   stackName.push(data);
            data = readFromFile();
        }
Step 3: loop until stack is empty
        while (stackName.isEmpty() != true)
        {   print(stackName.Top());
            stackName.pop(data);
        }
```

Checking for Palindromes

Algorithm:

```
// check if a string is a palindrome.
Step 1: declare variables and read a string form the user
        String string1 = "", strign2 = "";
        string1 = scan.next();  //read string1
Step 2: loop on string1 and push characters onto the stack
        For (int i = 1; i <= string1.length; i++)
             stackName.push(string1.charAt(i));
Step 3: loop until stack is empty to create new string
        while (stackName.isEmpty() != true)
        { string2 = string2 + stackName.Top();//concatenation
          stackName.pop();
        }
Step 4: Compare string1 to string2
        if (string1 == string2)
            print("string1 is palindrome");
        else
            print("string1 is not palindrome");
```

Prefix and Postfix Notation

Conversion Rules:

E1 **op** ***E2***	**==>**	***E1E2*****op**	(postfix)	A+B ==> AB+
E1 **op** ***E2***	**==>**	**op*****E1E2***	(prefix)	A+B ==> +AB
E1	**==>**	***E1***	(postfix)	A ==> A
E1	**==>**	***E1***	(prefix)	A ==> A

Manual conversion: fully parenthesize the expression and work it inside out following these rules.

Example:

9-4*2+1 => Prefix: +-9*421 postfix: 942*-1+

(4-2)*(2+3) => Prefix: *-42+23 postfix: 42-23+*

Now, try this one: **(3+2)-(5*7)-(4/2)**

Postfix Evaluation

Steps:

Scan expression left to right

If operand: push onto stack.

If operator: Get 2 top elements, do the math (R = Top2 OP Top1), pop 2 top elements, and push R back onto the stack.

Example:

```
1 2 3 + *  => 5    Note: infix is 1*(2+3)

Postfix        Stack( bottom -> top )
1 2 3 + *
  2 3 + *      1
    3 + *      1 2
      + *      1 2 3
        *      1 5     // 5 from 2 + 3
               5       // 5 from 1 * 5
```

Postfix Evaluation

Algorithm:

```
create a new stack
While (input stream is not empty)
{   token = getNextToken();//scan input left to right
    if (token instanceof operand) //if operand
         push(token);
    else if (token instance of operator) //if operator
         { operand1 = pop();
           operand2 = pop();
           result = calc(token, operand2, operand1);
           push(result);
         }
}
return pop();
```

Prefix Evaluation

Steps:

Scan expression right to left

If operand: push onto stack.

If operator: Get 2 top elements, do the math (R = Top1 OP Top2), pop 2 top elements, and push R back onto the stack.

Example:

```
* 1 + 2 3   => 5     Note: infix is 1*(2+3)

Postfix        Stack( bottom -> top )
* 1 + 2 3
* 1 + 2        3
* 1            3 2
* 1            5       // 5 from 2 + 3
*              5 1
               5       // 5 from 1 * 5
```

Postfix Evaluation

Algorithm:

```
create a new stack
While (input stream is not empty)
{   token = getNextToken(); //scan input right to left
    if (token instanceof operand) //if operand
         push(token);
    else if (token instance of operator) //if operator
         { operand1 = pop();
           operand2 = pop();
           result = calc(token, operand1, operand2);
           push(result);
         }
}
return pop();
```

Practice Example

Now, try this expression. Show conversion to prefix and postfix and then show stack evaluations of each.

```
3+2-5*7
```

Infix Transformation to Postfix

Algorithm:

1. Create an empty stack and an empty output string
2. Scan infix input string from left to right
3. If the current input token is an operand, append it to the output string.
4. If the current input token is an operator, pop off all operators that have equal or higher precedence and append them to the output string; push the current operator onto the stack.
5. If the current input token is '(', push it onto the stack.
6. If the current input token is ')', pop off all operators and append them to the output string until a '(' is popped; discard the '('.
7. If the end of the input string is found, pop all operators and append them to the output string.

Infix to Postfix Example

Example:

A * B + C becomes **A B * C +**

	current symbol	operator stack	postfix string
1	A		A
2	*	*	A
3	B	*	A B
4	+	+	A B * //pop and print '*' before pushing '+'
5	C	+	A B * C
6			A B * C +

Infix to Postfix Example

Example:

A * (B + C) becomes **A B C + ***

	current symbol	operator stack	postfix string
1	A		A
2	*	*	A
3	(	*(	A B
4	B	*(	A B
5	+	*(+	A B
6	C	*(+	A B C
7	)	*	A B C +
8			A B C + *

Infix to Postfix Example

Example:

A * B ^ C + D becomes **A B C ^ * D +**

	current symbol	operator stack	postfix string	
1	A		A	
2	*	*	A	
3	B	*	A B	
4	^	*^	A B	// ^ has higher precedence than *
5	C	*^	A B C	
6	+	+	A B C ^ *	
7	D	+	A B C ^ * D	
8			A B C ^ * D +	

Practice Example

Now, try this expression. Show both stacks.

`A * B - C / D ^ (E - F / G) / H`

Homework

Consider the following expression.

```
A - B - C / D * ((E + ((F - G) / H)) * I) / J
```

1. Use stack to convert the expression to postfix. Show all work.

2. Use stack to evaluate the postfix given the following values:

   A = 10; B = 1; C = 8; D = 2; E = 4;

   F = 4; G = 2; H = 2; I = 6; J = 5;

   The final value on the stack for this part should be -15.

End of Slides