Chapter 5

Linked Lists
Preliminaries

• Options for implementing an ADT
  – Array
    • Has a fixed size
    • Data must be shifted during insertions and deletions
  – Linked list
    • Is able to grow in size as needed
    • Does not require the shifting of items during insertions and deletions
Figure 5-1
a) A linked list of integers; b) insertion; c) deletion
Object References

• A reference variable
  – Contains the location of an object
  – Example
    ```java
    Integer intRef;
    intRef = new Integer(5);
    ```
  – As a data field of a class
    • Has the default value `null`
  – A local reference variable to a method
    • Does not have a default value
Object References

Figure 5-2
A reference to an *Integer* object

(a) This view shows the data members and methods for the object.

(b) This view only shows the data members for simplicity. This is the view used throughout the text.
Object References

• When one reference variable is assigned to another reference variable, both references then refer to the same object

  Integer p, q;
  p = new Integer(6);
  q = p;

• A reference variable that no longer references any object is marked for garbage collection
Object References

(a) Integer p;
   Integer q;

(b) p = new Integer(5);

(c) p = new Integer(6);

(d) q = p;

Figure 5-3a-d
a) Declaring reference variables; b) allocating an object; c) allocating another object, with the dereferenced object marked for garbage collection
Object References

(e) $q = \text{new Integer}(9)$;

(f) $p = \text{null}$;

(g) $q = p$;

Figure 5-3e-g

- e) allocating an object; f) assigning null to a reference variable; g) assigning a reference with a null value
Object References

• An array of objects
  – Is actually an array of references to the objects
  – Example

    ```java
    Integer[] scores = new Integer[30];
    ```

    – Instantiating Integer objects for each array reference

    ```java
    scores[0] = new Integer(7);
    scores[1] = new Integer(9);  // and so on ...
    ```
Object References

• Equality operators (== and !=)
  – Compare the values of the reference variables, not the objects that they reference

• equals method
  – Compares objects field by field

• When an object is passed to a method as an argument, the reference to the object is copied to the method’s formal parameter

• Reference-based ADT implementations and data structures use Java references
Resizable Arrays

• The number of references in a Java array is of fixed size

• Resizable array
  – An array that grows and shrinks as the program executes
  – An illusion that is created by using an allocate and copy strategy with fixed-size arrays

• `java.util.Vector` class
  – Uses a similar technique to implement a growable array of objects
Reference-Based Linked Lists

- Linked list
  - Contains nodes that are linked to one another
  - A node contains both data and a link to the next item
  - Access is package-private

```java
package List;
class Node {
    Object item;
    Node next;
    // constructors, accessors, // and mutators …
} // end class Node
```

Figure 5-5
A node
Reference-Based Linked Lists

• Using the Node class

```java
Node n = new Node(new Integer(6));
Node first = new Node(new Integer(9), n);
```

Figure 5-7
Using the `Node` constructor to initialize a data field and a link value
Reference-Based Linked Lists

- **Data field `next` in the last node is set to `null`**
- **`head` reference variable**
  - References the list’s first node
  - Always exists even when the list is empty

---

**Figure 5-8**

A *head* reference to a linked list
Reference-Based Linked Lists

- head reference variable can be assigned null without first using new
  - Following sequence results in a lost node
    
    ```java
    head = new Node(); // Don’t really need to use new here
    head = null; // since we lose the new Node object here
    ```

Figure 5-9
A lost node
Programming with Linked Lists: Displaying the Contents of a Linked List

• `curr` reference variable
  – References the current node
  – Initially references the first node

• To display the data portion of the current node
  ```java
  System.out.println(curr.item);
  ```

• To advance the current position to the next node
  ```java
  curr = curr.next;
  ```
Displaying the Contents of a Linked List

Figure 5-10
The effect of the assignment \( \text{curr} = \text{curr}.next \)
Displaying the Contents of a Linked List

- To display all the data items in a linked list
  
  ```java
  for (Node curr = head; curr != null; curr = curr.next) {
    System.out.println(curr.item);
  } // end for
  ```
Deleting a Specified Node from a Linked List

- To delete node N which `curr` references
  - Set `next` in the node that precedes N to reference the node that follows N
    
    ```
    prev.next = curr.next;
    ```

**Figure 5-11**

Deleting a node from a linked list
Deleting a Specified Node from a Linked List

- Deleting the first node is a special case

```java
head = head.next;
```

Figure 5-12
Deleting the first node
Deleting a Specified Node from a Linked List

• To return a node that is no longer needed to the system
  ```java
  curr.next = null;
  curr = null;
  ```

• Three steps to delete a node from a linked list
  - Locate the node that you want to delete
  - Disconnect this node from the linked list by changing references
  - Return the node to the system
Inserting a Node into a Specified Position of a Linked List

- To create a node for the new item
  
  ```java
  newNode = new Node(item);
  ```

- To insert a node between two nodes
  
  ```java
  newNode.next = curr;
  prev.next = newNode;
  ```

Figure 5-13

Inserting a new node into a linked list
Inserting a Node into a Specified Position of a Linked List

- To insert a node at the beginning of a linked list
  ```java
  newNode.next = head;
  head = newNode;
  ```

Figure 5-14
Inserting at the beginning of a linked list
Inserting a Node into a Specified Position of a Linked List

- Inserting at the end of a linked list is not a special case if `curr` is `null`
  ```
  newNode.next = curr;
  prev.next = newNode;
  ```
Inserting a Node into a Specified Position of a Linked List

• Three steps to insert a new node into a linked list
  – Determine the point of insertion
  – Create a new node and store the new data in it
  – Connect the new node to the linked list by changing references
Determining `curr` and `prev`

- Determining the point of insertion or deletion for a sorted linked list of objects

```java
for ( prev = null, curr = head;
     (curr != null) &&
     (newValue.compareTo(curr.item) > 0);
     prev = curr, curr = curr.next ) {
}
```

```java
} // end for
```
A Reference-Based Implementation of the ADT List

- A reference-based implementation of the ADT list
  - Does not shift items during insertions and deletions
  - Does not impose a fixed maximum length on the list

Figure 5-18
A reference-based implementation of the ADT list
A Reference-Based Implementation of the ADT List

- **Default constructor**
  - Initializes the data fields `numItems` and `head`

- **List operations**
  - **Public methods**
    - `isEmpty`
    - `size`
    - `add`
    - `remove`
    - `get`
    - `removeAll`
  - **Private method**
    - `find`
Comparing Array-Based and Referenced-Based Implementations

• Size
  – Array-based
  • Fixed size
    – Issues
      » Can you predict the maximum number of items in the ADT?
      » Will an array waste storage?
    – Resizable array
      » Increasing the size of a resizable array can waste storage and time
Comparing Array-Based and Referenced-Based Implementations

• Size (Continued)
  – Reference-based
    • Do not have a fixed size
      – Do not need to predict the maximum size of the list
      – Will not waste storage

• Storage requirements
  – Array-based
    • Requires less memory than a reference-based implementation
      – There is no need to store explicitly information about where to find the next data item
Comparing Array-Based and Referenced-Based Implementations

- **Storage requirements (Continued)**
  - Reference-based
    - Requires more storage
      - An item explicitly references the next item in the list

- **Access time**
  - Array-based
    - Constant access time
  - Reference-based
    - The time to access the $i^{th}$ node depends on $i$
Comparing Array-Based and Referenced-Based Implementations

- Insertion and deletions
  - Array-based
    - Require you to shift the data
  - Reference-based
    - Do not require you to shift the data
    - Require a list traversal
Passing a Linked List to a Method

- A method with access to a linked list’s head reference has access to the entire list.
- When head is an actual argument to a method, its value is copied into the corresponding formal parameter.

Figure 5-19
A head reference as an argument
Processing Linked Lists Recursively

• Traversal
  – Recursive strategy to display a list
    Write the first node of the list
    Write the list minus its first node
  – Recursive strategies to display a list backward
    • writeListBackward strategy
      Write the last node of the list
      Write the list minus its last node backward
    • writeListBackward2 strategy
      Write the list minus its first node backward
      Write the first node of the list
Processing Linked Lists Recursively

- Insertion
  - Recursive view of a sorted linked list
    - The linked list that head references is a sorted linked list if head is null (the empty list is a sorted linked list)
    - or
    - head.next is null (a list with a single node is a sorted linked list)
    - or
    - head.item < head.next.item,
    - and head.next references a sorted linked list
Variations of the Linked List: Tail References

- tail references
  - Remembers where the end of the linked list is
  - To add a node to the end of a linked list
    ```
    tail.next = new Node(request, null);
    ```

Figure 5-22
A linked list with head and tail references
Circular Linked List

- Last node references the first node
- Every node has a successor

Figure 5-23
A circular linked list
Circular Linked List

Figure 5-24
A circular linked list with an external reference to the last node
Dummy Head Nodes

• Dummy head node
  – Always present, even when the linked list is empty
  – Insertion and deletion algorithms initialize \texttt{prev} to reference the dummy head node, rather than \texttt{null}

Figure 5-25
A dummy head node
Doubly Linked List

- Each node references both its predecessor and its successor
- Dummy head nodes are useful in doubly linked lists

Figure 5-26
A doubly linked list
Doubly Linked List

- Circular doubly linked list
  - preceding reference of the dummy head node references the last node
  - next reference of the last node references the dummy head node
  - Eliminates special cases for insertions and deletions
Doubly Linked List

Figure 5-27
a) A circular doubly linked list with a dummy head node; b) an empty list with a dummy head node
Doubly Linked List

- To delete the node that `curr` references:
  ```
  curr.preceding.next = curr.next;
  curr.next.preceding = curr.preceding;
  ```

**Figure 5-28**
Reference changes for deletion
Doubly Linked List

- To insert a new node that `newNode` references before the node referenced by `curr`:
  
  ```
  newNode.next = curr;
  newNode.preceding = curr.preceding;
  curr.preceding = newNode;
  newNode.preceding.next = newNode;
  ```

Figure 5-29
Reference changes for insertion
Application: Maintaining an Inventory

• Stages of the problem-solving process
  – Design of a solution
  – Implementation of the solution
  – Final set of refinements to the program

• Operations on the inventory
  – List the inventory in alphabetical order by title (L command)
  – Find the inventory item associated with title (I, M, D, O, and S commands)
  – Replace the inventory item associated with a title (M, D, R, and S commands)
  – Insert new inventory items (A and D commands)
The Java Collections Framework

• Implements many of the more commonly used ADTs
• Collections framework
  – Unified architecture for representing and manipulating collections
  – Includes
    • Interfaces
    • Implementations
    • Algorithms
Generics

- JCF relies heavily on Java generics
- Generics
  - Develop classes and interfaces and defer certain data-type information
    - Until you are actually ready to use the class or interface
- Definition of the class or interface is followed by $<E>$
  - $E$ represents the data type that client code will specify
Iterators

- **Iterator**
  - Gives the ability to cycle through items in a collection
  - Access next item in a collection by using `iter.next()`
- **JCF provides two primary iterator interfaces**
  - `java.util.Iterator`
  - `java.util.ListIterator`
- **Every ADT collection in the JCF have a method to return an iterator object**
Iterators

- **ListIterator methods**
  - `void add(E o)`
  - `boolean hasNext()`
  - `boolean hasPrevious()`
  - `E next()`
  - `int nextIndex()`
  - `E previous()`
  - `int previousIndex()`
  - `void remove()`
  - `void set(E o)`
The Java Collection’s Framework

**List Interface**

- JCF provides an interface `java.util.List`
- List interface supports an ordered collection
  - Also known as a sequence
- Methods
  - `boolean add(E o)`
  - `void add(int index, E element)`
  - `void clear()`
  - `boolean contains(Object o)`
  - `boolean equals(Object o)`
  - `E get(int index)`
  - `int indexOf(Object o)`
The Java Collection’s Framework

List Interface

• Methods (continued)
  - `boolean isEmpty()`
  - `Iterator<E> iterator()`
  - `ListIterator<E> listIterator()`
  - `ListIterator<E> listIterator(int index)`
  - `E remove(int index)`
  - `boolean remove(Object o)`
List Interface

- E set(int index, E element)
- int size()
- List<E> subList(int fromIndex, int toIndex)
- Object[] toArray()
Summary

• Reference variables can be used to implement the data structure known as a linked list
• Each reference in a linked list is a reference to the next node in the list
• Algorithms for insertions and deletions in a linked list involve
  – Traversing the list from the beginning until you reach the appropriate position
  – Performing reference changes to alter the structure of the list
Summary

• Inserting a new node at the beginning of a linked list and deleting the first node of a linked list are special cases
• An array-based implementation uses an implicit ordering scheme; a reference-based implementation uses an explicit ordering scheme
• Any element in an array can be accessed directly; you must traverse a linked list to access a particular node
• Items can be inserted into and deleted from a reference-based linked list without shifting data
Summary

• The new operator can be used to allocate memory dynamically for both an array and a linked list
  – The size of a linked list can be increased one node at a time more efficiently than that of an array
• A binary search of a linked list is impractical
• Recursion can be used to perform operations on a linked list
• The recursive insertion algorithm for a sorted linked list works because each smaller linked list is also sorted
Summary

• A tail reference can be used to facilitate locating the end of a list
• In a circular linked list, the last node references the first node
• Dummy head nodes eliminate the special cases for insertion into and deletion from the beginning of a linked list
• A head record contains global information about a linked list
• A doubly linked list allows you to traverse the list in either direction
Summary

• Generic class or interface
  – Enables you to defer the choice of certain data-type information until its use

• Java Collections Framework
  – Contains interfaces, implementations, and algorithms for many common ADTs

• Collection
  – Object that holds other objects
  – Iterator cycles through its contents