CMPT 225

Stacks

Stacks

- A stack is a data structure that only allows items to be inserted and removed at ***one end***
  - We call this end the ***top*** of the stack
  - The other end is called the bottom
- Access to other items in the stack is not allowed
- A **LIFO** (Last In First Out) data structure

Using a Stack

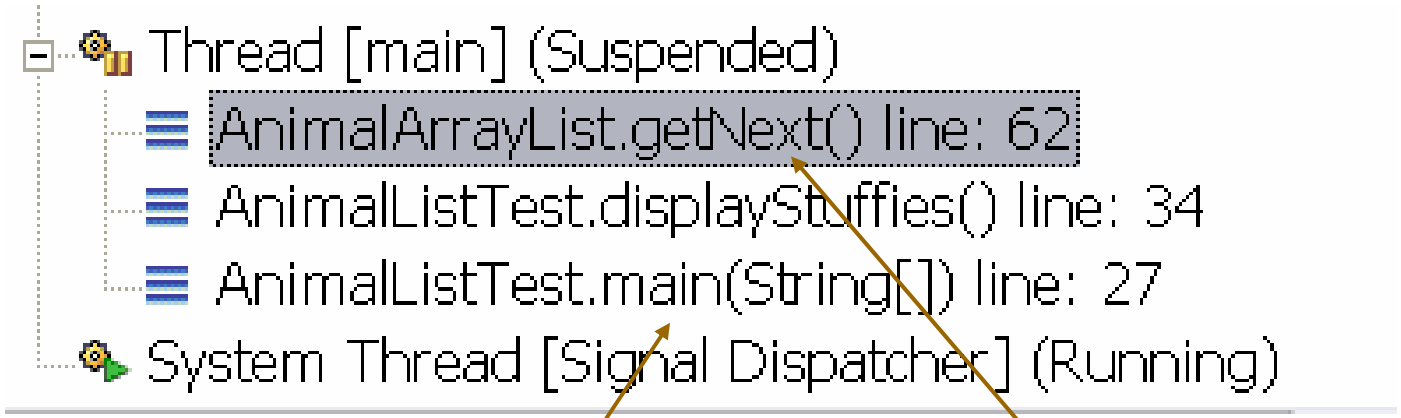


What Are Stacks Used For?

- Most programming languages use a “call stack” to implement function calling
  - When a method is called, its line number and other useful information are pushed (inserted) on the call stack
  - When a method ends, it is popped (removed) from the call stack and execution restarts at the indicated line number in the method that is now at the top of the stack

The Call Stack

This is a display of the call stack (from the Eclipse Debug window)



Bottom of the stack: least recently called method

Top of the stack: most recently called method.

Call Stacks and Recursion

- A call stack is what makes recursion possible
- Stacks are also important when traversing tree data structures
  - They enable us to “backtrack” through the tree
  - We’ll see more about this later in the course

Developing and ADT during the design of a solution.

- Suppose you want to write an editor program that allows the user to correct his/her mistakes using a backspace.
- Each backspace erases the previous.
- E.g
  - If you type: abcc←ddde ← ← ← ef ← ←fg
  - The corrected input is: abcdefg

- You need to store the input date in an ADT
- A first attempt to solve the editor problem:

```
while(not end of the line){
   Read a new character ch
   if(ch is not a <- )
        add ch to the ADT
   else
        remove from the ADT the item added most recently
}
```

```
ab <- b <- <- a
```

a	
b	
<-	
b	
<-	
<-	
a	

- There will be a problem if you type a backspace when the ADT is empty
- A revised solution.

```
while(not end of the line){
   Read a new character ch
   if(ch is not a <- )
        add ch to the ADT
   else if (ADT is not empty)
        remove from the ADT the item added most recently
   else
        ignore the <-
}
```

- The solution to the editor problem suggests that the ADT should implement the following operations:
  - Add a new item to ADT
  - Remove from the ADT the last placed item.
  - Check whether the ADT is empty.
- This ADT is known as a *Stack*.

Stack Operations

- A stack should implement (at least) these operations:
  - **push** – insert an item at the top of the stack
  - **pop** – remove and return the top item
  - **peek** – return the top item (without removing it)
  - **isEmpty** – returns true when the stack is empty
- These operations should be performed efficiently – in O(1) time

Stack analogy

- A stack
  - Last-in, first-out (LIFO) property
    - The last item placed on the stack will be the first item removed
  - Analogy
    - A stack of dishes in a cafeteria
- A queue on the other hand is a FIFO (First in, first out) structure
    - The first item added is the first item to be removed

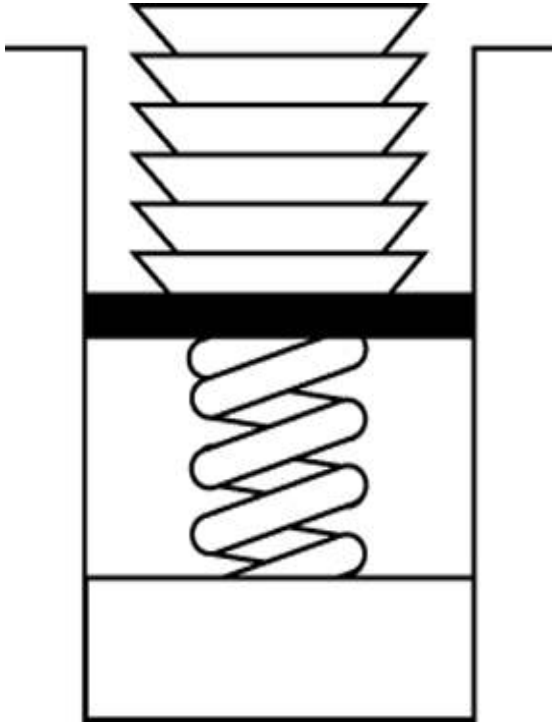


Figure 7-1

Stack of cafeteria dishes

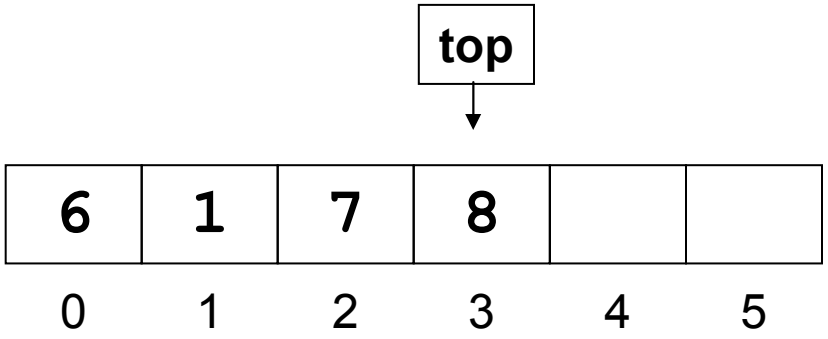
Stack Implementation

- The stack ADT can be implemented using a variety of data structures. We will look at two:
  - Arrays
  - Linked Lists
- Both implementations must implement all the stack operations

Stack: Array Implementation

- If an array is used to implement a stack what is a good index for the top item?
  - Is it position 0? O(n) to insert or remove
  - Is it position n-1? O(1) to insert or remove
- Note that push and pop must both work in O(1) time as stacks are usually assumed to be extremely fast
- The index of the top of the stack is the number of items in the stack - 1

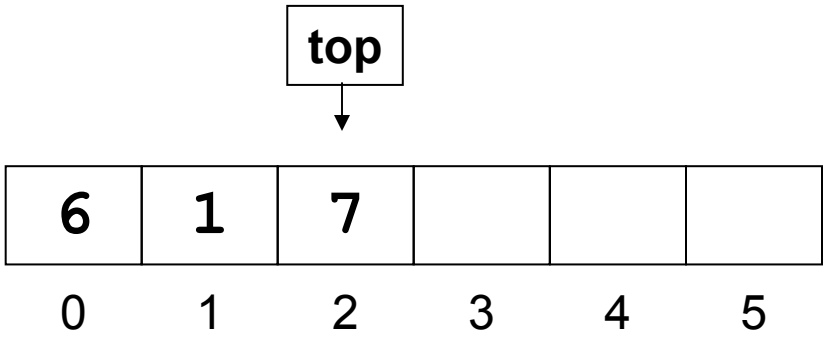
Array Stack Example



index of **top** is
current size – 1

```
//Java Code
Stack st = new Stack();
st.push(6); //top = 0
st.push(1); //top = 1
st.push(7); //top = 2
st.push(8); //top = 3
```

Array Stack Example



index of **top** is
current size – 1

```
//Java Code
Stack st = new Stack();
st.push(6); //top = 0
st.push(1); //top = 1
st.push(7); //top = 2
st.push(8); //top = 3
st.pop(); //top = 2
```

StackInterface.java

```
public interface StackInterface {
   public boolean isEmpty();
   // Determines whether the stack is empty.
   // Precondition: None.
   // Postcondition: Returns true if the stack is empty;
   // otherwise returns false.

   public void popAll();
   // Removes all the items from the stack.
   // Precondition: None.
   // Postcondition: Stack is empty.

   public void push(Object newItem) throws StackException;
   // Adds an item to the top of a stack.
   // Precondition: newItem is the item to be added.
   // Postcondition: If insertion is successful, newItem
   // is on the top of the stack.
   // Exception: Some implementations may throw
   // StackException when newItem cannot be placed on
   // the stack.

   public Object pop() throws StackException;
   // Removes the top of a stack.
   // Precondition: None.
   // Postcondition: If the stack is not empty, the item
   // that was added most recently is removed from the
   // stack and returned.
   // Exception: Throws StackException if the stack is
   // empty.

   public Object peek() throws StackException;
   // Retrieves the top of a stack.
   // Precondition: None.
   // Postcondition: If the stack is not empty, the item
   // that was added most recently is returned. The
   // stack is unchanged.
   // Exception: Throws StackException if the stack is
   // empty.
}  // end StackInterface
```

```
public class StackArrayBased implements StackInterface {

  final int MAX_STACK = 50;  // maximum size of stack
  private Object items[];
  private int top; //indicates the index of the top element

  public StackArrayBased() {
    items = new Object[MAX_STACK];
    top = -1;
  }

  public boolean isEmpty() {
    return top < 0;
  }

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

```
  public void push(Object newItem) throws StackException {
    if (!isFull()) {
      items[++top] = newItem;
    }
    else {
      throw new StackException("StackException on " + "push: stack full");
    }
  }

  public Object pop() throws StackException {
    if (!isEmpty()) {
      return items[top--];
    }
    else {
      throw new StackException("StackException on " + "pop: stack empty");
    }
  }

  public Object peek() throws StackException {
    if (!isEmpty()) {
      return items[top];
    }
    else {
      throw new StackException("Stack exception on " + "peek - stack empty");
    }
}
```

Array Implementation Summary

- Easy to implement a stack with an array
- `push` and `pop` can be performed in O(1) time
- But the implementation is subject to the limitation of arrays that the size must be initially specified because
  - The array size must be known when the array is created and is fixed, so that the right amount of memory can be reserved
  - Once the array is full no new items can be inserted
- If the maximum size of the stack is not known (or is much larger than the expected size) a dynamic array can be used
  - But occasionally push will take O(*n*) time

Stack: Linked List Implementation

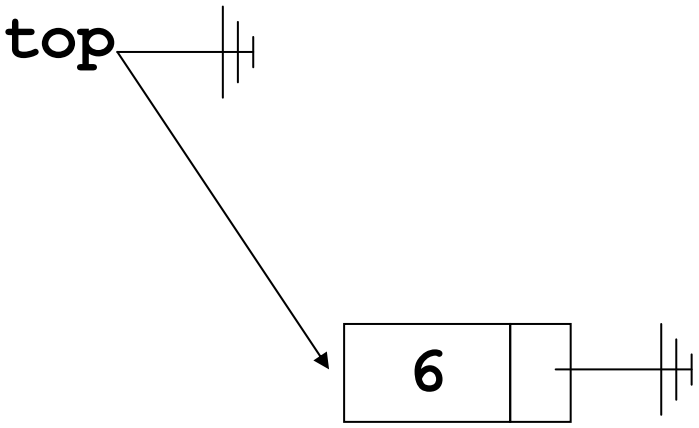
- Push and pop at the head of the list
  - New nodes should be inserted at the front of the list, so that they become the top of the stack
  - Nodes are removed from the front (top) of the list
- Straight-forward linked list implementation
  - `push` and `pop` can be implemented fairly easily, e.g. assuming that `top` is a reference to the node at the front of the list

```
public void push(int x){
    // Make a new node whose next reference is
    // the existing list
    Node newNode = new Node(x, top);
    top = newNode; // top points to new node
}
public int pop(){
    if(!isEmpty()){
            Node temp=top;
            top=top.getNext(); // top points to the next node
            return temp.getItem();
    }
}
```

List Stack Example

Java Code

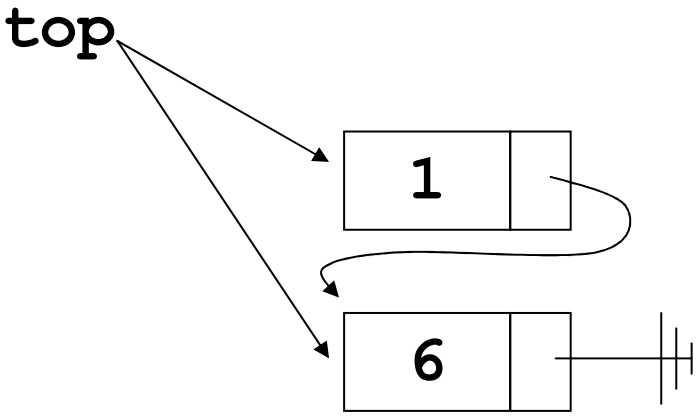
```
Stack st = new Stack();
st.push(6);
```



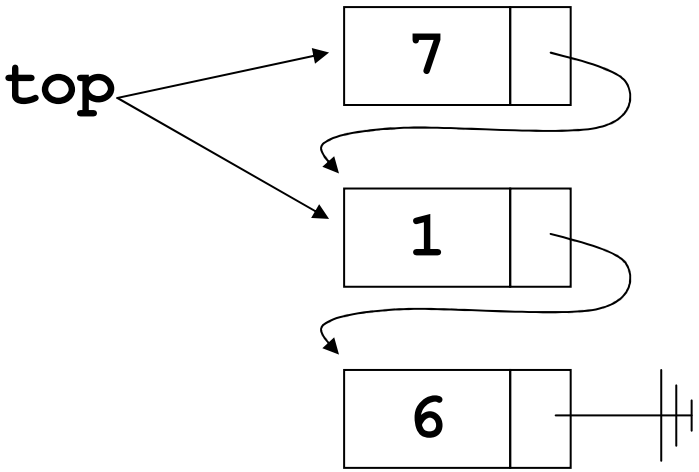
List Stack Example

Java Code

```
Stack st = new Stack();
st.push(6);
st.push(1);
```



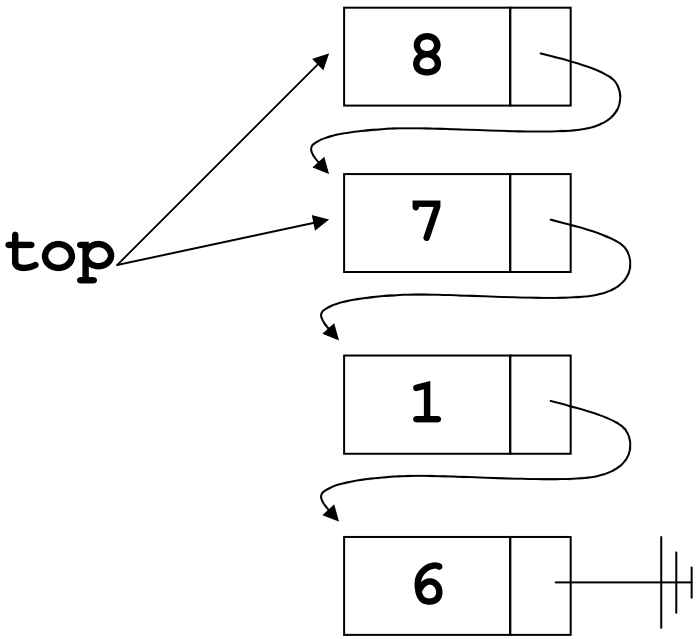
List Stack Example



Java Code

```
Stack st = new Stack();
st.push(6);
st.push(1);
st.push(7);
```

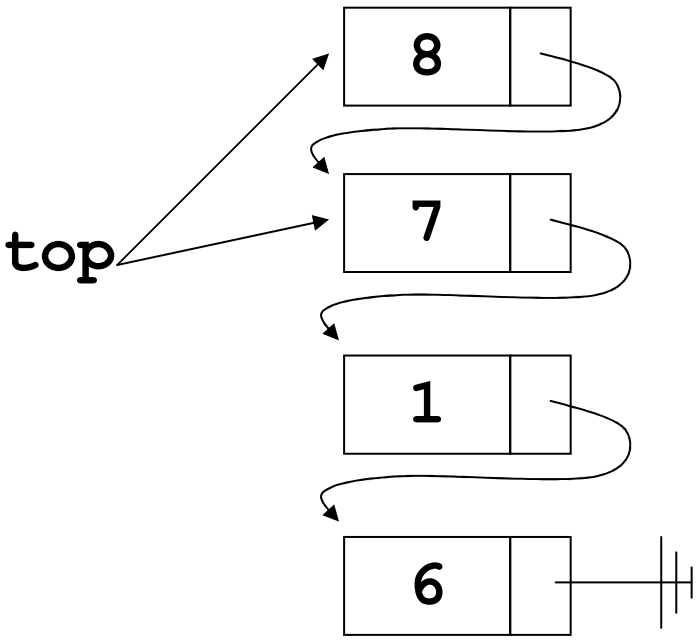
List Stack Example



Java Code

```
Stack st = new Stack();
st.push(6);
st.push(1);
st.push(7);
st.push(8);
```

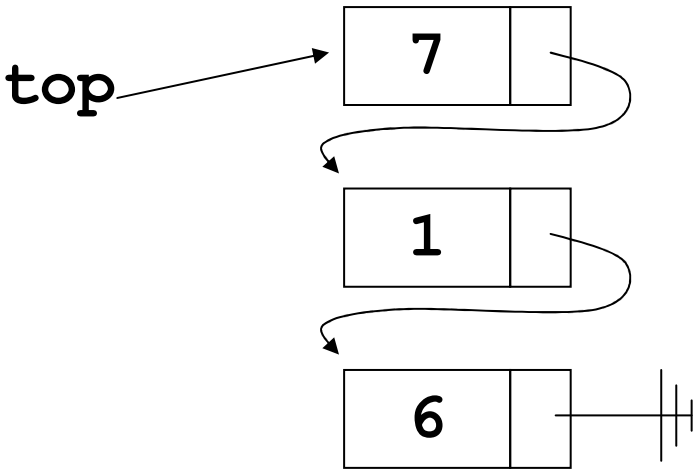
List Stack Example



Java Code

```
Stack st = new Stack();
st.push(6);
st.push(1);
st.push(7);
st.push(8);
st.pop();
```

List Stack Example



Java Code

```
Stack st = new Stack();
st.push(6);
st.push(1);
st.push(7);
st.push(8);
st.pop();
```

Stack: ADT List Implementation

- Push() and pop() either at the beginning or at the end of ADT List
  - at the beginning:

```
public void push(Object newItem) {
  list.add(1, newItem);
}  // end push
public Object pop() {
  Object temp = list.get(1);
  list.remove(1);
  return temp;
}  // end pop
```

Stack: ADT List Implementation

- Push() and pop() either at the beginning or at the end of ADT List
  - at the end:

```
public void push(Object newItem) {
  list.add(list.size()+1, newItem);
}  // end push
public Object pop() {
  Object temp = list.get(list.size());
  list.remove(list.size());
  return temp;
}  // end pop
```

Stack: ADT List Implementation

- Push() and pop() either at the beginning or at the end of ADT List
- Efficiency depends on implementation of ADT List (not guaranteed)
- On other hand: it was very fast to implement (code is easy, unlikely that errors were introduced when coding).

Applications of Stacks

- Call stack (recursion).
- Searching networks, traversing trees (keeping a track where we are).

Examples:

- Checking balanced expressions
- Recognizing palindromes
- Evaluating algebraic expressions

Simple Applications of the ADT Stack: Checking for Balanced Braces

- A stack can be used to verify whether a program contains balanced braces
  - An example of balanced braces

    ```
    abc{defg{ijk}{l{mn}}op}qr
    ```

  - An example of unbalanced braces

    ```
    abc{def}}{ghij{kl}m
    ```

    ```
    abc{def}{ghij{kl}m
    ```

Checking for Balanced Braces

- Requirements for balanced braces
  - Each time you encounter a “}”, it matches an already encountered “{”
  - When you reach the end of the string, you have matched each “{”
- If you process the text from left to right, each time you see a “}” it must be matched with the ***last seen*** “{“.
- This suggests that a stack is an appropriate ADT for solving the problem.

Checking for Balanced Braces

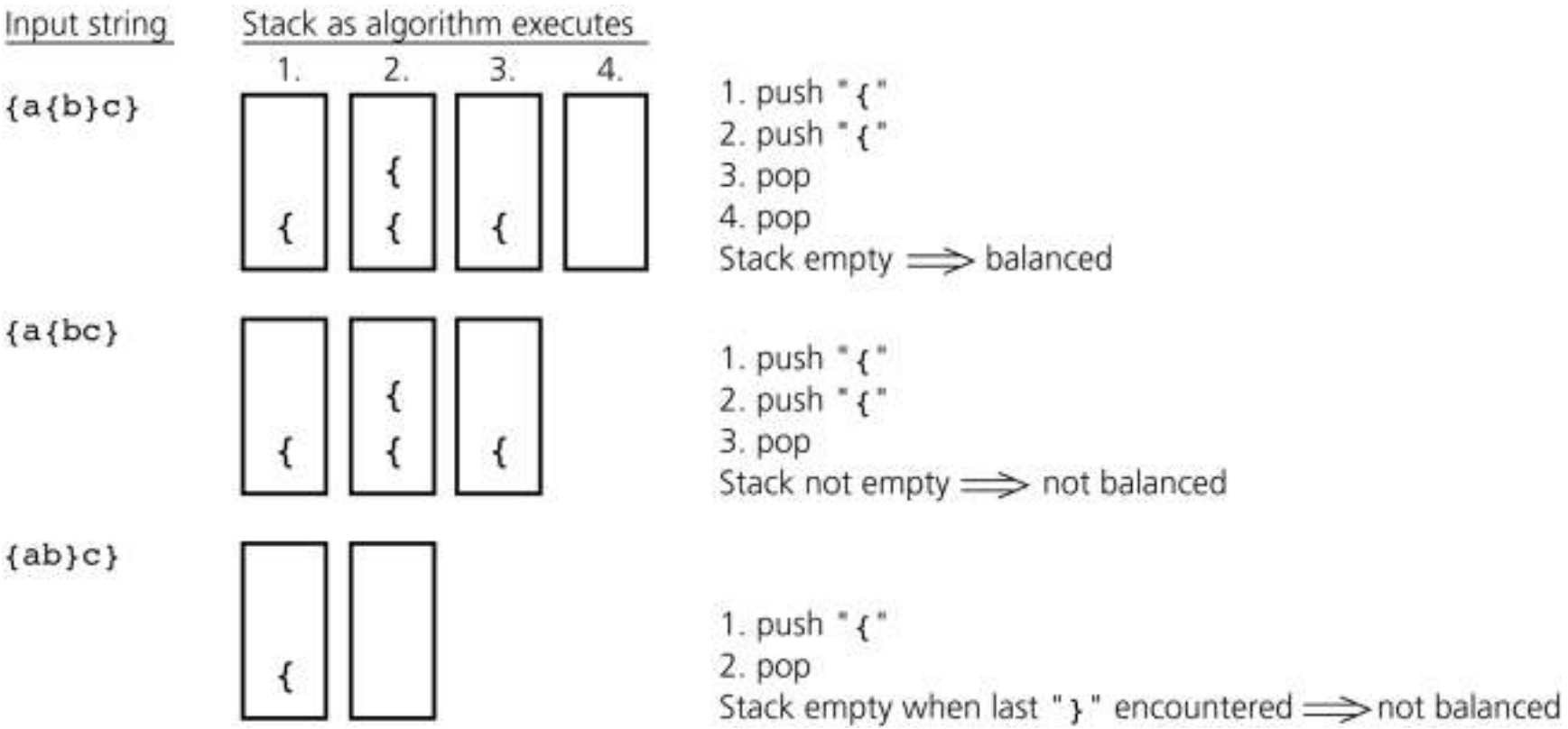


Figure 7-3

Traces of the algorithm that checks for balanced braces

A solution for balanced braces using the stack.

```
aStack.createStack();
balancedSoFar = true;
while (balancedSoFar and not at the of the string){
   ch=read next character;
   if (ch == '{')
        aStack.push('{');
   else if (ch == '}')
        if (!aStack.isEmpty())
               aStack.pop();
        else
               balancedSoFar = false;
}
if (balancedSoFar and aStack.isEmpty())
   string has balanced braces
else
   the string does not have balanced braces.
```

Algebraic operations.

- The algebraic expressions you learned about in school are called infix expressions
  - The term infix indicates that every binary operator appears between its operands. a+b , a+b*c
  - This convention necessitates associatively rules, precedence rules, and the use of parenthesis to avoid ambiguity.
    - a+b*c => (a+b)*c or a + b*c //precedence rule
    - a/b*c => (a/b)* c or a/(b*c) //associatively rule.
- Two alternative to the traditional infix convention
  - Postfix ab +
  - Prefix +ab

Prefix and Postfix

- Prefix
  - The operator appears before the operands.
    - a+(b*c) => a+(*bc) => +a*bc
    - (a+b)*c => (+ab)*c => *+abc
- Postfix
  - The operator appears before the operands.
    - a+(b*c) => a+(bc*) => abc*+
    - (a+b)*c => (ab+)*c => ab+c*
- The prefix and postfix convention do not need associatively rules, precedence rules, and the use of parenthesis.

Evaluating Postfix Expressions

- A postfix calculator
  - Requires you to enter postfix expressions
    - Example: 2 3 4 + *
  - An operator in a postfix expr applies to two operands that immediately precede it.
    - This suggests that stack is an appropriate ADT to solve this problem.
  - When an operand is entered, the calculator
    - Pushes it onto a stack
  - When an operator is entered, the calculator
    - Applies it to the top two operands of the stack
    - Pops the operands from the stack
    - Pushes the result of the operation on the stack

Evaluating Postfix Expressions

Key entered	Calculator action		Stack (bottom to top)
2	push 2		2
3	push 3		2 3
4	push 4		2 3 4
+	operand2 = pop stack	(4)	2 3
	operand1 = pop stack	(3)	2
	result = operand1 + operand2	(7)	2
	push result		2 7
*	operand2 = pop stack	(7)	2
	operand1 = pop stack	(2)	
	result = operand1 * operand2	(14)	
	push result		14

Figure 7-8

The action of a postfix calculator when evaluating the expression 2 * (3 + 4)

Evaluating Postfix Expressions

- To evaluate a postfix expression which is entered as a string of characters
  - Simplifying assumptions
    - The string is a syntactically correct postfix expression
    - No unary operators are present
    - No exponentiation operators are present
    - Operands are single lowercase letters that represent integer values

Evaluating Postfix Expressions

- Pseudo code:

```
int evaluate(String expression)
{
  Stack stack=new Stack(); // creaty empty stack
  while (true) {
    String c=expression.getNextItem();
    if (c==ENDOFLINE)
      return stack.pop();

    if (c is operand)
      stack.push(c);
    else { // operation
      int operand2=stack.pop();
      int operand1=stack.pop();
      stack.push(execute(c,operand1,operand2));
    }
  }
}
```