Linked Lists and Algorithm Analysis

Reading LDC Ch 14.10 - 14.14

Efficiency of Algorithms
Determining the Efficiency of Algorithms

- Analysis of algorithms
  - Major field that provides tools for evaluating the efficiency of different solutions

- What is an efficient algorithm?
  - Faster is better
    - How do you measure time? Wall clock? Computer clock?
    - Using less space is better
      - But if you need to get data out of main memory it takes time

- Algorithm analysis should be independent of
  - Specific implementations and coding tricks (programming language, control statements)
  - Specific Computers (hardware chip, OS, clock speed)
  - Particular set of data
- But size of data should matter

Comparing Two Versions of Selection Sort

```java
public void selectionSort (int[] data) {
    int maxNum;        // max integer
    int maxIndex;      // index of max integer
    int i, j;
    for (j = data.length - 1; j > 0; j--)
    {
        maxIndex = 0;
        maxNum = data[0];
        for (i = 1; i <= j; i++)
        {
            if (data[i] > maxNum)
            {
                maxNum = data[i];
                maxIndex = i;
            }
        }
        swap(data, maxIndex, j);
    }
}

private static int indexOfLargest (int[] theArray, int size) {
    int indexSoFar = 0;
    for (int currIndex = 1; currIndex < size; ++currIndex)
    {
        if (theArray[currIndex] > theArray[indexSoFar])
        {
            indexSoFar = currIndex;
        }
    }
    return indexSoFar;  // index of largest item
}
```

```java
public void selectionSort (int[] theArray, int n) {
    int temp = theArray[n-1];
    for (int last = n-1; last > 0; last--)
    {
        int largest = indexOfLargest (theArray, last+1);
        temp = theArray[largest];
        theArray[largest] = theArray[last];
        theArray[last] = temp;
    }
}

private static void swap (int[] data, int i, int j) {
    //exchanges the contents of data[i] and data[j]
    int temp = data[i];
    data[i] = data[j];
    data[j] = temp;
}
```

Same idea, different implementation. Is it important?
Execution Time of Algorithms

- Counting an algorithm's operations is a good way to assess its efficiency.
  - An algorithm's execution time is related to the number of operations it requires in a worst case scenario.
- Examples of worst case scenarios:
  - Searching a linked list: Operations: about as many as elements.
  - Selection sort: Operations: as we go through the array to select the next minimum, we traverse most of the array again.
  - The Towers of Hanoi: Operations: to solve an instance of n disks, we need to solve 2 instances of n-1 disks...
- We could also consider an average case scenario, but is harder...

Single-statement Execution

- A statement that the computer can execute in one or a few instructions that do not depend on the input size,
- We count it as 1 step:
  \[ O(1) = \text{"order of one"} \]
- This includes arithmetic, logical operations, assignments, but not necessarily function calls, recursive steps, etc.
- For example:

```c
// examples of code with O(1) steps
int i = 100;
if ( (i < n) && (n%2 == 0) ) {
    i = i/n;
}
else {
    i = (i+1)/x*2;
}
```

Each of these statements is considered \( O(1) \).
Thus the overall order of this piece of code is \( k \cdot O(1) = O(1) \)
where \( k \) is the number of individual statements.
What about function calls?

- Although a function call is written in a single statement
  - You cannot automatically count it as \( O(1) \)
  - The complexity of a function call is equal to the complexity of the body of that function.
  - For example:

```java
public void addCD (String title, String artist, double cost, int tracks) {
    if (count == collection.length)
        increaseSize();
    collection[count] = new CD(title, totalCost += cost;
    count++;
}
```

Each of these statements is considered \( O(1) \). Except for increaseSize()
So, this code may not be \( O(1) \)

Analyzing Loop Execution

- With loops, you need to determine the complexity of a set of statements, and
- Need to determine how often a set of statements gets executed to determine the order of an algorithm
- To analyze loop execution, first determine the order of the body of the loop, and then multiply that by the number of times the loop will execute

```java
for (int i = 0; i < n; i++) {
    // some sequence of \( O(1) \) steps
}
```

The loop executes \( n \) times, and the body of the loop is \( O(1) \).
Thus the overall order is \( O(n) * O(1) = O(n) \)
Analyzing Nested Loop Execution

- When loops are nested, we must multiply the complexity of the outer loop by the complexity of the inner loop

```java
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        { // some sequence of O(1) steps
        }
```

- Both the inner and outer loops have complexity of O(n)
- When multiplied together, the order becomes O(n²)

Analyzing Recursive Algorithms

- Determining the order of a recursive algorithm
  - determine the order of the recursion (# of times recursive definition is followed) and multiply it by the order of the body of the recursive method

- Example: Consider the recursive method to compute the product of integers from 1 to some n > 1

```java
public int fact (int n)
{
    int result;
    if (n == 1)
        result = 1;
    else
        result = n * fact(n-1);
    return result;
}
```

- Size of the problem is n, the number of values to be multiplied
- Operation of interest is the multiplication operation
- The body of the method performs one multiplication and therefore is O(1)
- Each time the recursive method is invoked, n is decreased by 1, thus...
- ...the recursive method is called n times
- The order of the entire algorithm is O(n)
Algorithm Growth Rates

- An algorithm’s time requirements can be measured as a function of the problem size n
- Growth rate enables the comparison between algorithms
  - Examples
    - Searching a sorted array requires time proportional to lg n
    - Searching a list requires time proportional to n
    - Selection sort requires time proportional to n^2
  - Notation: Big-Oh aka “order of”
    - Searching a sorted array requires time O(lg n)
    - Searching a list requires time O(n)
    - Selection sort requires time O(n^2)

- Algorithm efficiency is typically a concern for large problems only (as n grows...)

<table>
<thead>
<tr>
<th>Function</th>
<th>10</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>log n</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>n</td>
<td>10</td>
<td>10^2</td>
<td>10^3</td>
<td>10^4</td>
<td>10^5</td>
<td>10^6</td>
</tr>
<tr>
<td>n + log n</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>13</td>
<td>16</td>
<td>19</td>
</tr>
<tr>
<td>n^2</td>
<td>10^2</td>
<td>10^4</td>
<td>10^6</td>
<td>10^8</td>
<td>10^10</td>
<td>10^12</td>
</tr>
<tr>
<td>2^n</td>
<td>10</td>
<td>10^2</td>
<td>10^4</td>
<td>10^6</td>
<td>10^8</td>
<td>10^10</td>
</tr>
</tbody>
</table>
• Definition of the order of an algorithm
  Algorithm A is order f(n), denoted $O(f(n))$, if constants $k$ and $n_0$ exist such that A requires no more than $k \cdot f(n)$ time units to solve a problem of size $n \geq n_0$.

• Growth-rate function
  - A mathematical function used to specify an algorithm’s order in terms of the size of the problem

• “Big-Oh” notation
  - A notation that uses the capital letter O to specify an algorithm’s order
  - Example: $O(n)$, $O(n^2 \log n)$, $O(n^4)$, in general, $O(f(n))$

• Order of growth of some common functions
  $O(1) < O(n \log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$

• Properties of growth-rate functions
  - Summing orders is dominated by the larger order
    $O(f(n)) + O(g(n)) = O(f(n) + g(n)) = O(\max\{f(n),g(n)\})$
  - Therefore you can ignore low-order terms
    $O(n^2 + n) = O(n^2)$
  - Multiplying orders means multiplying terms
    $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$
  - Therefore you can ignore a multiplicative constant in the high-order term
    $O(12 \cdot n^2) = O(n^2)$
/**
   * Adds a CD to the collection of n CDs, increasing the size of the collection if necessary
   */
   public void addCD(String title, String artist, double cost, int tracks) {
      if (count == collection.length)
         increaseSize();
      collection[count] = new CD(title, artist, cost, tracks);
      totalCost += cost;
      count++;
   }

   private void increaseSize(){
      CD[] temp = new CD[collection.length * 2];
      for (int cd = 0; cd < collection.length; cd++)
         temp[cd] = collection[cd];
      collection = temp;
   }
Worst Case and Average Case Analyses

- An algorithm can require different times to solve different problems of the same size
  - Worst-case analysis
    - A determination of the maximum amount of time that an algorithm requires to solve problems of size n
  - Average-case analysis
    - A determination of the average amount of time that an algorithm requires to solve problems of size n

Keeping your Perspective

- Throughout the course of an analysis, keep in mind that you are interested only in significant differences in efficiency
- When choosing an ADT’s implementation, consider how frequently particular ADT operations occur in a given application
- Some seldom-used but critical operations must be efficient
- If the problem size is always small, you can probably ignore an algorithm’s efficiency
- Weigh the trade-offs between an algorithm’s time requirements and its memory requirements
- Compare algorithms for both style and efficiency
- Order-of-magnitude analysis focuses on large problems
A collection is an object that serves as a repository for other objects. A collection provides services to add, remove, and manage the elements it contains. The underlying data structure used to implement the collection is independent of the operations provided. Collections can be separated into two categories:

- **Linear**: elements are organized in a straight line
- **Nonlinear**: elements are organized in something other than a straight line

Ordering of elements, relative to each other, is usually determined by either:

- the order in which they were added to the collection
- or some inherent relationship among the elements
Generic Types <E>

Assume we have defined a Group class that stores and manages a group of objects:
- Group could store Objects (which can hold any type - polymorphism!)
- But it’s no longer clear what methods are available for the objects in my group...

```java
(Die) (myGroup.getMember()).roll();
```

Instead, a generic type Group will be able to store, operate on, and manage objects whose type is not specified until the class is instantiated:

```java
public class Group <E>{
    //definition
}
```

Instantiating a Group of Product objects:
```java
Group<Product> group1 = new Group<Product>;
```

Instantiating a Group of Friend objects:
```java
Group<Friend> group2 = new Group<Friend>;
```

You **cannot** instantiate a generic type E:
```java
Group<E> bad_group = new Group<E>;
```

You can be more specific: We want to store Comparable items:
```java
class Group<E extends Comparable<E>> {
    // declarations and code that manages objects of type E
}
```
An abstract data type (ADT) is a set of data and the particular operations that are allowed on that data. Abstract means the operations you can perform on it are separated from the underlying implementation.

For every collection we examine, we should consider:
- How does the collection operate, conceptually?
- What operations are included in the interface to the collection?
- What kinds of problems does the collection help us solve?
- How might the collection be implemented?
- How do the implementations compare from an efficiency point of view?
**Linked Lists**

- Arrays are limited in the sense they have a fixed size
  - Resizing as needed must be done carefully and is not efficient
- A *linked structure* is the primary alternative to an array-based implementation of a collection
- Initial implementation idea: a class `Person` as part of a community

```java
public class Person{
    private String name;
    private String address;
    private Person next;
    // and whatever else
}
```

**Elements with Links**

- Using only this one class, a linked structure of “nodes” is created
  - One Person object contains a link to another Person object
  - This second object contains a reference to a third Person, etc.
  - This type of object is sometimes called *self-referential*
  - This kind of relationship forms the basis of a *linked list*
- Has capacity limited only by memory in the computer
  - Is a *dynamic structure* because its size grows and shrinks as needed to accommodate the number of elements stored
Doubly Linked Lists

- A simple linked list is only one kind of linked structure
- In a *doubly linked list*, each node in the list stores both a reference to the next element and a reference to the previous one
- Java’s library implementation uses doubly linked lists

Nonlinear Linked Structures

- Using multiple links you can create non-linear structures
Managing Linked Lists

- A node may be inserted or deleted at any location
  - at the front of the list,
  - among the interior nodes, or
  - at the end of the list
- There are a few basic techniques when managing nodes on the list, no matter what the list is used to store
- Special care must be taken when dealing with the first or last node in the list so that the reference to the entire list is maintained appropriately

Inserting a node at the front
Inserting a node in the middle

Deleting the first node in the list
Deleting an interior node

Link nodes not elements
A flaw in the Person class is that the self-referential Person class must be designed so that it “knows” it may become a node in a linked list. This violates the goal of separating the implementation details from the parts of the system that use the collection. We better separate the details of the linked list structure from the elements that the list stores. Better: define a separate node class LinearNode that serves to link the elements together.
package javafoundations;

public class LinearNode<T> {
    private LinearNode<T> next;
    private T element;

    //Creates an empty node
    public LinearNode() {
        next = null;
        element = null;
    }

    //Creates a node storing the specified element
    public LinearNode(T elem) {
        next = null;
        element = elem;
    }

    //Returns the node that follows this one
    public LinearNode<T> getNext() {
        return next;
    }

    //Sets the node that follows this one
    public void setNext(LinearNode<T> node) {
        next = node;
    }

    //Returns the element stored in this node
    public T getElement() {
        return element;
    }

    //Sets the element stored in this node
    public void setElement(T elem) {
        element = elem;
    }
}
// Four people
Person p1 = new Person("Takis", "1600 Pennsylvania Ave");
Person p2 = new Person("Stella", "106 Central Street");
Person p3 = new Person("Jean", "123 Sesame Street");
Person p4 = new Person("Christine", "4YawkeyWay");

LinearNode<Person> front; // Refers to first node in list
LinearNode<Person> temp; // Refers to new node as it is
// being added to list

front = new LinearNode<Person>(p4); // List contains one node
Using LinearNodes

```java
// Add new node to front of list
temp = new LinearNode<Person>(p3); // Create new node
temp.setNext(front); // Add to front of list
front = temp; // Update "front"
```

Using LinearNodes

```java
// Add new node to front of list
temp = new LinearNode<Person>(p3); // Create new node
temp.setNext(front); // Add to front of list
front = temp; // Update "front"
```

// Add new node to front of list
temp = new LinearNode<Person>(p2); // Create new node
temp.setNext(front); // Add to front of list
front = temp; // Update "front"
Using LinearNodes

// Add new node to front of list
temp = new LinearNode<Person>(p3);  // Create new node
temp.setNext(front);  // Add to front of list
front = temp;  // Update "front"

// Add new node to front of list
temp = new LinearNode<Person>(p2);  // Create new node
temp.setNext(front);  // Add to front of list
front = temp;  // Update "front"

// Add new node to front of list
temp = new LinearNode<Person>(p1);  // Create new node
temp.setNext(front);  // Add to front of list
front = temp;  // Update "front"

// Returns the number of nodes in list L. Uses a loop.
public static int getLength_loop(LinearNode<Person> L) {
    int count = 0;
    for (LinearNode<Person> current=L; current!=null; current= current.getNext()){
        count++;
    }
    return count;
}
Using LinearNodes

// Returns the number of nodes in list L. Uses recursion.
public static int getLength_rec(LinearNode<Person> L) {
    if (L == null) return 0;
    else return 1 + getLength_rec(L.getNext());
}

Get Person at Specified Index

// If the index is valid, returns the element in the list at the
// specified index. If the index is invalid, e.g., less than 0 or
// greater than or equal to the list length, then "null" is returned.
public static Person getPersonAtIndex(LinearNode<Person> L, int index) {
    int count = 0;
    for (LinearNode<Person> current=L; current!=null; current = current.getNext()){
        if (count == index) return current.getElement();
        count++;
    }
    return null;
}
String Representation

// Return a String representation of the specified list L.
public static String listToString(LinearNode<Person> L) {

}

LinkedList in Java

Seems like an array, but YOU KNOW BETTER!
Java’s LinkedList\<E\> “feels” like an array:

Even though Java uses doubly linked list, we will use the simplified picture because it is simple and clear about accessibility.

To use it you need to start your code by:

```java
import java.util.*;
```

To get a new list:

```java
LinkedList\<E\> L = new LinkedList\<E\>();
```

### Methods for Adding LinkedList Nodes

- **public void add** (int index, E element)
  - indices change automatically!
  - if index > size() throws `IndexOutOfBoundsException`

- **public void addFirst** (E element)

- **public void addLast** (E element)

**aka:** public void **add** (E element)

```java
LinkedList\<String\> L1 =
    new LinkedList\<String\>();
L1.add(0, "eggs");
L1.addFirst("milk");
L1.addLast("bread");
L1.add(2, "chicken");
System.out.println("contents of L1: " + L1);
```
Methods for Getting List Items

public E getFirst ()
public E getLast ()
public E get (int index)

```
LinkedList<String> L2 = new LinkedList<String>();
for (int i = L1.size() - 1; i >= 0; i--)
    L2.add(L1.get(i));
System.out.println("contents of L2: " + L2);
```

What does L2 contain?

Methods for Setting and Removing Items

public E set (int index, E element)
public E remove (int index)
public E removeFirst ()
public E removeLast ()

```
L1.remove(1);
L1.set(2, "beer");
L1.set(1, L1.removeFirst());
L1.addFirst(L1.getLast());
L1.add(1, "butter");
System.out.println("new contents of L1: " + L1);
```

new contents of L1: (what happened to the beer?)
The Contract for the LinkedNodes Collection

<table>
<thead>
<tr>
<th>LinkedNodes ADT operations</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>Adds an element to the front of the list</td>
</tr>
<tr>
<td>addAt</td>
<td>Adds an element to some specific index in the list</td>
</tr>
<tr>
<td>get</td>
<td>Retrieves an element from some specific index in the list</td>
</tr>
<tr>
<td>remove</td>
<td>Removes an element from some specific index in the list</td>
</tr>
<tr>
<td>isEmpty</td>
<td>Determines if the list is empty</td>
</tr>
<tr>
<td>size</td>
<td>Determines the number of elements in the list</td>
</tr>
</tbody>
</table>
public interface LinkedNodes<E>
{
    /** Adds an element to the front of the list. */
    public void add(E newItem);
    /** Adds an element to some specific index in the list. */
    public void addAt(E newItem, int index);
    /** Retrieves an element from some specific index in the list. */
    public E get(int index);
    /** Removes an element from some specific index in the list. */
    public E remove(int index);
    /** Determines whether the list is empty. */
    public boolean isEmpty();
    /** Gets the size of the list. */
    public int size();
}

The need for Interfaces

// A class that a software designer wants implemented
// She wants clients to be able to use without seeing the code
// (they should just see only the INTERFACE to the class)
// She wants coders to update without messing up clients
// (coders should have freedom on how to IMPLEMENT it best)

public class Doable
{
    public void doThis ()
    {
        // code to do this
    }

    public void doThat (int num)
    {
        // code to do that
    }

    public boolean doTheOther ()
    {
        // whatever
    }
}
The need for Interfaces

// coders can implement this file
// without bothering Clients
public class CanDo implements Doable {
    public void doThis ()
    { // code to do this
    }
    public void doThat (int num)
    { // code to do that
    }
    public boolean doTheOther ()
    { // whatever
    }
}

What is an Interface?

- A Java interface is composed of a collection of abstract methods and constants

public interface Doable {
    public void doThis ();
    public int doThat (int num);
    public boolean doTheOther ();
}

interface is a reserved word
Since all methods in an interface are abstract, the keyword abstract is left off

public interface Doable {
    public void doThis ();
    public int doThat (int num);
    public boolean doTheOther ();
}

None of the methods in an interface are given a definition (body)

A semicolon immediately follows each method header
Interfaces

Why may an interface not be instantiated?

Why are interface methods public by default?

Why must a class implementing an interface, define all methods in the interface?

Why may a class implementing an interface also implement other methods?

```java
public class CanDo implements Doable {
    public void doThis () {
        // code to do this
    }
    public void doThat (int num) {
        // code to do that
    }
    public boolean doTheOther () {
        // whatever
    }
}
```

```java
public interface LinkedNodes<E> {
    /** Adds an element to the front of the list. */
    public void add(E newItem);

    /** Adds an element to some specific index in the list. */
    public void addAt(E newItem, int index);

    /** Retrieves an element from some specific index in the list. */
    public E get(int index);

    /** Removes an element from some specific index in the list. */
    public E remove(int index);

    /** Determines whether the list is empty. */
    public boolean isEmpty();

    /** Gets the size of the list. */
    public int size();
}
```
public class SinglyLinkedList<E> implements LinkedNodes<E> {
    // We'll do part of this in class together!
}