Objectives

- Understand the concept of a dynamic data structure.
- Be able to create and use dynamic data structures such as linked lists.
- Grasp the concept of an Abstract Data Type (ADT).
- Understand the stack and queue ADTs.
- Know how to use inheritance to define extensible data structures.
Outline

- Introduction
- The Linked List Data Structure
- Object-Oriented Design: The List Abstract Data Type (ADT)
- The Stack ADT
- The Queue ADT
- From the Java Library: java.util.Stack
- In the Laboratory: Capital Gains

Interview

- A data structure is organized information so that it efficient to access and process.
- An array is a static structure -- it can’t change size once it is created.
- A vector is a dynamic structure -- it can grow in size after creation.
- In this chapter we study several dynamic data structures -- lists, queues, and stacks.

The Linked List Data Structure

- A linked list is based on the concept of a self-referential object -- an object that refers to an object of the same class.
Basic Node Design

- A node in a linked list contains data elements and link elements.

  Stores any kind of object.

  Data elements.

  Link elements.

  Node
  - data: Object
  - next: Node
    - Node
    - Node
    - getNode(): Object
      - getData(): Object
        - getNext(): Node

Example: The PhoneListNode

  Data elements.

  PhoneListNode
  - name: String
  - phone: String
  - next: PhoneListNode
    - PhoneListNode
    - PhoneListNode
    - getName(): String
    - getPhone(): String
    - getNext(): PhoneListNode
    - getNode(): PhoneListNode

A List of PhoneListNodes

  The next links.

  Roger M
  090-997-2918

  Jane M
  090-997-1987

  Stacy K
  090-997-9188
### Implementation: PhoneListNode

```java
public class PhoneListNode {
    private String name;
    private String phone;
    private PhoneListNode next;

    public PhoneListNode(String s1, String s2) {
        name = s1;
        phone = s2;
        next = null;
    }

    public void setNext(PhoneListNode nextPtr) {
        next = nextPtr;
    }

    public PhoneListNode getNext() {
        return next;
    }

    public void setData(String s1, String s2) {
        name = s1;
        phone = s2;
    }

    public String getName() {
        return name;
    }

    public String getData() {
        return name + " " + phone;
    }

    public String toString() {
        return name + " " + phone;
    }
}
```

### The PhoneList Class

- A class to manipulate the PhoneListNode.

#### Design:
These methods require little knowledge of the list.

**Points to first node. Initially set to null.**

### Inserting Nodes into a List

- Insert at the end of the list.

#### Case 1: Empty list

#### Case 2: Nonempty list.
The insert() Method

- If the list is not empty, insert() traverses the list and inserts at the end.

```java
public void insert(PhoneListNode newNode) {
    if (isEmpty()) // Insert at head of list
        head = newNode;
    else {
        PhoneListNode current = head; // Start traversal at head
        while (current.getNext() != null) // While not at last node
            current = current.getNext(); // go to the next node
        current.setNext( newNode ); // Do the insertion
    }
} // insert()
```

**Case 1: Empty list**

**Case 2: Nonempty list.**

Note loop bound.

---

Traversing a List

Traverse: current always points to current node.

New node inserted after the last node.

---

Printing the Nodes of a List

- Traverse the list, printing each node.

```java
public void print() {
    if (isEmpty())
        System.out.println("Phone list is empty");
    PhoneListNode current = head; // Start traversal at head
    while (current != null) { // While not at end of list
        System.out.println(current.toString()); // print node's data
        current = current.getNext(); // go to the next node
    }
} // print()
```

Traverse: current always points to current node.

Note loop bound.

- Terminating Traversal: The loop bound depends on whether you need a reference to the last node after the loop terminates.
Looking up a Node in a List

- Searching: **traverse** the list until the name is found or until the end is reached (not found).

```java
public String getPhone(String name) {
    if (isEmpty())                           // Case 1: empty list
        return "Phone list is empty";
    else {
        PhoneListNode current = head;
        while ((current.getNext() != null) &&
                (!current.getName().equals(name)))
            current = current.getNext();
        if (current.getName().equals(name))  // Case 2: found the name
            return current.getData();
        else                             // Case 3: no such person
            return "Sorry. No entry for " + name;
    } // getPhone()
```

Removing a Node From a List

- To remove, the list must be re-linked.

```java
public String remove(String name) { // Remove an entry by name
    if (isEmpty())                              // Case 1: empty list
        return "Phone list is empty";
    PhoneListNode current = head;
    PhoneListNode previous = null;
    if (current.getName().equals(name)) {       // Case 2: remove first node
        head = current.getNext();
        return "Removed " + current.toString();
    }
    while ((current.getNext() != null) && (!current.getName().equals(name))) {
        previous = current;
        current = current.getNext();
    }
    if (current.getName().equals(name)) {       // Case 3: remove named node
        previous.setNext(current.getNext());
        return "Removed " + current.toString();
    } else
        return "Sorry. No entry for " + name; // Case 4: node not found
} // remove()
```
Testing the List

• Strategy
  – Test insertions.
  – Test lookups, both successes and failures.
  – Test removals at different locations.

```java
public static void main(String argv[]) {
    // Create list and insert some nodes
    PhoneList list = new PhoneList();
    list.insert(new PhoneListNode("Roger M", "997-0020"));
    list.insert(new PhoneListNode("Roger W", "997-0086"));
    list.insert(new PhoneListNode("Rich P", "997-0010"));
    list.insert(new PhoneListNode("Jane M", "997-2101"));
    list.insert(new PhoneListNode("Stacy K", "997-2517"));
    // Test whether insertions worked
    System.out.println("Phone Directory");
    list.print();
    // main()
```

Testing the List (cont)

• Searches:
  // Test whether lookups work
  System.out.println("Looking up numbers by name");
  System.out.println(list.getPhone("Roger M"));
  System.out.println(list.getPhone("Rich P"));
  System.out.println(list.getPhone("Stacy K"));
  System.out.println(list.remove("Rich P"));
  System.out.println("Phone Directory");
  list.print();
• Removals:
  System.out.println("Phone Directory");
  list.print();
  // Test removals, printing list after each removal
  System.out.println(list.remove("Roger M"));
  System.out.println("Phone Directory");
  list.print();
  System.out.println(list.remove("Stacy K"));
  System.out.println("Phone Directory");
  list.print();

OOD: The List Abstract Data Type (ADT)

• Wanted: a generic list structure.
• An Abstract Data Type (ADT) has two parts:
  – the data that are being stored and manipulated,
  – the methods and operations on those data.
• Example: Integer data
  – Data: all the integral values
  – Operations: +, -, /, *, %
• Example: List
  – Data: the objects to be stored.
  – Operations: insert, delete, search.
• Information Hiding: Hide implementation!
The Generic Node Class: Design

```
class Node {
    private Object data;  // Stores any kind of data
    private Node next;

    public Node(Object obj) {  // Constructor
        data = obj;
        next = null;
    }

    public void setNext(Node nextPtr) {
        next = nextPtr;
    }

    public Node getNext() {
        return next;
    }

    // Data access methods
    public void setData(Object obj) {
        data = obj;
    }

    public Object getData() {
        return data;
    }

    public String toString() {
        return data.toString();
    }
}
```

The Generic List Class

```
Two kinds of insertions ...

... and two kinds of removals.
```
The Generic List Class: Implementation

public class List {
private Node head;
public List() {
  head = null;
}
public boolean isEmpty() {
  return head == null;
}
public void print() {}
public void insertAtFront(Object newObj) {
}
public void insertAtRear(Object newObj) {
  if (isEmpty())
    head = new Node(obj);
  else {
    Node current = head;                // Start at head of list
    while (current.getNext() != null)   // Find the end of the list
      current = current.getNext();
    current.setNext(new Node(obj));    // Insert the new Object
  }
}
public Object removeFirst() {
}
public Object removeLast() {
  if (isEmpty())  // Empty list
    return null;
  Node current = head;
  if (current.getNext() == null) { // Singleton list
    head = null;
    return current.getData();
  }
  Node previous = null;            // All other cases
  while (current.getNext() != null) {
    previous = current;
    current = current.getNext();
  }
  previous.setNext(null);
  return current.getData();
}
}

Two kinds of insertions ...

... and kinds of removals.

List Insertion Methods

public void insertAtFront(Object obj) {
  Node newnode = new Node(obj);
  newnode.setNext(head);
  head = newnode;
}

public void insertAtRear(Object obj) {
  if (isEmpty())
    head = new Node(obj);
  else {
    Node current = head;                // Start at head of list
    while (current.getNext() != null)   // Find the end of the list
      current = current.getNext();
    current.setNext(new Node(obj));    // Insert the new Object
  }
}

The inserted object goes into a new Node which is inserted in the list.

List Removal Methods

public Object removeLast() {
  if (isEmpty())  // Empty list
    return null;
  Node current = head;
  if (current.getNext() == null) { // Singleton list
    head = null;
    return current.getData();
  }
  Node previous = null;            // All other cases
  while (current.getNext() != null) {
    previous = current;
    current = current.getNext();
  }
  previous.setNext(null);
  return current.getData();
}

public Object removeFirst() {
  if (isEmpty())
    return null;
  Node first = head;
  head = head.getNext();
  return first.getData();
}

Re-link the list.
Testing the List ADT

```java
public static void main(String argv[]) {
    // Create list and insert heterogeneous nodes
    List list = new List();
    list.insertAtFront(new PhoneRecord("Roger M", "997-0020");
    list.insertAtFront(new Integer(8647));
    list.insertAtFront(new String("Hello World");
    list.insertAtRear(new PhoneRecord("Jane M", "997-2101");
    list.insertAtRear(new PhoneRecord("Stacy K", "997-2517");
    System.out.println("Generic List");
    list.print();
    // Remove objects and print resulting list
    Object o;
    o = list.removeLast();
    System.out.println("Removed "+o.toString());
    System.out.println("Generic List:");
    list.print();
    o = list.removeFirst();
    System.out.println("Removed "+o.toString());
    System.out.println("Generic List:");
    list.print();
}
```

Insert different types of objects.
Note use of toString().

The PhoneRecord Class

• The PhoneRecord stores data for a telephone directory.

```java
PhoneRecord {
    name: String
    phone: String
    PhoneRecord(name: String, phone: String)
    toString(): String
    getName(): String
    getPhone(): String
```

The Stack ADT

• A stack is a list that limits insertions and removals to the front (top) of the list.

• Operations
  - Push: insert an object onto the top of the stack.
  - Pop: remove the top object from the stack.
  - Empty: returns true if the stack is empty.
  - Peek: retrieve the top object without removing it.
The Stack Class

• A Stack is very easy to design as an extension of our generic List structure.

Inheritance: Protected elements are inherited by Stack class.

Abstract Data Type: We limit access to the data to push() and pop().

The Stack Class: Implementation

• We only implement the push() and pop() methods, thereby restricting the stack’s behavior.

public class Stack extends List {
    public Stack() {
        super();
    }
    public void push(Object obj) {
        insertAtFront(obj);
    }
    public Object pop() {
        return removeFirst();
    }
} // Stack

Invoke the List() constructor.

Design Issues

• **Changes to List Class:** Change head from private to protected and change the public methods to protected.

• **Protected Elements.** *Protected* elements are hidden from all other objects except instances of the same class or its subclasses.

• **Information Hiding.** Use private and protected qualifiers to hide an ADT’s implementation details from other objects. Use the public to define the ADT’s interface.
Testing the Stack Class

- Stack behavior: Last In First Out (LIFO)
- Use a stack to reverse a string.

```java
public static void main(String argv[]) {
    Stack stack = new Stack();
    String string = "Hello this is a test string";
    System.out.println("String: " + string);
    for (int k = 0; k < string.length(); k++)
        stack.push(new Character(string.charAt(k)));
    Object o = null;
    String reversed = "";
    while (!stack.isEmpty()) {
        o = stack.pop();
        reversed = reversed + o.toString();
    }
    System.out.println("Reversed String: " + reversed);
}
```

Push each character.

Then pop each character.

The Queue ADT

- A queue is a list that limits insertions to the rear and removals to the front of a list.

  - Operations
    - Enqueue: insert an object onto the rear of the list.
    - Dequeue: remove the object at the front of the list.
    - Empty: return true if the queue is empty.

The Queue Class

- A Queue is very easy to design as an extension of our generic List structure.

Abstract Data Type: We limit access to the data to enqueue() and dequeue().

Inheritance: Protected elements are inherited by Queue class.
The Queue Class: Implementation

- We only implement the enqueue() and dequeue() methods, thereby restricting the queue’s behavior.

```
public class Queue extends List {
    public Queue() {
        super();
    }
    public void enqueue(Object obj) {
        insertAtRear(obj);
    }
    public Object dequeue() {
        return removeFirst();
    }
}
```

Invoke the List() constructor.

Abstract Data Type: We limit access to the data to enqueue() and dequeue().

From the Java Library: java.util.Stack

- Java’s Stack class is an extension of Vector:

```
public class Stack extends Vector {
    // Stack methods
}
```

Java’s Stack class in the java.util package.

Java Library: java.util.LinkedList

- Java provides a LinkedList class
- Implements the List interface.
- Advantage: The methods can be attached to a wide range of sequential structures.
In the Laboratory: Capital Gains

- Problem Statement: Write a program that will calculate the capital gain or loss for a stock account.
- Example: LIFO and FIFO accounting.
  - Jan: Buy 100 @ $10
  - Feb: Buy 100 @ $15
  - Mar: Sell 100 @ $20
  - May: Buy 100 @ $3
    - LIFO: 100 * 20 - 100 * 3 = $1700 gain
    - FIFO: 100 * 20 - 100 * 10 = $1000 gain

Lab: GUI Design

- The user can input stock transactions and the program should display the capital gain for both FIFO and LIFO.
- GUI:

Capital Gains Lab: Decomposition

- Decomposition (objects):
  - AccountantFrame: Top-level user interface
  - Accountant: FIFO/LIFO accounting expert
  - Transaction: Stock transaction object
  - Queue: FIFO queue of transactions
  - Stack: LIFO stack of transactions
  - List: Superclass of Queue and Stack
  - Node: Data element of List
Lab Design: Transaction Class

• Represents a buy or sell transaction.

Transaction
- BUY: boolean = true
- SELL: boolean = false
- pricePerShare: double
- numShares: double
- type: boolean

+ Transaction(in price: double, in shares: double, in type: boolean)
+ getNumShares(): String
+ getPricePerShare(): double

Lab Design: Accountant Class

• Calculates capital gains in FIFO or LIFO.

Accountant
- buyQueue: Queue
- sellQueue: Queue
- buyStack: Stack
- sellStack: Stack
+ Accountant()
+ insert()
+ fifoAccounting(): double
+ lifoAccounting(): double

• Algorithm: Assume X shares per transaction:

Set capital gain to zero. // Algorithm for FIFO.
For each sale transaction in the Sales Queue
Add the amount of the sale transaction to capital gain.
Get the next purchase in the Purchases Queue.
Add the amount of the purchase transaction to capital gain.

Lab: Testing the Accountant Helper
Technical Terms

- abstract data type (ADT)
- data structure
- dequeue
- dynamic structure
- enqueue
- first-in-first-out (FIFO)
- last-in-first-out (LIFO)
- link
- list
- linked list
- pop
- push
- queue
- reference
- self-referential object
- stack
- static structure
- traverse
- vector

Summary Of Important Points

- A data structure is used to organize data and make them more efficient to process.
- An array is a static structure, whereas a vector is a dynamic structure.
- A linked list is a linear structure whose individual nodes are linked by references.
- An object that can refer to the same kind of object is said to be self-referential.
- The Node class is a self-referential class.

Summary Of Important Points (cont)

- Traversal algorithms are used to access elements of a singly linked list.
- An Abstract Data Type (ADT) consists of two parts: (1) a collection of data, and (2) the operations on the data.
- Information Hiding: An ADT’s public interface constrains the way it can be used.
- A stack is a LIFO list.
- A queue is a LIFO list.