

**CITS2200**  
**Data Structures and Algorithms**

Cara MacNish

School of Computer Science & Software Engineering  
University of Western Australia

## Topic 3

# Recursive Data Structures and Linked Lists

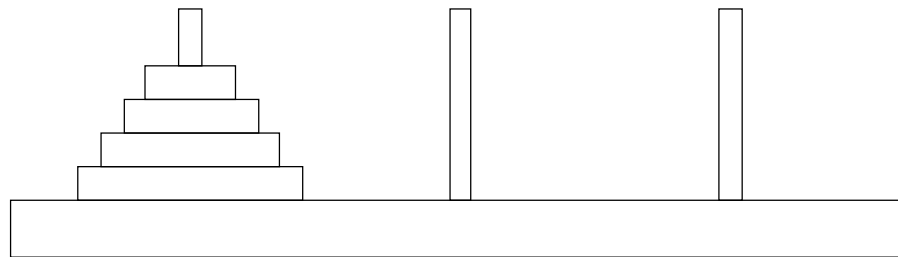
- Review of recursion: mathematical functions
- Recursive data structures: lists
- Implementing linked lists in Java
- Java and pointers
- Trees

Reading: L & O, Sections 10.1, 5.3–5.4

## 3.1 Recursion

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

eg. **Towers of Hanoi**



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

### 3.1.1 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).

public 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 `int`'s.

They are “called” by commands of the form

```
RMaths.increment(4)
```

— that is, the method `increment` belonging to the class `RMaths`.

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

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?

# 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  $\Rightarrow$  see Exercises

## Recursion is:

- powerful — can solve arbitrarily large problems
- concise — code doesn't increase in size with problem
- closely linked to very important proof technique called **mathematical induction**
- basis of **logic programming** and **functional programming**  
(logic program to solve 'Towers of Hanoi' takes just two lines!)

- 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

## 3.2 Recursive Data Structures

Recursive programs usually operate on **recursive data structures**

⇒ data structure **defined in terms of itself**

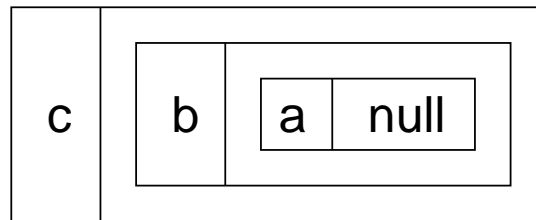
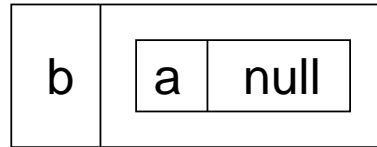
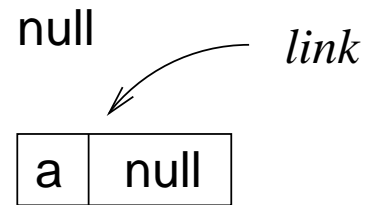
### 3.2.1 Lists

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 the definition is like a recursive program — it has a base case and a recursive case!

# Building a list...



## 3.3 A LinkedList Class in Java

### 3.3.1 The Links

Defined recursively...

```
// 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 constructor makes a new link from an item and an existing link.

## 3.3.2 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**

```
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 = 

a	null
---	------

first = 

b	<table border="1"><tr><td>a</td><td>null</td></tr></table>	a	null
a	null		

first = 

c	<table border="1"><tr><td>b</td><td><table border="1"><tr><td>a</td><td>null</td></tr></table></td></tr></table>	b	<table border="1"><tr><td>a</td><td>null</td></tr></table>	a	null
b	<table border="1"><tr><td>a</td><td>null</td></tr></table>	a	null		
a	null				

To create the list shown above, the class that **uses** `LinkedListChar`, say `LinkedListCharTest`, would include something like...

```
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**

```
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.



```
/**
 * 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.
    // In this case it occurs (or is ‘‘thrown’’)
    // if the user tries to examine an empty list.
```

```
/**
 * 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");
}
```

```
        // Many classes provide a string representation
        // of the data, for example for printing,
        // defined by a method called ‘‘toString()’’.
/**
 * 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;
}
}
```

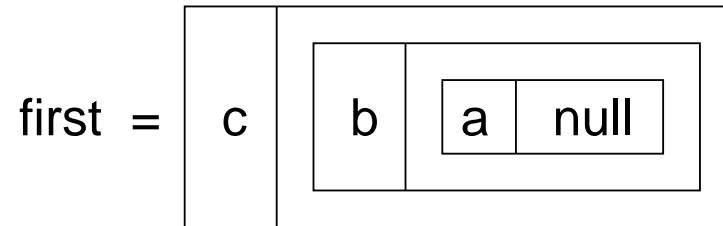
## 3.4 Java and Pointers

Conceptually, the successor of a list **is** a list.

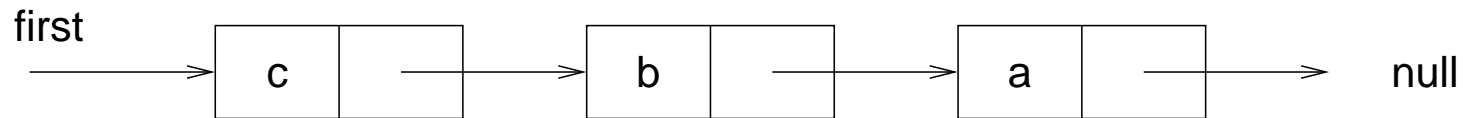
One of the great things about Java (and other suitable object oriented languages) is that the program closely reflects this “theoretical” concept — from a programmer’s point-of-view the successor of a `LinkChar` **is** a `LinkChar`.

Internally, however, all instance variables act as **references**, or “**pointers**”, to the actual data.

Therefore, a list that looks conceptually like



internally looks more like



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

### 3.4.1 Freedom from Pointers

While Java uses references or pointers internally, the programmer is freed from the task of having to manipulate them. This is in contrast to many traditional languages (eg Pascal, C, C++) where pointers must be explicitly handled by the programmer.

Example: Pascal

```
type linktype = ^celltype;
   celltype = record
       item: char;
       successor: linktype
   end;

First = linktype;
```

A procedure to insert an item looks like:

```
procedure insert(c: char; var l: First);
var p: linktype;
begin
    new(p);
    p^.item := c;
    p^.successor := l;
    l := p;
end;
```

Compare this to:

```
void insert (char c) {first = new Link(c, first);}
```

Java allows us to **abstract** away from the details.

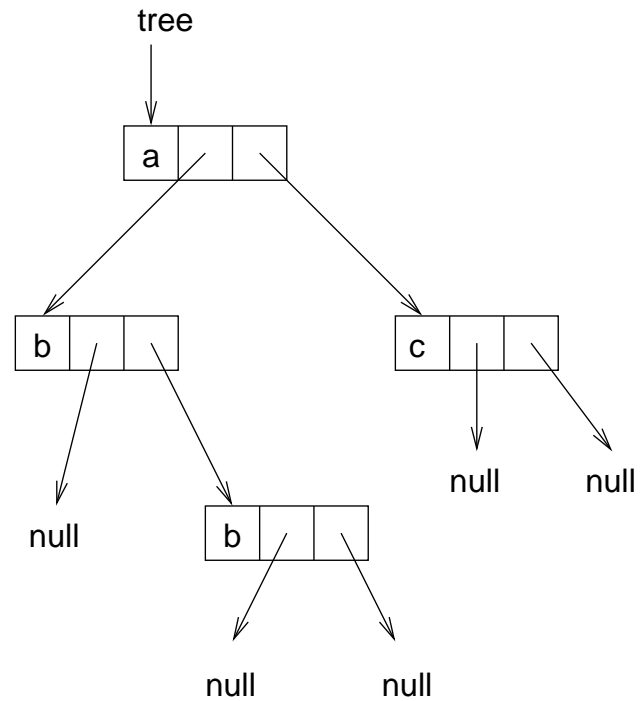
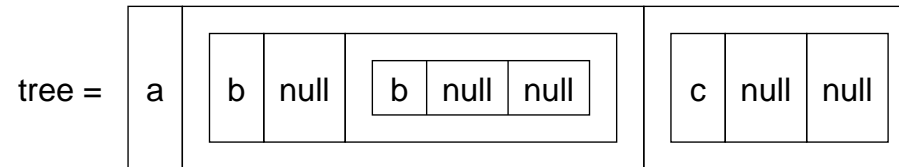
## 3.5 Trees

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

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

[Some examples of trees — see Wood p142]

# Graphical representations...



More on trees later.

## 3.6 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 courses

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

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