Recursion

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

In the previous lecture we saw recursion in the QuickSort Algorithm:

\[
\text{procedure QUICKSORT}(A, p, r) \\
\text{if } p < r \\
\text{ then } q \leftarrow \text{PARTITION}(A, p, r) \\
\text{ QUICKSORT}(A, p, q - 1); \text{ QUICKSORT}(A, q + 1, r)
\]

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 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

Recursion is:

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

Reading: Weiss, Chapter 6 and 7

2. Recursive Data Structures

Recursive programs usually operate on recursive data structures

⇒ data structure defined in terms of itself

2.1 Lists

A list may be 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...

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 simpler data structure called a stack that allows us to insert, delete, and examine only the first element in the list.

3. A Stack Class in Java

3.1 The Links

Defined recursively...

```java
// link class for chars
class Link {
  Object item; // the item stored in this link
  Link successor; // the link stored in this link

  Link (Object c, Link 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 `LinkedStack`.

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 `Link`...

```java
class LinkedStack {
  Link first;
}
```

Now the methods...
• Constructing an empty stack

class LinkedStack {
    Link first;

    LinkedStack () {first = null;}  // constructor
}

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

• Adding to the stack

class LinkedStack {
    Link first;

    LinkedStack () {first = null;}

    // insert a char at the front of the list
    void insert (Object o) {first = new Link(o, first);}
}

To create the stack shown above, the class that uses LinkedStack, say LinkedStackTest, would include something like...

LinkedStack myStack;  // myStack is an object of type LinkedStack
myStack = new LinkedStack();  // call constructor to create empty stack
myStack.insert('a');
myStack.insert('b');
myStack.insert('c');
• Examining the first item in the Stack

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

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

• Deleting the first item in the stack

```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 stack (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
import CITS2200.*; // Use a package of
// exceptions defined elsewhere.

public class LinkedStack {
  private Link first; // Private - users cannot access
  // this directly.

  public LinkedStack() {first = null;} // The constructor.

  public boolean isEmpty () {return first == null;}

  public void insert (Object o) {first = new Link(o, first);}

  public Link examine () {if (!isEmpty()) return first;} // The test.

  public void delete () {if (!isEmpty()) first = first.successor;}

  // Other methods...
}
```

/* Create an empty stack.
 */
public LinkedStack() {first = null;} // The constructor.

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

/*
 * Insert an item at the front of the stack.
 * @param o the Object to insert
 */
public void insert (Object o) {first = new Link(o, first);}
4. Java and Pointers

Conceptually, the successor of a stack is a stack.

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 Link is a Link.

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

Therefore, a list that looks conceptually like

```
first = c | b | a | null |
```

internally looks more like

```
first

  c ——— b ——— a ——— null
```

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

The above class provides:

- data, and instructions to access it
- “higher-level” role of the program

We specified our stack so that we always added, examined and deleted at the front of a sequence.

We could change these operations slightly so that we always examined and deleted from the front of the sequence, but added at the back of the sequence.

This is just what a queue, or FIFO (first-in, first-out buffer), does!

Operations on a stack:

1. Create an empty stack
2. Test whether the stack is empty
3. Add (push) a new element on the top
4. Examine (peek at) the top element
5. Delete (pop) the top element

In general, the operations on a queue include:

1. Create an empty queue
2. Test whether the queue is empty
3. Add (enqueue) a new latest element
4. Examine the earliest element
5. Delete (dequeue) the earliest element
6. Specifying ADTs

We saw in Topic 1 that ADTs consist of a set of operations on a set of data values. We can specify ADTs by listing the operations (or methods).

The lists of operations on the previous pages are very informal and not sufficient for writing code. For example

2. Test whether the queue is empty

doesn’t tell us the name of the method, what arguments it is called with, what is returned, and whether it can throw an exception.

Thus, a Queue ADT might be specified by the following operations:

1. `Queue()`: create an empty queue
2. `isEmpty()`: return `true` if the queue is empty, `false` otherwise
3. `enqueue(e)`: e is added as the last item in the queue
4. `examine()`: return the first item in the queue, or throw an exception if the queue is empty
5. `dequeue()`: remove and return the first item in the queue, or throw an exception if the queue is empty

Similarly, the specification of a Stack ADT:

1. `Stack()`: create an empty stack
2. `isEmpty()`: return `true` if the stack is empty, `false` otherwise
3. `push(e)`: item e is pushed onto the top of the stack
4. `peek()`: return the item on the top of the stack, or throw an exception if the stack is empty
5. `pop()`: remove and return the item on the top of the stack, or throw an exception if the stack is empty

Note: The use of upper and lowercase in method names should follow the rules described in the document *Java Programming Conventions*. 
7. 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.