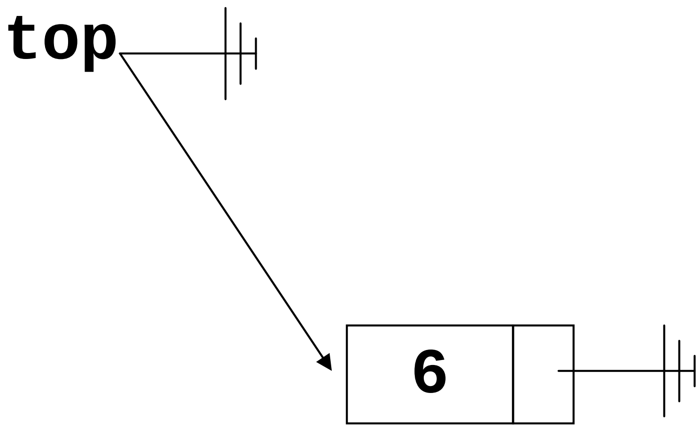
Stack: Linked List Implementation

- 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 head is a reference to the node at the front of the list

```java
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
}
```

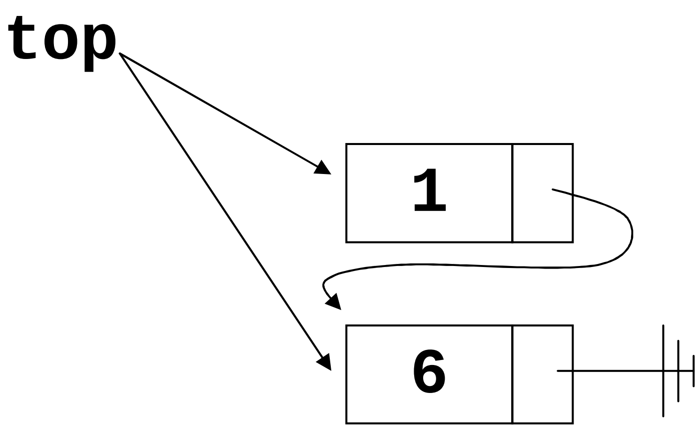
List Stack Example

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

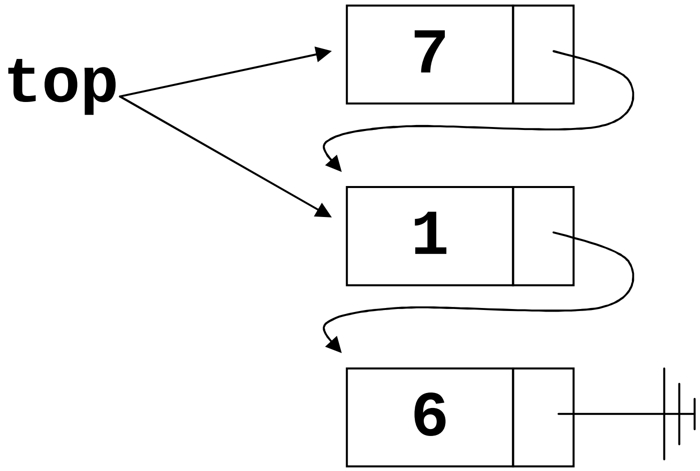


List Stack Example

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



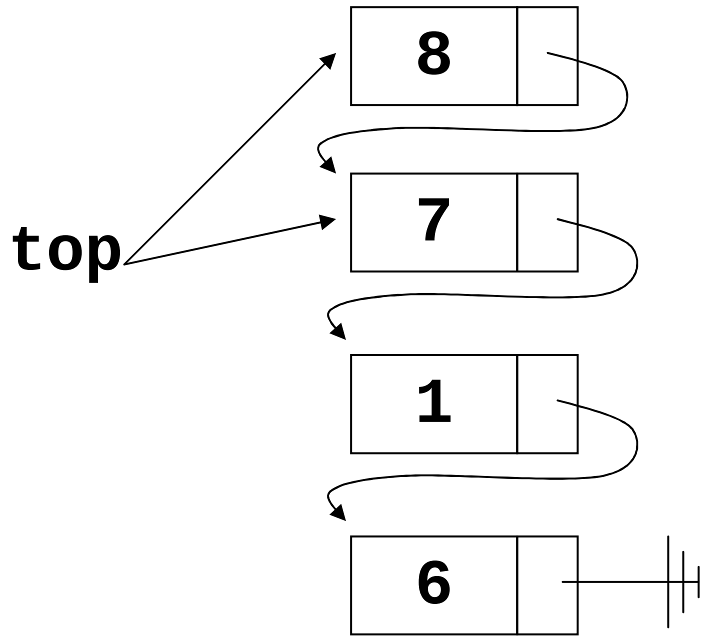
List Stack Example



Java Code

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

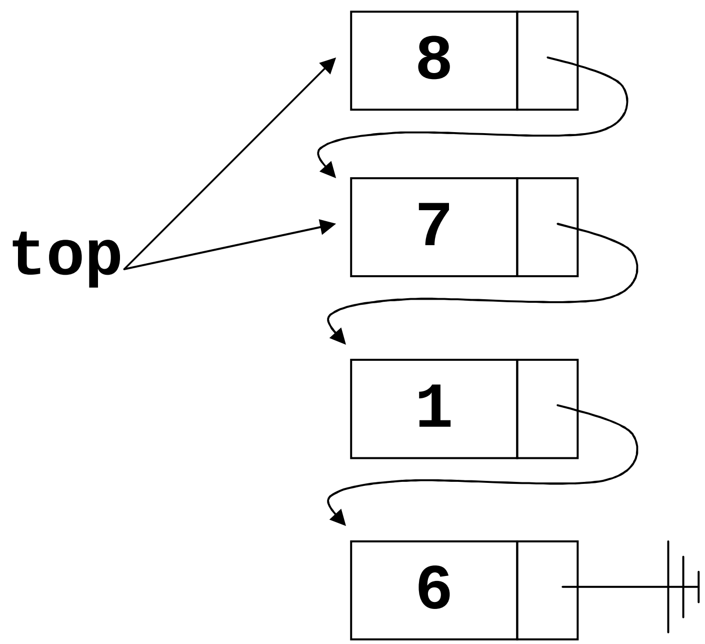
List Stack Example



Java Code

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

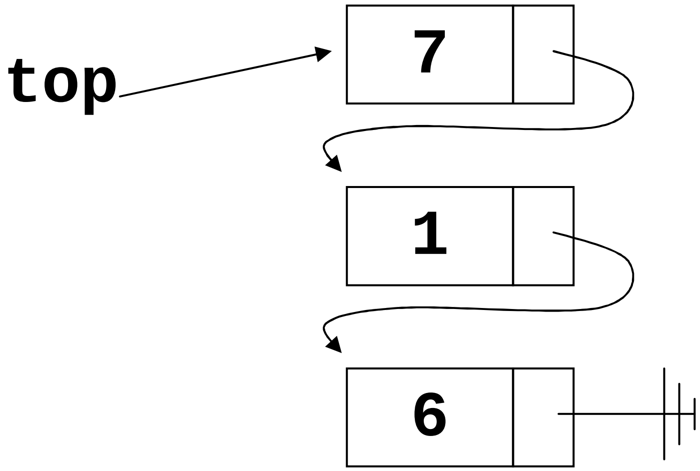
List Stack Example



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 “{”

Checking for Balanced Braces

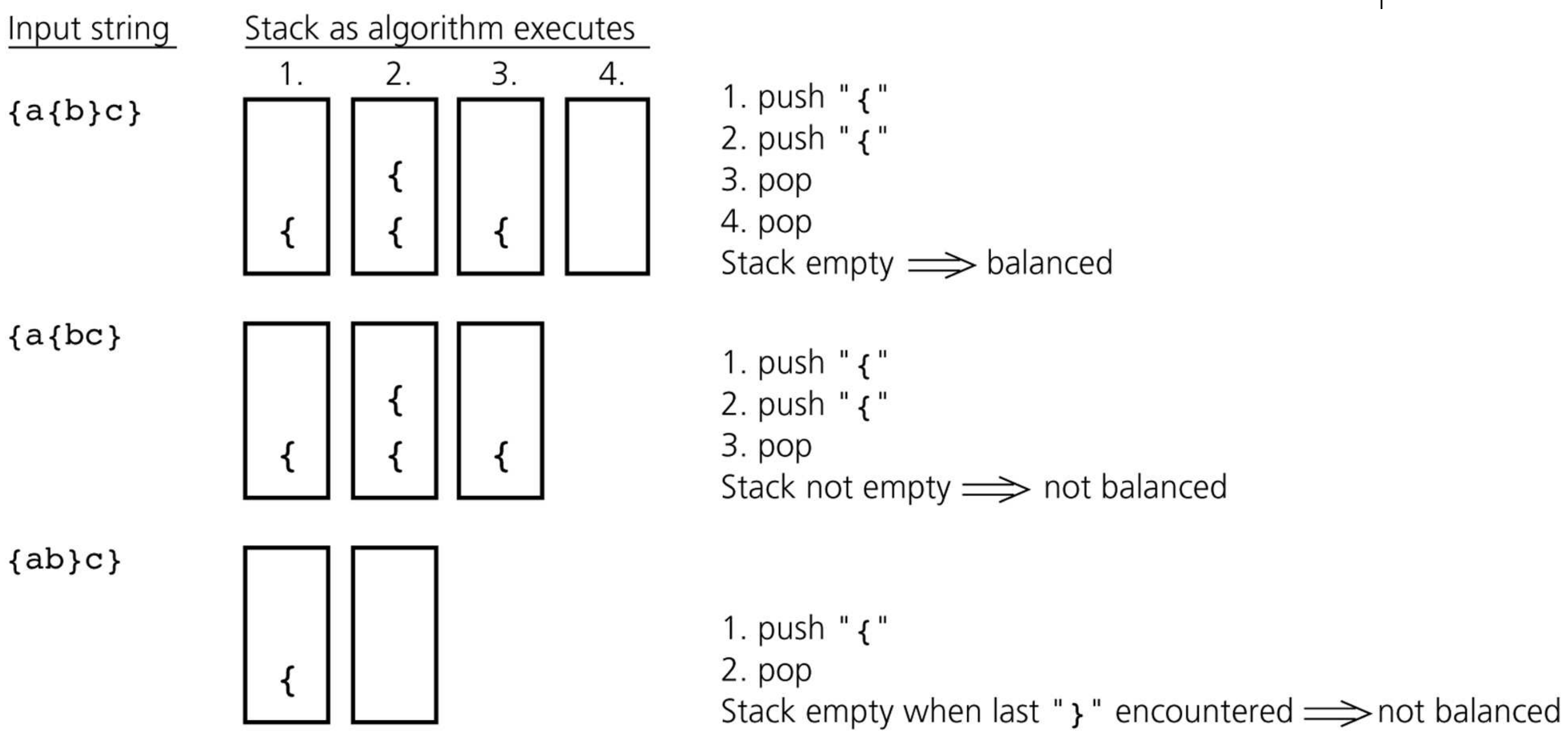


Figure 7-3

Traces of the algorithm that checks for balanced braces

Evaluating Postfix Expressions

- A postfix (reverse Polish logic) calculator
  - Requires you to enter postfix expressions
    - Example: 2 3 4 + *
  - 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

- 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));
    }
  }
}
```

Queues

- A queue is a data structure that only allows items to be inserted at the end and removed from the front
- “Queue” is the British word for a line (or line-up)
- Queues are **FIFO** (First In First Out) data structures – “fair” data structures

Using a Queue

What Can You Use a Queue For?

- Processing inputs and outputs to screen (console)
- Server requests
  - Instant messaging servers queue up incoming messages
  - Database requests
- Print queues
  - One printer for dozens of computers
- Operating systems use queues to schedule CPU jobs
- Simulations

Queue Operations

- A queue should implement (at least) these operations:
  - **enqueue** – insert an item at the back of the queue
  - **dequeue** – remove an item from the front
  - **peek** – return the item at the front of the queue without removing it
- Like stacks it is assumed that these operations will be implemented efficiently
  - That is, in constant time

Queue: Array Implementation

- First consider using an array as the underlying structure for a queue, one plan would be to
  - Make the back of the queue the current size of the queue (i.e., the number of elements stored)
  - Make the front of the queue index 0
  - Inserting an item can be performed in constant time
  - But removing an item would require shifting all elements in the queue to the left which is **too slow**!
- **Therefore we need to find another way**

An Array-Based Implementation

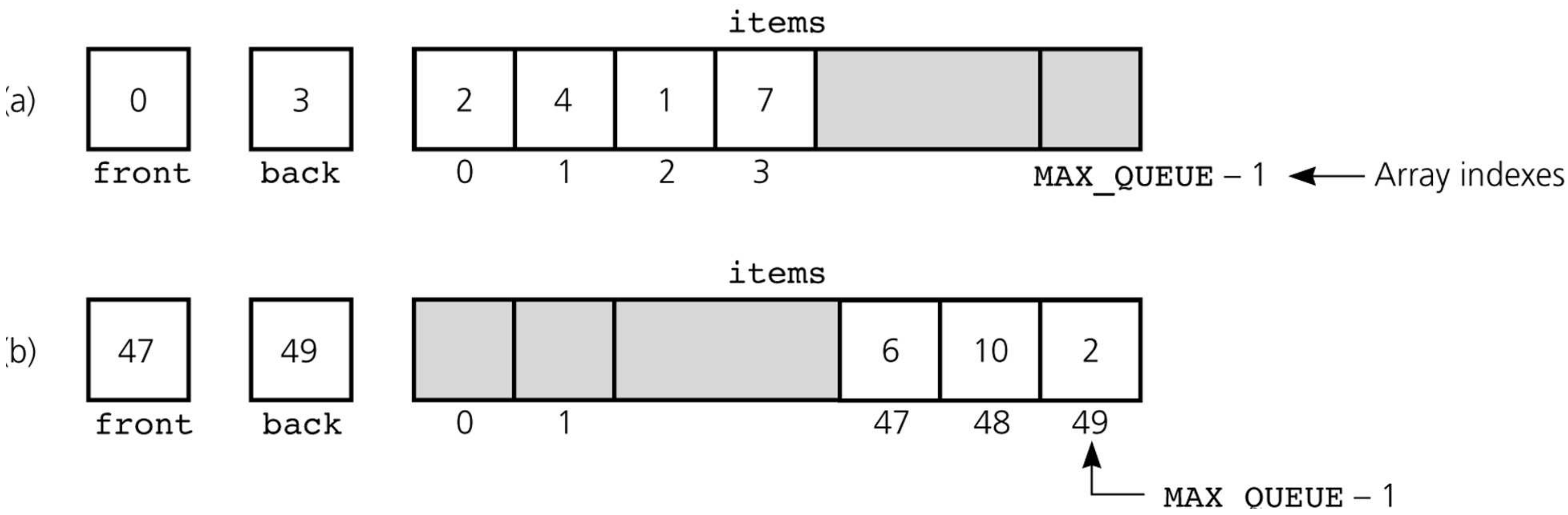


Figure 8-8

a) A naive array-based implementation of a queue; b) rightward drift can cause the queue to appear full