Chapter 3

Linked Lists
Objectives

Discuss the following topics:
• Singly Linked Lists
• Doubly Linked Lists
• Circular Lists
• Skip Lists
• Self-Organizing Lists
• Sparse Tables
• Lists in java.util
• Case Study: A Library
Singly Linked Lists

• A **linked structure** is a collection of nodes storing data and links to other nodes
• A **linked list** is a data structure composed of nodes, each node holding some information and a reference to another node in the list
• A **singly linked list** is a node that has a link only to its successor in this sequence
Singly Linked Lists (continued)

Figure 3-1 A singly linked list
Singly Linked Lists (continued)

Figure 3-1 A singly linked list (continued)
public class IntSLLNode {
    public int info;
    public IntSLLNode next;
    public IntSLLNode(int i) {
        this(i,null);
    }
    public IntSLLNode(int i, IntSLLNode n) {
        info = i; next = n;
    }
}
public class IntSLLNode {
    public int info;
    public IntSLLNode next;
    public IntSLLNode(int i) {
        this(i, null);
    }
    public IntSLLNode(int i, IntSLLNode n) {
        info = i; next = n;
    }
}
Singly Linked Lists (continued)

```java
//************************************************** IntSLList.java **************************************************
//             singly linked list class to store integers

public class IntSLList {
    protected IntSLLNode head, tail;
    public IntSLLList() {
        head = tail = null;
    }
    public boolean isEmpty() {
        return head == null;
    }
}

Figure 3-2 An implementation of a singly linked list of integers (continued)
Singly Linked Lists (continued)

```java
public void addToHead(int el) {
    head = new IntSLLNode(el, head);
    if (tail == null)
        tail = head;
}
public void addToTail(int el) {
    if (!isEmpty()) {
        tail.next = new IntSLLNode(el);
        tail = tail.next;
    } else head = tail = new IntSLLNode(el);
}
public int deleteFromHead() { // delete the head and return its info;
    int el = head.info;
    if (head == tail) // if only one node on the list;
        head = tail = null;
    else head = head.next;
    return el;
}
```

Figure 3-2 An implementation of a singly linked list of integers (continued)
public int deleteFromTail() { // delete the tail and return its info;
    int el = tail.info;
    if (head == tail) // if only one node on the list;
        head = tail = null;
    else { // if more than one node on the list,
        IntSLLNode tmp; // find the predecessor of tail;
        for (tmp = head; tmp.next != tail; tmp = tmp.next);
        tail = tmp; // the predecessor of tail becomes tail;
        tail.next = null;
    }
    return el;
}

public void printAll() {
    for (IntSLLNode tmp = head; tmp != null; tmp = tmp.next)
        System.out.print(tmp.info + " ");
}

public boolean isInList(int el) {
    IntSLLNode tmp;
    for (tmp = head; tmp != null && tmp.info != el; tmp = tmp.next);
    return tmp != null;
}

Figure 3-2 An implementation of a singly linked list of integers (continued)
Singly Linked Lists (continued)

public void delete(int el) { // delete the node with an element el;
    if (!isEmpty())
        if (head == tail && el == head.info) // if only one
            head = tail = null; // node on the list;
        else if (el == head.info) // if more than one node on the
            head = head.next; // list; and el is in the head node;
        else { // if more than one node in the list
            IntSLLNode pred, tmp;// and el is in a non-head node;
            for (pred = head, tmp = head.next;
                tmp != null && tmp.info != el;
                pred = pred.next, tmp = tmp.next);
            if (tmp != null) { // if el was found;
                pred.next = tmp.next;
                if (tmp == tail) // if el is in the last node;
                    tail = pred;
            }
        }
    }

Figure 3-2 An implementation of a singly linked list of integers (continued)
Singly Linked Lists (continued)

Figure 3-3 A singly linked list of integers
Singly Linked Lists (continued)

Figure 3-4 Inserting a new node at the beginning of a singly linked list
Singly Linked Lists (continued)

Figure 3-5 Inserting a new node at the end of a singly linked list
Singly Linked Lists (continued)

Figure 3-6 Deleting a node from the beginning of a singly linked list
Figure 3-7 Deleting a node from the end of a singly linked list
Singly Linked Lists (continued)

```java
//************************************************************* SLLNode.java *************************************************************

public class SLLNode {
    public Object info;
    public SLLNode next;
    public SLLNode() {
        next = null;
    }
    public SLLNode(Object el) {
        info = el; next = null;
    }
    public SLLNode(Object el, SLLNode ptr) {
        info = el; next = ptr;
    }
}

Figure 3-9 Implementation of a generic singly linked list
```

Data Structures and Algorithms in Java
Singly Linked Lists (continued)

```java
/**
 * generic singly linked list class with head only
 */

public class SLLList {
    protected SLLNode head = null;
    public SLLList() {
    }
    public boolean isEmpty() {
        return head == null;
    }
    public Object first() {
        return head.info;
    }
}
```

Figure 3-9 Implementation of a generic singly linked list (continued)
public void printAll(java.io.PrintStream out) {
    for (SLLNode tmp = head; tmp != null; tmp = tmp.next)
        out.print(tmp.info);
}

public void add(Object el) {
    head = new SLLNode(el, head);
}

public Object find(Object el) {
    SLLNode tmp = head;
    for ( ; tmp != null && !el.equals(tmp.info); tmp = tmp.next);
    if (tmp == null)
        return null;
    else return tmp.info;
}

Figure 3-9 Implementation of a generic singly linked list (continued)
Singly Linked Lists (continued)

```java
public Object deleteHead() { // remove the head and return its info;
    Object el = head.info;
    head = head.next;
    return el;
}

public void delete(Object el) { // find and remove el;
    if (head != null) { // if nonempty list;
        if (el.equals(head.info)) { // if head needs to be removed;
            head = head.next;
        } else {
            SLINode pred = head, tmp = head.next;
            for (; tmp != null && !(tmp.info.equals(el));
                pred = pred.next, tmp = tmp.next);
            if (tmp != null) { // if found
                pred.next = tmp.next;
            }
        }
    }
}
```

Figure 3-9 Implementation of a generic singly linked list (continued)
Doubly Linked Lists

- A **doubly linked list** is when each node in a linked list has two reference fields, one to the successor and one to the predecessor.

![Figure 3-10 A doubly linked list](image)

Figure 3-10 A doubly linked list
Doubly Linked Lists (continued)

```java
/****************************** IntDLLNode.java *******************************/

public class IntDLLNode {
    public int info;
    public IntDLLNode next, prev;
    public IntDLLNode(int el) {
        this(el,null,null);
    }
    public IntDLLNode(int el, IntDLLNode n, IntDLLNode p) {
        info = el; next = n; prev = p;
    }
}

Figure 3-11 An implementation of a doubly linked list
Doubly Linked Lists (continued)

/**************************** IntDLList.java ****************************/

class IntDLList {
    private IntDLLNode head, tail;
    public IntDLList() {
        head = tail = null;
    }
    public boolean isEmpty() {
        return head == null;
    }
    public void addToTail(int el) {
        if (!isEmpty()) {
            tail = new IntDLLNode(el, null, tail);
            tail.prev.next = tail;
        } else head = tail = new IntDLLNode(el);
    }
}

Figure 3-11 An implementation of a doubly linked list (continued)
public int removeFromTail() {
    int el = tail.info;
    if (head == tail) // if only one node in the list;
        head = tail = null;
    else { // if more than one node in the list;
        tail = tail.prev;
        tail.next = null;
    }
    return el;
}

Figure 3-11 An implementation of a doubly linked list (continued)
Doubly Linked Lists (continued)

![Diagram of adding a new node at the end of a doubly linked list]

Figure 3-12 Adding a new node at the end of a doubly linked list
Figure 3-12 Adding a new node at the end of a doubly linked list (continued)
Figure 3-13 Deleting a node from the end of a doubly linked list
Circular Lists

- **A circular list** is when nodes form a ring: The list is finite and each node has a successor.

![Figure 3-14 A circular singly linked list](image)

Data Structures and Algorithms in Java

28
Circular Lists (continued)

Figure 3-15  Inserting nodes at the front of a circular singly linked list (a) and at its end (b)
Circular Lists (continued)

Figure 3-16 A circular doubly linked list
Skip Lists

• A skip list is a variant of the ordered linked list that makes a nonsequential search possible.

• In a skip list of $n$ nodes, for each $k$ and $i$ such that $1 \leq k \leq \lfloor \lg n \rfloor$ and $1 \leq i \leq \lfloor n/2^{k-1} \rfloor – 1$, the node in position $2^{k-1} \cdot i$ points to the node in position $2^{k-1} \cdot (i + 1)$.

• The number of reference fields indicates the level of each node, and the number of levels is $\maxLevel = \lfloor \lg n \rfloor + 1$. 
Skip Lists (continued)

Figure 3-17 A skip list with (a) evenly and (b) unevenly spaced nodes of different levels
Skip Lists (continued)

Figure 3-17 (c) the skip list with reference nodes clearly shown (continued)
Skip Lists (continued)

```
/** ************************ IntSkipListNode.java **************************/

public class IntSkipListNode {
    public int key;
    public IntSkipListNode[] next;
    IntSkipListNode(int i, int n) {
        key = i;
        next = new IntSkipListNode[n];
        for (int j = 0; j < n; j++)
            next[j] = null;
    }
}
```

Figure 3-18 An implementation of a skip list
Data Structures and Algorithms in Java

Skip Lists (continued)

```java
import java.util.Random;

public class IntSkipList {
    private int maxLevel;
    private IntSkipListNode[] root;
    private int[] powers;
    private Random rd = new Random();
    IntSkipList() {
        this(4);
    }
    IntSkipList (int i) {
        maxLevel = i;
        root = new IntSkipListNode[maxLevel];
        powers = new int[maxLevel];
        for (int j = 0; j < maxLevel; j++)
            root[j] = null;
        choosePowers();
    }
}
```

Figure 3-18 An implementation of a skip list (continued)
public boolean isEmpty() {
    return root[0] == null;
}

public void choosePowers() {
    powers[maxLevel-1] = (2 << (maxLevel-1)) - 1;  // 2^maxLevel - 1
    for (int i = maxLevel - 2, j = 0; i >= 0; i--, j++)
        powers[i] = powers[i+1] - (2 << j);          // 2^(j+1)
}

public int chooseLevel() {
    int i, r = Math.abs(rd.nextInt()) % powers[maxLevel-1] + 1;
    for (i = 1; i < maxLevel; i++)
        if (r < powers[i])
            return i-1; // return a level < the highest level;
    return i-1;    // return the highest level;
}
Skip Lists (continued)

```java
// make sure (with isEmpty()) that skipListSearch() is called for a
// nonempty list;
public int skipListSearch (int key) {
  int lvl;
  IntSkipListNode prev, curr; // find the highest non-null
  for (lvl = maxLevel-1; lvl >= 0 && root[lvl] == null; lvl--); // level;
  prev = curr = root[lvl];
  while (true) {
    if (key == curr.key) // success if equal;
      return curr.key;
    else if (key < curr.key) { // if smaller, go down,
      if (lvl == 0) // if possible,
        return 0;
      else if (curr == root[lvl]) // by one level
        curr = root[--lvl]; // starting from the
      else curr = prev.next[--lvl]; // predecessor which
    } // can be the root;
    else { // if greater,
      prev = curr; // go to the next
      if (curr.next[lvl] != null) // non-null node
        curr = curr.next[lvl]; // on the same level
      else { // or to a list on a lower level;
        for (lvl--; lvl >= 0 && curr.next[lvl] == null; lvl--);
        if (lvl == 0)
          curr = curr.next[lvl];
        else return 0;
      }
    }
  }
}
```

Figure 3-18 An implementation of a skip list (continued)
Skip Lists (continued)

```java
public void skipListInsert (int key) {
    IntSkipListNode[] curr = new IntSkipListNode[maxLevel];
    IntSkipListNode[] prev = new IntSkipListNode[maxLevel];
    IntSkipListNode newNode;
    int lvl, i;
    curr[maxLevel-1] = root[maxLevel-1];
    prev[maxLevel-1] = null;
    for (lvl = maxLevel - 1; lvl >= 0; lvl--) {
        while (curr[lvl] != null && curr[lvl].key < key) { // go to the next
            prev[lvl] = curr[lvl]; // if smaller;
            curr[lvl] = curr[lvl].next[lvl];
        }
    }
}
```

Figure 3-18 An implementation of a skip list (continued)
Skip Lists (continued)

```java
if (curr[lvl] != null && curr[lvl].key == key) // don't include
    return; // duplicates;
if (lvl > 0) // go one level down
    if (prev[lvl] == null) { // if not the lowest
        curr[lvl-1] = root[lvl-1]; // level, using a link
        prev[lvl-1] = null; // either from the root
    }
else { // or from the predecessor;
    curr[lvl-1] = prev[lvl].next[lvl-1];
    prev[lvl-1] = prev[lvl];
}
```

Figure 3-18 An implementation of a skip list (continued)
Skip Lists (continued)

```java
}
lvl = chooseLevel();       // generate randomly level
                          // for newNode;
newNode = new IntSkipListNode(key, lvl+1);
for (i = 0; i <= lvl; i++) { // initialize next fields of
    newNode.next[i] = curr[i]; // newNode and reset to newNode
    if (prev[i] == null)     // either fields of the root
        root[i] = newNode;  // or next fields of newNode's
    else prev[i].next[i] = newNode; // predecessors;
}
}
```

Figure 3-18 An implementation of a skip list (continued)
Self-Organizing Lists

There are four methods for organizing lists:

• **Move-to-front method** – after the desired element is located, put it at the beginning of the list (Figure 3.19a)

• **Transpose method** – after the desired element is located, swap it with its predecessor unless it is at the head of the list (Figure 3.19b)

• **Count method** – order the list by the number of times elements are being accessed (Figure 3.19c)
Self-Organizing Lists (continued)

- **Ordering method** – order the list using certain criteria natural for the information under scrutiny (Figure 3.19d)

- **Optimal static ordering** all the data are already ordered by the frequency of their occurrence in the body of data so that the list is used only for searching, not for inserting new items
Self-Organizing Lists (continued)

<table>
<thead>
<tr>
<th>Element Searched For</th>
<th>Plain</th>
<th>Move-to-Front</th>
<th>Transpose</th>
<th>Count</th>
<th>Ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>A C</td>
<td>A C</td>
<td>A C</td>
<td>A C</td>
<td>A C</td>
</tr>
<tr>
<td>B</td>
<td>A C B</td>
<td>A C B</td>
<td>A C B</td>
<td>A C B</td>
<td>A B C</td>
</tr>
<tr>
<td>C</td>
<td>A C B</td>
<td>C A B</td>
<td>C A B</td>
<td>C A B</td>
<td>A B C</td>
</tr>
<tr>
<td>D</td>
<td>A C B D</td>
<td>C A B D</td>
<td>C A B D</td>
<td>C A B D</td>
<td>A B C D</td>
</tr>
<tr>
<td>A</td>
<td>A C B D</td>
<td>A C B D</td>
<td>A C B D</td>
<td>A C B D</td>
<td>A B C D</td>
</tr>
<tr>
<td>D</td>
<td>A C B D</td>
<td>D A C B</td>
<td>A C D B</td>
<td>D C A B</td>
<td>A B C D</td>
</tr>
<tr>
<td>A</td>
<td>A C B D</td>
<td>A D C B</td>
<td>A C D B</td>
<td>A D C B</td>
<td>A B C D</td>
</tr>
<tr>
<td>C</td>
<td>A C B D</td>
<td>C A D B</td>
<td>C A D B</td>
<td>C A D B</td>
<td>A B C D</td>
</tr>
<tr>
<td>A</td>
<td>A C B D</td>
<td>A C D B</td>
<td>A C D B</td>
<td>A C D B</td>
<td>A B C D</td>
</tr>
<tr>
<td>C</td>
<td>A C B D</td>
<td>C A D B</td>
<td>C A D B</td>
<td>C A D B</td>
<td>A B C D</td>
</tr>
<tr>
<td>E</td>
<td>A C B D E</td>
<td>C A D B E</td>
<td>C A D B E</td>
<td>C A D B E</td>
<td>A B C D E</td>
</tr>
<tr>
<td>E</td>
<td>A C B D E</td>
<td>E C A D B</td>
<td>C A D E B</td>
<td>C A E D B</td>
<td>A B C D E</td>
</tr>
</tbody>
</table>

Figure 3-20 Processing the stream of data, A C B C D A D A C A C C E E, by different methods of organizing linked lists
Self-Organizing Lists (continued)

<table>
<thead>
<tr>
<th>Different Words/All Words</th>
<th>156/347</th>
<th>149/423</th>
<th>609/1510</th>
<th>550/2847</th>
<th>1163/5866</th>
<th>2013/23065</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal</td>
<td>28.5</td>
<td>26.4</td>
<td>24.5</td>
<td>17.6</td>
<td>16.2</td>
<td>10.0</td>
</tr>
<tr>
<td>Plain</td>
<td>70.3</td>
<td>71.2</td>
<td>67.1</td>
<td>56.3</td>
<td>51.7</td>
<td>35.4</td>
</tr>
<tr>
<td>Move-to-Front</td>
<td>61.3</td>
<td>49.5</td>
<td>54.5</td>
<td>31.3</td>
<td>30.5</td>
<td>18.4</td>
</tr>
<tr>
<td>Transpose</td>
<td>68.8</td>
<td>69.5</td>
<td>66.1</td>
<td>53.3</td>
<td>49.4</td>
<td>32.9</td>
</tr>
<tr>
<td>Count</td>
<td>61.2</td>
<td>51.6</td>
<td>54.7</td>
<td>34.0</td>
<td>32.0</td>
<td>19.8</td>
</tr>
<tr>
<td>Alphabetical Order</td>
<td>50.9</td>
<td>45.6</td>
<td>48.0</td>
<td>55.7</td>
<td>50.4</td>
<td>50.0</td>
</tr>
<tr>
<td>Skip List</td>
<td>15.1</td>
<td>12.3</td>
<td>6.6</td>
<td>5.5</td>
<td>4.8</td>
<td>3.8</td>
</tr>
</tbody>
</table>

**Figure 3-21** Measuring the efficiency of different methods using formula \((\text{number of data comparisons})/(\text{combined length})\) expressed in percentages.