CITS2200 Data Structures and Algorithms

Topic 3

Recursive Data Structures and Linked Lists

- Review of recursion: mathematical functions
- Recursive data structures: lists
- Implementing linked lists in Java
- Java and pointers
- Trees

Reading: Lambert and Osborne, Sections 10.1 and 5.3–5.4
1. Recursion

Powerful technique for solving problems which can be expressed in terms of smaller problems of the same kind.

eg. *Towers of Hanoi*

![Towers of Hanoi Diagram]

**Aim**: move all disks to the middle peg, moving one disk at a time, without ever putting a larger disk on a smaller one.

**Exercise**: Provide a recursive strategy for solving the Towers of Hanoi for arbitrary numbers of disks.
The Towers of Hanoi is also a good example of computational explosion.

It is alleged that the priests of Hanoi attempted to solve this puzzle with 64 disks. Even if they were able to move one hundred disks every second, this would have taken them more than 5,000,000,000 years!
1.1 Example: Common Mathematical Functions

Start with just increment and decrement...

// Class for doing recursive maths. Assumes all integers
// are non-negative (for simplicity no checks are made).

class RMaths {

    // method to increment an integer
    public static int increment(int i) { return i + 1; }

    // method to decrement an integer
    public static int decrement(int i) { return i - 1; }

    // more methods to come here...
}
**Note:** All methods are:

- **public** — any program can access (use) the methods
- **static** — methods belong to the class (*class methods*), rather than objects (instances) of that class

In fact, we are not using objects here at all.

*increment* and *decrement* take *int* arguments and return *ints*.
They are “called” by commands of the form `RMaths.increment(4)` — that is, the method `increment` belonging to the class `RMaths`.

```java
public class RMathsTest {
    // simple method for testing RMaths
    public static void main(String[] args) {
        System.out.println(RMaths.increment(4));
    }
}
```
**Addition**: express what it means to add something to \( y \) in terms of adding something to \( y - 1 \) (the decrement of \( y \))

\[
x + y = (x + 1) + (y - 1)
\]

/*
 * add two integers
 */

```java
public static int add(int x, int y) {
    if (y == 0) return x;
    else return add(increment(x), decrement(y));
}
```
**Multiplication**

\[ x \times y = x + (x \times (y - 1)) \]

```
/*
 * multiply two integers
 */
public static int multiply(int x, int y) {
    if (y == 0) return 0;
    else return add(x, multiply(x, decrement(y)));
}
```

Similar code can be written for other functions such as power and factorial
⇒ see Exercises
Recursive programs require:

- one or more *base cases* or *terminating conditions*
- one or more *recursive cases* or *steps* — routine “calls itself”

**Q**: What if there is no base case?
Recursion is:

- powerful — can solve arbitrarily large problems
- concise — code doesn’t increase in size with problem
- closely linked to the very important proof technique called \textit{mathematical induction}.
- not necessarily efficient
  - we’ll see later that the time taken by this implementation of multiplication increases with approximately the square of the second argument
  - long multiplication taught in school is approximately linear in the number of digits in the second argument
2. Recursive Data Structures

Recursive programs usually operate on recursive data structures

⇒ data structure defined in terms of itself

2.1 Lists

A list is defined recursively as follows:

- an empty list (or null list) is a list
- an item followed by (or linked to) a list is a list

Notice that the definition is like a recursive program — it has a base case and a recursive case!
Building a list...

null

link

a null

b a null

c b a null
2.2 List ADT

As an *abstract data type*, a list should allow us to:

1. Construct an empty list
2. Insert an element into the list
3. Look at an element in the list
4. Delete an element from the list
5. Move up and down the list

We will specify the List ADT more formally later . . .

For now, we will just look at a simple list that allows us to insert, delete, and examine only at the front of the list.
3. **A LinkedList Class in Java**

3.1 **The Links**

Defined recursively...

```java
// link class for chars
class LinkChar {
    char item; // the item stored in this link
    LinkChar successor; // the link stored in this link

    LinkChar (char c, LinkChar s) {item = c; successor = s;}
}
```

Notice that the constructor makes a new link from an item and an existing link.
3.2 The Linked List

Next we need an object to “hold” the links. We will call this LinkedListChar.

Contains a variable which is either equal to “null” or to the first link (which in turn contains any other links), so it must be of type LinkChar...

    class LinkedListChar {
        LinkChar first;
    }

Now the methods...
• **Constructing an empty list**

    ```java
    class LinkedListChar {
        LinkChar first;
        LinkedListChar () {first = null;}    // constructor
    }
    ```

    Conceptually, think of this as assigning a “null object” (a null list) to first. (Technically it makes first a null-reference, but don’t worry about this subtlety for now.)
• Adding to the list

class LinkedListChar {
    LinkChar first;
    LinkedListChar () {first = null;}

    // insert a char at the front of the list
    void insert (char c) {first = new LinkChar(c, first);}
}
first = null

first = \[ \text{a} \mid \text{null} \]

first = \[ \text{b} \mid \text{a} \mid \text{null} \]

first = \[ \text{c} \mid \text{b} \mid \text{a} \mid \text{null} \]
To create the list shown above, the class that uses `LinkedListChar`, say `LinkedListCharTest`, would include something like...

```java
LinkedListChar myList; // myList is an object
// of type LinkedListChar
myList = new LinkedListChar(); // call constructor to
// create empty list
myList.insert('a');
myList.insert('b');
myList.insert('c');
```
• Examining the first item in the list

    // define a test for the empty list
    boolean isEmpty () {return first == null;}

    // if not empty return the first item
    char examine () {if (!isEmpty()) return first.item;}

• Deleting the first item in the list

```java
void delete () {if (!isEmpty()) first = first.successor;
```

*first* then refers to the “*tail*” of the list.

Note that we no longer have a reference to the previous first link in the list (and can never get it back). We haven’t really “deleted” it so much as “abandoned” it. Java’s automatic *garbage collection* reclaims the space that the first link used.

⇒ This is one of the advantages of Java — in C/C++ we have to reclaim that space with additional code.
The Complete Program

```java
package DAT; // It’s part of Cara’s DAT package.

import Exceptions.*; // Use a package of
// exceptions defined elsewhere.

/**
 * A basic recursive (linked) list of chars.
 * @author Cara MacNish // Lines between /** and */ generate
 */ // automatic documentation.

public class LinkedListChar {
    /**
     * Reference to the first link in the list, or null if
     * the list is empty.
     */
    private LinkChar first; // Private – users cannot access
    // this directly.
```
/**
 * Create an empty list.
 */
public LinkedListChar () {first = null;}    // The constructor.

/**
 * Test whether the list is empty.
 * @return true if the list is empty, false otherwise
 */
public boolean isEmpty () {return first == null;}

/**
 * Insert an item at the front of the list.
 * @param c the character to insert
 */
public void insert (char c) {first = new LinkChar(c, first);}
/**
 * Examine the first item in the list.
 * @return the first item in the list
 * @exception Underflow if the list is empty
 */
public char examine () throws Underflow {
    if (!isEmpty()) return first.item;
    else throw new Underflow("examining empty list");
}

// Underflow is an example of an exception that
// occurs (or is ‘thrown’) if the list is empty.

/**
 * Delete the first item in the list.
 * @exception Underflow if the list is empty
 */
public void delete () throws Underflow {
    if (!isEmpty()) first = first.successor;
    else throw new Underflow("deleting from empty list");
}
/**
 * construct a string representation of the list
 * @return the string representation
 */

public String toString () {
    LinkChar cursor = first;
    String s = "";
    while (cursor != null) {
        s = s + cursor.item;
        cursor = cursor.successor;
    }
    return s;
}
4. Java and Pointers

Conceptually, the successor of a list is a list.

One of the great things about Java (and other suitable object oriented languages) is that the program closely reflects this “theoretical” concept — from a programmer’s point-of-view the successor of a LinkChar is a LinkChar.

Internally, however, all instance variables act as references, or “pointers”, to the actual data.
Therefore, a list that looks conceptually like

\[
\text{first} = \begin{array}{c}
\text{c} \\
\text{b} \\
\text{a} \\
\text{null}
\end{array}
\]

internally looks more like

\[
\begin{array}{c}
\text{first} \\
\text{c} \\
\text{b} \\
\text{a} \\
\text{null}
\end{array}
\]

For simplicity of drawing, we will often use the latter type of diagram for representing recursive data structures.
5. Trees

A tree is another example of a recursive data structure. It might be defined as follows:

- a null tree (or empty tree) is a tree
- an item followed by one or more trees is a tree

Some examples of trees:

- family trees
- XML files
- inheritance hierarchies
Graphical representations...

Tree representation:

```
  tree = a
       /   \
      b   null  
      /\        
     b null null 
    / \          
   c   null      
```

More on trees later.
6. Summary

Recursive data structures:

- can be arbitrarily large
- support recursive programs
- are a fundamental part of computer science — they will appear again and again in this and other units

⇒ You need to understand them. If not, seek help!

We will see many in this unit, including more on lists and trees.