Introduction to Data Structures

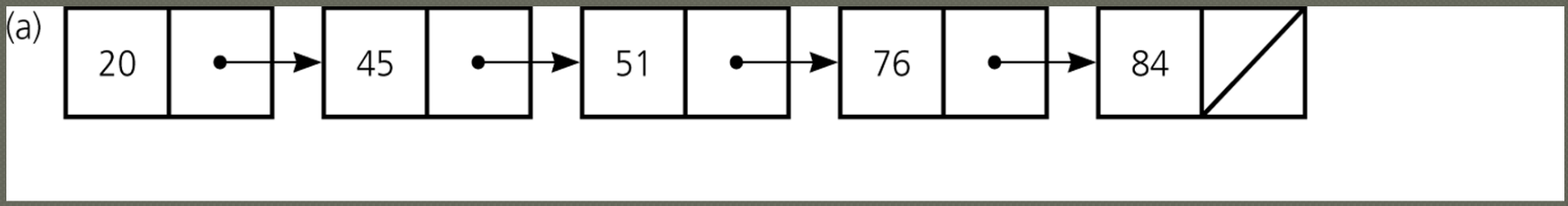
Chapter 15

Linked List

- A data structure used for collecting a sequence of objects
- Easy to add and remove elements
- Example
  - Maintaining a list of employees
  - Maintained by name
- Problem with storing in array
  - Shifting when inserting or deleting
- Answer
  - Inserting or deleting in linked list does not require

Linked List

- Consists of a number of notes, each of which has a reference to the next node.



Linked List

- Visiting elements
  - Sequential order is effective
  - Random order is not effective

Linked List

- Sequence of Nodes
  - Node
    - Value or object
    - Reference to next node
  - Remove a Node
    - Change the reference
  - How do I get to the node
    - List iterator
    - Goes to 1st node
    - Goes to node it is pointing to
    - Continues till you get to the node

Java Class Linked List

- Generic class
- Must use < >
- Put the kind of object in < >
- Use methods to add to beginning and end
- Traversing the linked list
  - List Iterator

Creating a Linked List Type String

- Type String

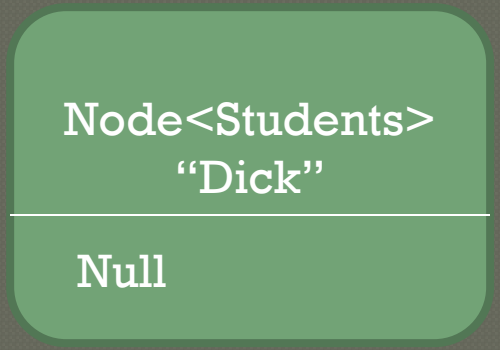
```
LinkedList<String> students = new
                    LinkedList<String>();

Type BankAccounts
LinkedList<BankAccount> students = new
                    LinkedList<BankAccount>();
```

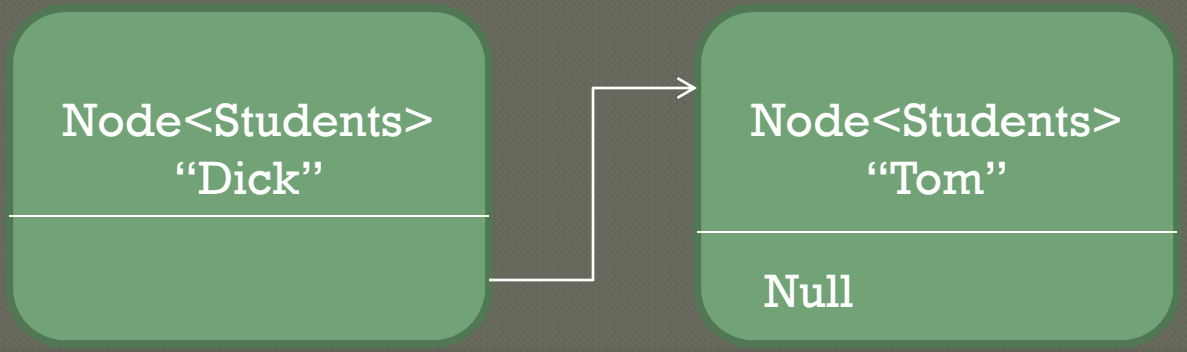
Adding Nodes

- Add first node

  Students.addLast("Dick");



- Add another node after "Dick"
  - Student.addLast("Tom");



LinkedList Class Methods

- void addFirst(element)
- void addLast(element)
- e getFirst( )
- e getLast( )
- e removeFirst( )
- e removeLast( )

- Notice no way to move within list

ListIterator

- listIterator is a method of the LinkedList class.
- Create

LinkedIterator<String>
iterator = students.listIterator();

- Purpose: to iterate through a list of and visit each element.

ListIterator

- Begins before the first element.
- Move Forward

  iterator.next( );
  - Returns the element that the iterator is passing.
  - The type to be returned will depend on the type specified in the <>
- Check to see if there is another element

  if (iterator.hasNext());

ListIterator

- Add
  - Iterator.add("Kathy");
  - Added after iterator position
  - Moves position
- Remove
  - Iterator.remove()
  - Removes the object that was returned by last call to next or previous

ListIterator

- Careful using the remove method
  - Can be called only once after calling next()
  - Cannot call remove immediately after a call to add.

LinkedList Class

- Nodes store tow links:
  - One to next element
  - One to previous element
- Doubly linked list
- For listIterator method
  - Has previous
  - previous

ListIterator

- Traverse entire list
  - For each loop

```
For (String name: students)
{
    String studentName = name.getName();
    System.out.println(studentName);
}
```

Under the Hood

- Node Class

```
public class Node
{
   public Object data;
   public Node next;
}
```

  - data is the object we want to add
  - next is the location of the next node

Under the Hood

- Create First or Head
  - Pointer to initial element in the linked list
  - Initially it will be blank
  - Would go in constructor of linked list

```
public LinkedList ( )
{
    first = null;
}
```

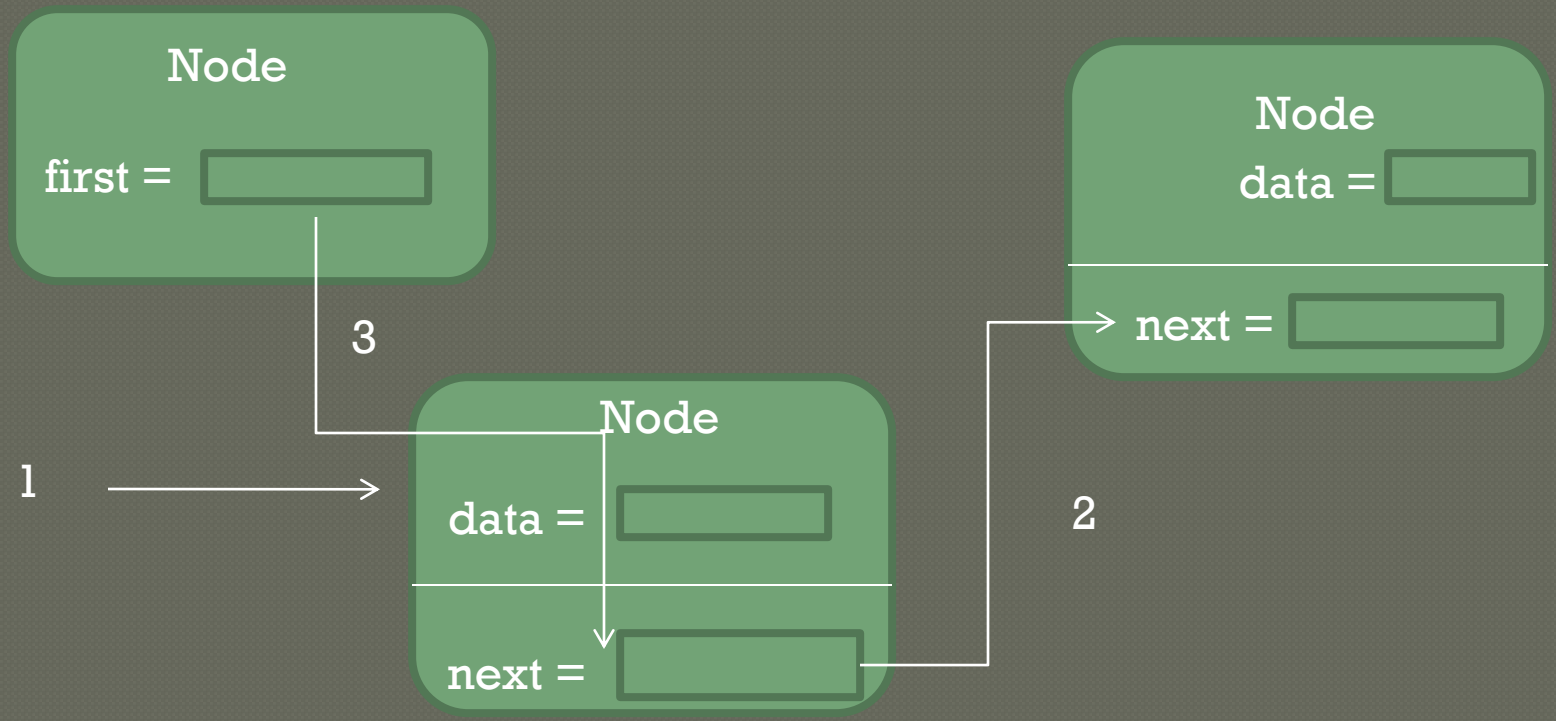
Under the Hood

- Method to Retrieve First/Head

```
 public Object getFirst()
{
   if (first == null)
         throw new NoSuchElement Exception();
   return first;
}
```

Under the Hood

- Method to Add First/Head
- Special case
- New list pointer must point to it



Under the Hood

- Method to Add First/Head

```
public void addFirst(Object element)
{
        Node newNode = newNode();
        newNode.data = element;
        newNode.next = first;
        first = newNode;
}
```

Under the Hood

- Method to Remove first element in the list

```
public void removeFirst()
{
   if (first = = null)
         throw new NoSuchElementException();
   Object element = first.data;
   first = first.next();
   return element;
}
```

Implementing the Iterator Class

- Standard library implements 9 methods we will only implement only 5
- Iterator class
  - Inner class to LinkedList
  - Has access to private member of LinkedList
  - Has access to first and the private Node class

Method to define the listIterator

```
public ListIterator listIterator()
{
   return new LinkedListIterator();
}
```

Create the LinkedListIterator

```
private class LinkedListIterator
                implements ListIterator
{
   public LinkedListIterator()
   {
         position = null;
         previous = null;
   }
private Node position;
private Node previous;
……
}
```

Next Method

```
// position is the lsat visited node.
private class LinkedListIterator
        implement ListIterator
{
  .....
    public Object next()
    {
    if(!hasNext())
        throw new NoSuch ElementExceeption
    previous = position; // remember for remove
    if (position == null)
        position = first;
    else
        position = position.next;
    return position.data;
    }
```

hasNext Method

```
private class LinkedListIterator
        implement ListIterator
{
   ……
   public boolean hasNext()
   {
        // check for no element after current
        if (position == null)
                return first ! = null;
        else
                return position.next !=null;
```

Remove Method

```
public void remove()
{
        if (previous == position)
                throw new IllegalStateException();
        if (postion == first)
        {
                removeFirst();
        }
        else
        {
                previous.next = position.next;
        }
        position = previous;
}
```

Remember: position points to the last visited node.
previous points to the last node before that.

Set Method

```
public void set(Object element)
   {
        if (previous == position)
                throw new IllegalStateException();
        position.data = element;
   }
```

Add Method

```
public void add(Object element)
   {
        if (position == null)
        {
                addFirst(element);
                postion = first;
        }
        else
        {
                Node newNode = new Node();
                newNode.data = element;
                newNode.next = postion.next;
                position.next = newNode;
                postion = newNode;
        }
        previous = position;
   }
```

Abstract and Concrete Data Types

- Concrete
  - Sequence of node objects with the links between them.
- Abstract
  - A linked list is an ordered sequence of data items that can be traversed with a iterator
- Abstract Data Type
  - Define the fundamental operations on the data but does not specify an implementation

Define the Fundamental Operations

```
public interface ListIterator
{
   Object next();
   boolean hasNext( );
   void add(Object element);
   void remove( );
   void set(Object element);
   .....
}
```

Stacks and Queues

- Stack
  - Collection of items with "last in first out" retrieval.
  - Can insert or remove at the top only
  - Can insert in middle
- Queue
  - Collection of times with "first in first out" retrieval.
  - Add at the end
  - Remove at the top

Stack

- Stack class in Java Library
- How to use
  - Stack <String> s = new Stack<String>();
  - s.push()
  - s.pop()
  - s.peek()
- Java class uses an array to implement
- Can be easily implemented in a linked list

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
    ```

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

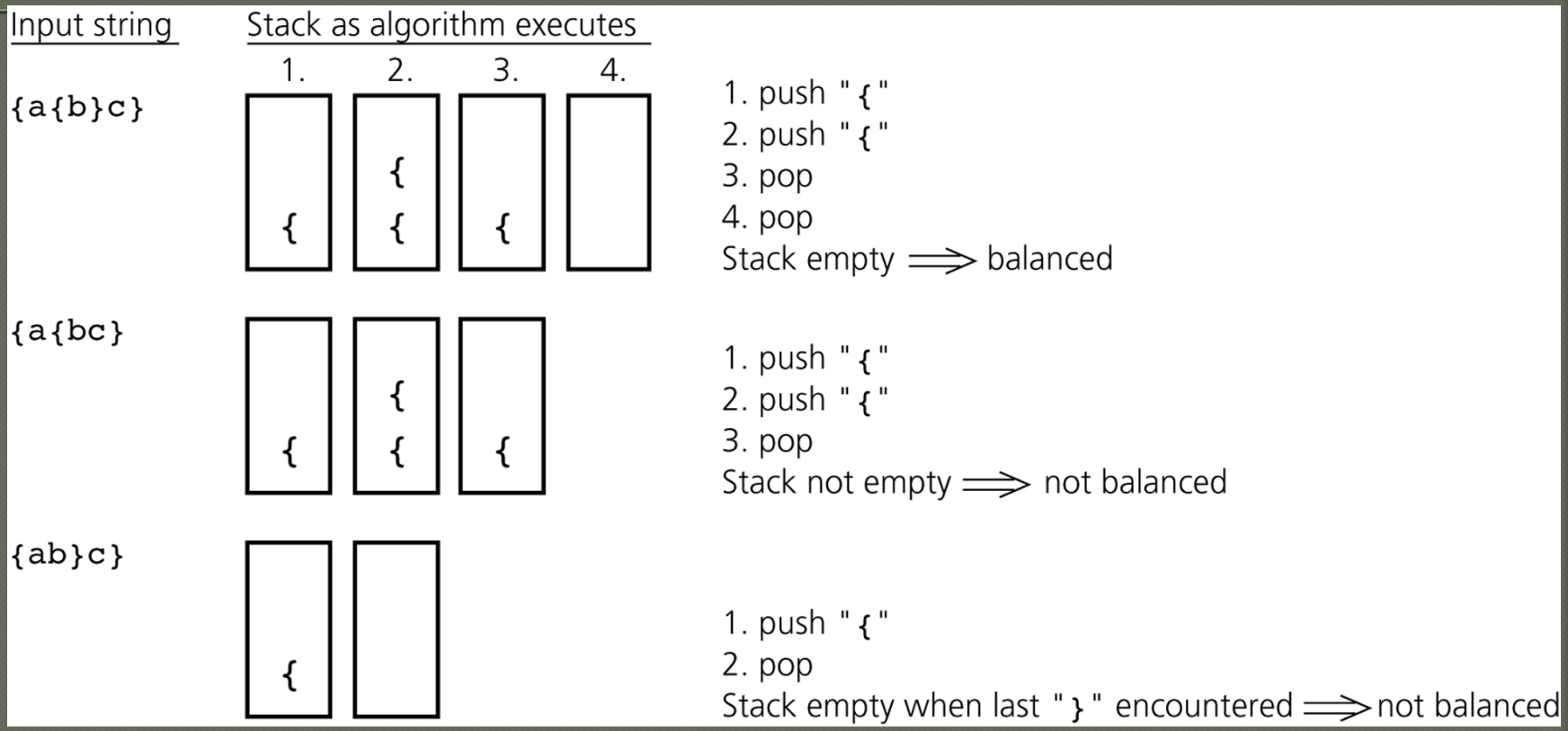


Figure 7.2

Traces of the algorithm that checks for balanced braces

Checking for Balanced Braces

- `StackException`
  - A Java method that implements the balanced-braces algorithm should do one of the following
    - Take precautions to avoid an exception
    - Provide `try` and `catch` blocks to handle a possible exception

Queue

- Queue class in Java Library
- How to use
  - Queue <Integer> q = new Queue<Integer>();
  - q.add() // adds to the tail
  - q.remove() // removes from the top
  - q.peek() //get the head of the queue without removing
- Java class uses an array to implement
- Can be easily implemented in a linked list

Queue

```
import java.util.LinkedList;
public class LinkedListQueue
{
    public LinkedListQueue()
    {
        list = new LinkedList();
    }
    public void add(Object element)
    {
        list.addLast(element);
    }
    public Object remove()
    {
        return list.removeFirst();
    }
    int size()
    {
        return list.size();
    }
    private LinkedList list;
}
```

Application: Algebraic Expressions

- When the ADT stack is used to solve a problem, the use of the ADT's operations should not depend on its implementation
- To evaluate an infix expressions
  - Convert the infix expression to postfix form
  - Evaluate the postfix expression

Infix, Prefix and Postfix Expression

Prefix Notation

- In prefix notation the operator is written before its operands without the use of parentheses or rules of operator precedence.
- The expression (A+B)/(C-D) would be written as /+AB-CD in prefix notation.

Postfix Notation

- Postfix notation is a way of writing algebraic expressions without the use of parentheses or rules of operator precedence.
- The expression (A+B)/(C-D) would be written as AB+CD-/ in postfix notation.

Evaluating Postfix Expressions

- A postfix 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

Converting Infix to Postfix

- http://scriptasylum.com/tutorials/infix_postfix/algorithms/infix-postfix/index.htm

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 7.7

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

to Equivalent Postfix Expressions

- An infix expression can be evaluated by first being converted into an equivalent postfix expression
- Facts about converting from infix to postfix
  - Operands always stay in the same order with respect to one another
  - An operator will move only "to the right" with respect to the operands
  - All parentheses are removed

to Equivalent Postfix Expressions

ch	stack (bottom to top)	postfixExp	
a		a	
–	–	a	
(	– (	a	
b	– (	ab	
+	– (+	ab	
c	– (+	abc	
*	– (+ *	abc	
d	– (+ *	abcd	
)	– (+	abcd*	Move operators
	– (	abcd*+	from stack to
	–	abcd*+	**postfixExp** until " ("
/	– /	abcd*+	
e	– /	abcd*+e	Copy operators from
		abcd*+e/–	stack to **postfixExp**

Figure 7.8

A trace of the algorithm that converts the infix expression *a* - (*b* + *c* * *d*)/*e* to postfix form