Data Structures

Read Chapter 3 of Weiss

Firstly, an overview.

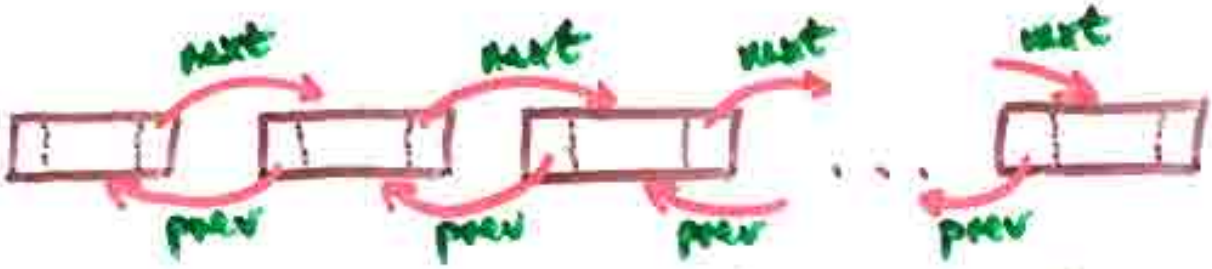
1. Lists

A list is a collection of things in order. We refer to a *linked list* when the 'link' points to the next object in the list.

Either

A singly linked list

or



A doubly (two-way) linked list

Each list node contains information about what lives there and where to go next (and for two-way lists, where it came from).

```
class ListNode
  {
    Object data;
    ListNode next;
  }
```

this is for a one-way list. For a two-way list, we need:

ListNode next, prev;

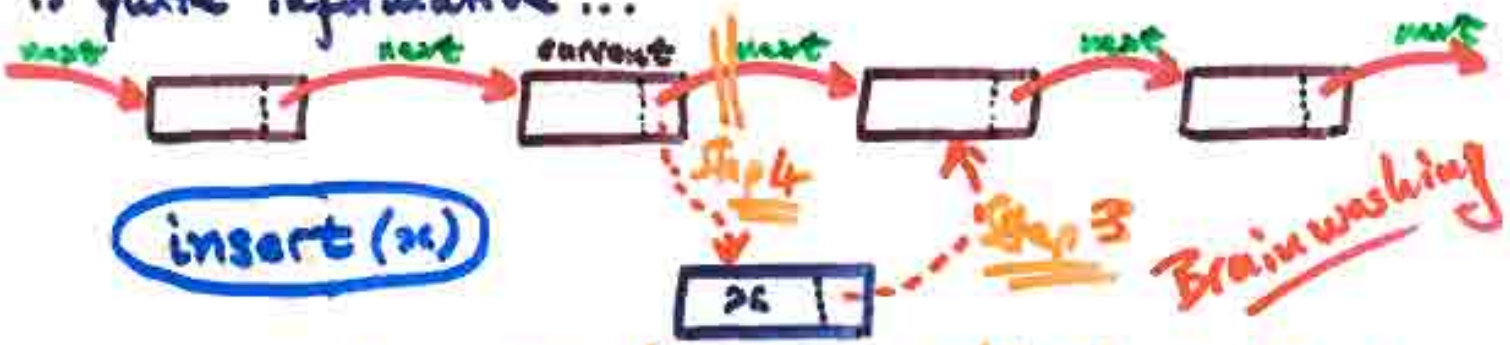
One approach commonly taken (but which has some serious weaknesses) is to set up an interface ...

```
import Exceptions.*;          <-- your personal collection of favourite exceptions

public interface Lister
  {
    boolean isEmpty();
    void makeEmpty();
    void insert(Object x)   throws ItemNotFound;     <-- in case we're in a bad place
    boolean find(Object x)  throws ItemNotFound;
    void remove(Object x)   throws ItemNotFound;
    Object retrieve()       throws ItemNotFound;     <-- return item in current position
    boolean isInList();
    void zeroth();          <-- set current position prior to first!
    void first();           <-- set current position to first
    void advance();         <-- move current position to next
  }
```

then the implementation ...

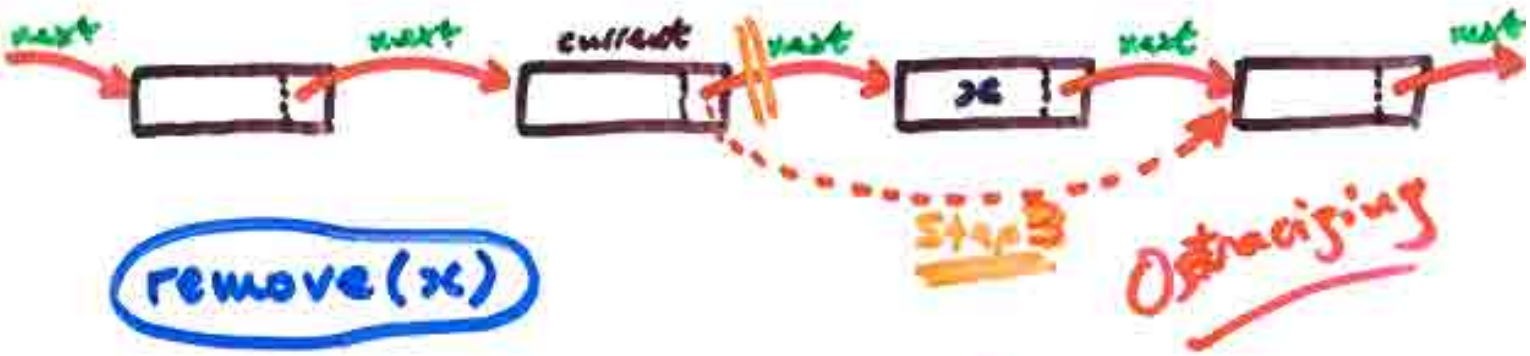
```
public class AList implements Lister
  {
    ListNode current;
    various implementations of the above methods, e.g.:
    void advance()
      { current = current.next; }     <-- caveats about current being null should be here!
    various constructors ...
  }
```

How will these methods be implemented? For now this is of little concern to us, although getting an intuitive picture of how insert and remove will operate is quite informative ...



insert(x)

1. create a new object to insert
2. look at "current" and find its next
3. make the new object's next point there
4. make current's next point to new object



remove(x)

1. find "current" whose next is x
2. find current's next's next
3. make current's next point there

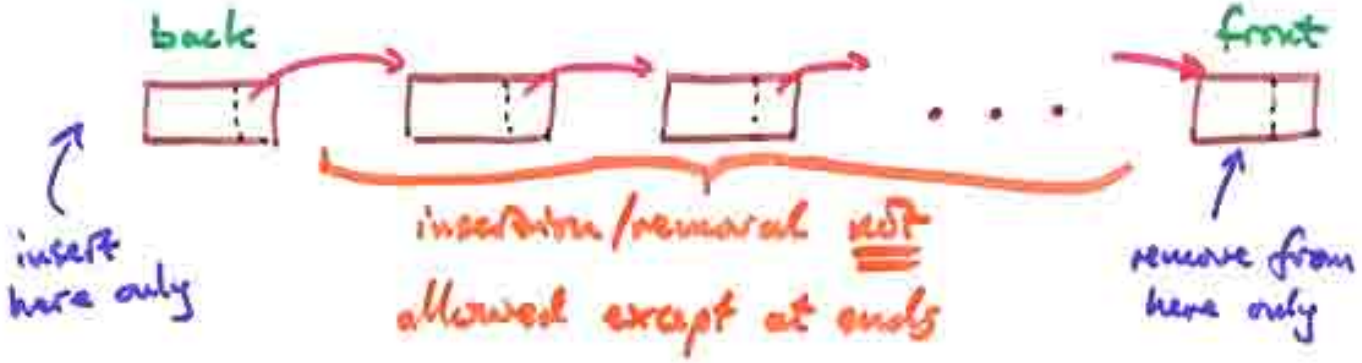
The obvious extra steps would be needed if working with a two-way linked list.

It's time now to get into the details ...

Three close relatives of linked lists are ...

2. Queues

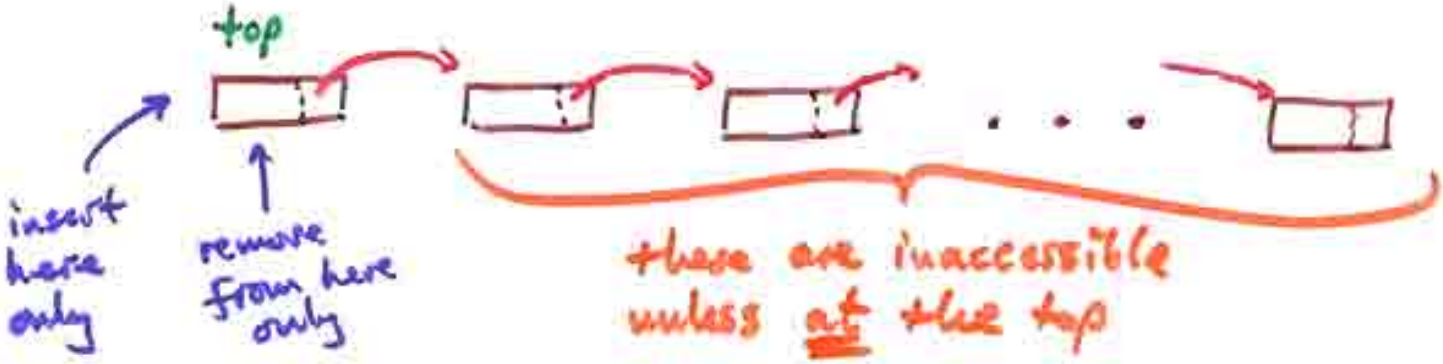
In a natural sense, a queue is an associated list which allows insertion at one end and removal from the other ...



This is fun to implement as an array as well (try aiming for a 'circular' array using modular/remainder arithmetic).

3. Stacks

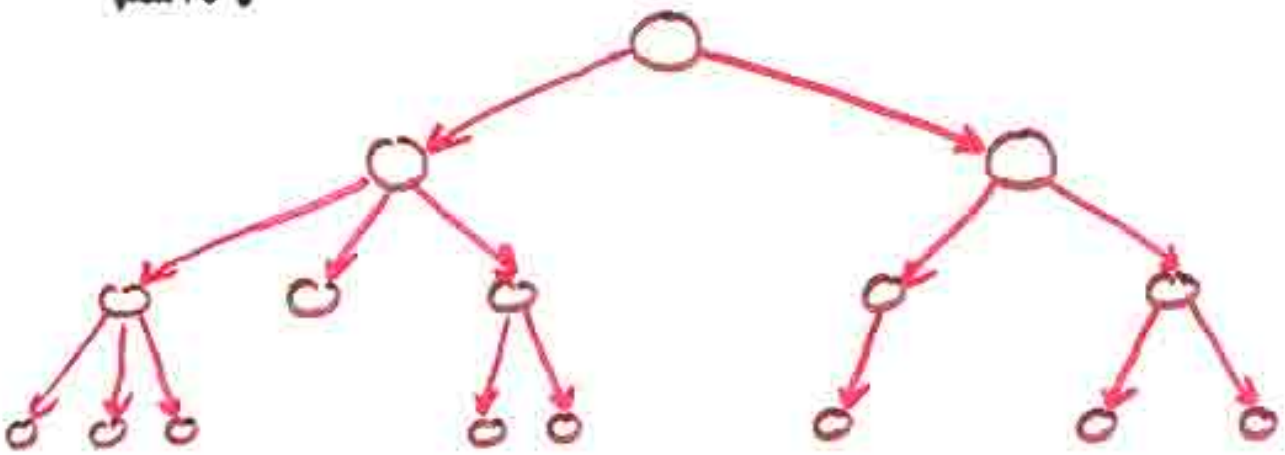
In a natural sense, a stack is a really associated list which only allows insertion and removal from one and the same end!



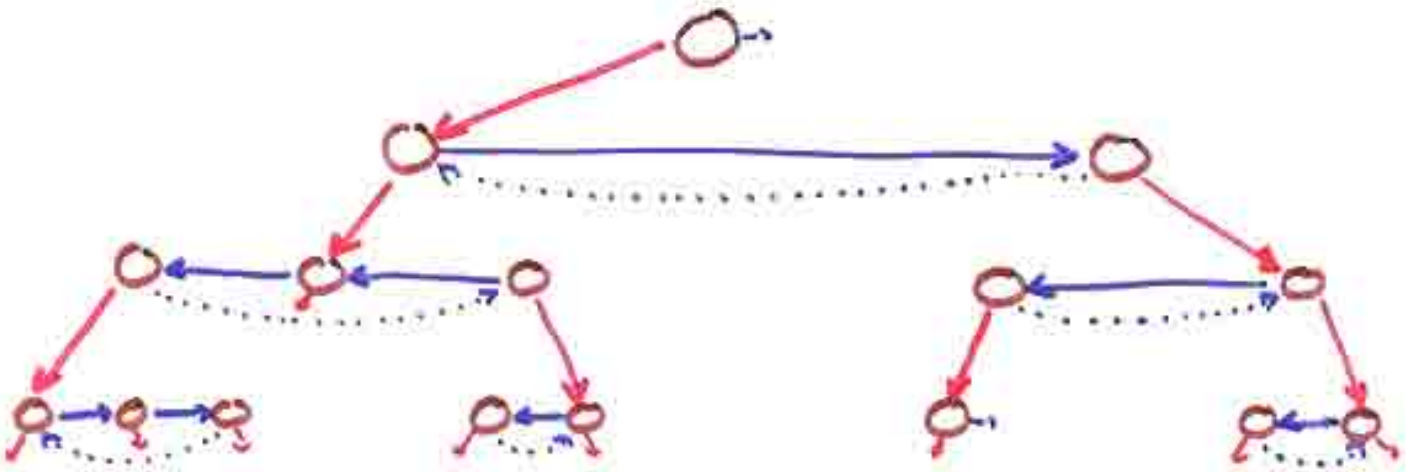
Stacks are used heavily in many areas of program compilation, use and design.

41. Trees

In a natural sense, a tree is a one-way linked list where each node may have multiple nexts!



Although this approach can work well for binary trees, in the general case illustrated above it's often better not to think of each parent node having multiple children, but rather of each parent node having a favourite child and a favourite sibling (to the parent).



In this equivalent tree, the red arrows point to the child and the purple arrows to the sibling. It's easy to see that we can still navigate through this representation of the tree.

Thinking of an array as a kind of list, it would be very easy and quick to implement most of these operations. However, insert/remove would be messy — either of these happening near the front of an array would require the copying/moving of most of the array. On average we'd have to move ½ the list, an O(N) operation.

The real problem with having everything in just one interface is that we often need to manage two or more current positions in the same list, for example when sorting. So we'll set up two interfaces, one to deal with global statements about the list, and the other to handle local actions within the list (eg for iterating along the list). Then writing an implementation which really exploits the linking of nodes will be far more natural ...

```
package DataStructures ;          ← to hold all of our work for later use
class ListNode
  {
    Object data;
    ListNode next;

    ListNode (Object theData, ListNode theNode)
      { data = theData ; next = theNode ; }
    ListNode (Object theData)
      { this (theData, null) ; }
    ListNode ()
      { this (null) ; }
  }
```

We could have a single list class which implements both the List and ListIter interfaces, however there is a philosophical and practical distinction between an actual "list" and the processes which move us around a the list, so it may prove convenient to implement these seperately. Maintaining this distinction allows us to implement a nice split of 'functionality'...

```
package DataStructures;

import Exceptions.*;

public interface List
  { boolean isEmpty();
    void makeEmpty();
    void insert (Object x, ListIter p ) throws
                                    ItemNotFound;
    ListIter find (Object x)
    void remove (Object x) throws ItemNotFound;

    ListIter zeroth();
    ListIter first();
  }
```

we lose the 'current' position since we want to use the Iterator class to maintain several currents!

this way we can say explicitly where we put x, rather than just after current.

throwing exceptions can be expensive

we don't throw an exception here since failure isn't exceptional!!

these will make sense looking at the implementation

The differences between this & the Lister interface introduced earlier have been marked. The main benefit of maintaining a seperation between a list and

a list_iteration is that, if needed, we can instantiate list_Iteration several times for a given list, i.e., we can maintain several 'current' positions in one list.

```
package DataStructures;
public interface ListItr
   {
      boolean isInList();
      Object  retrieve();
      void    advance();
   }
```

Now for the implementations ...

```
package DataStructures;
public class LListItr implements ListItr
   {
      ListNode current;
      LListItr (ListNode theNode)
         { current = theNode; }
      public boolean atEnd()
         { return current.next == null; }
      public boolean isInList()
         { return current != null && current.data != null; }
      public Object retrieve()
         { return isInList() ? current.data : null; }
      public void advance()
         { if (current != null) current = current.next; }
   }
```

so the LListItr class holds 'the' current position in the list.

the constructor 'creates' a current position !!

extra method to make life easier later

this at most goes to the null just beyond the end

```
package DataStructures
import Exceptions.*;
public class LList implements List
  {
    private ListNode header;

    public LList()
      { header = new ListNode(null); }

    public boolean isEmpty()
      { return header.next == null; }
    public void makeEmpty()
      { header.next = null; }

    public void insert(Object x, ListItr p)
                        throws ItemNotFound
      {
        if (p == null || p.current == null)
          throw new ItemNotFound("bad insertion locn");
        p.current.next = new ListNode(x, p.current.next);
      }
```

so the LList class only holds the header for the list

Let's illustrate this insert slowly with a fresh list...

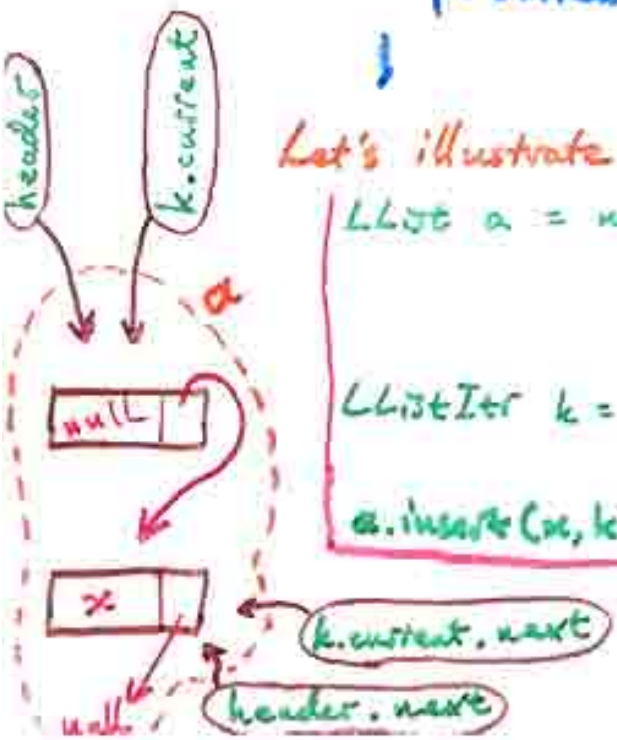
```
LList a = new LList();
```


```
LListItr k = new LListItr(a.header);
a.insert(x, k);
```

is k == null?... NO

is k.current == null?... NO

k.current.next = new (data = x
                       next = k.current.next)



```
public LListItr zeroth()
  { return new LListItr(header); }

public LListItr first()
  { return new LListItr(header.next); }

public LListItr find(Object x)
  {
    LListItr k = first();
    while (k.isInList() && !k.current.data.equals(x))
      k.advance();
    return k;
  }
```

So zeroth().current == header

a bad thing here is that we have 2 testings for isInList — better to have an "advance" & a "safeAdvance".

Now for a extra method to simplify building "remove".

```
public LListItr findPrev(Object x)
  {
    LListItr k = zeroth();
    while(!k.atEnd() && !k.current.next.data.equals(x))
      k.advance();
    return k;
  }

public void remove(Object x) throws ItemNotFound
  {
    LListItr p = findPrev(x);
    if (p.atEnd())
      throw new ItemNotFound("bad remove");
    p.current.next = p.current.next.next;
  }
}
```

End of LList class

Notice that for our linked list implementation, most of the operations are O(1), since in these cases only a fixed number of instructions are executed (independent of list size). The exceptions to this are find(x) and remove(x), which uses findPrev(x), since these find routines will be O(N) in both worst case and average case scenarios.

When we first introduced lists, we talked also of stacks and queues. Their linked list implementations are now easy, but for illustration we'll look at a (linked) Stack. Recall...

```
package DataStructures;
public interface Stack
  {
    void push (Object x);
    void pop () throws Underflow;
    Object top()
    Object topAndPop()
    boolean isEmpty();
    void makeEmpty();
  }
```

put x on top of the Stack

remove the top element from the Stack

find out what is on top of the Stack

```
package DataStructures;
public class LStack implements Stack
  {
    private ListNode topElt;

    public LStack() { topElt = null; }

    public boolean isFull { return false; }
```

as before for lists, this has an Object data and a ListNode next plus two natural constructors

refreshingly simple!

not in the interface, but a distinction from arrays!

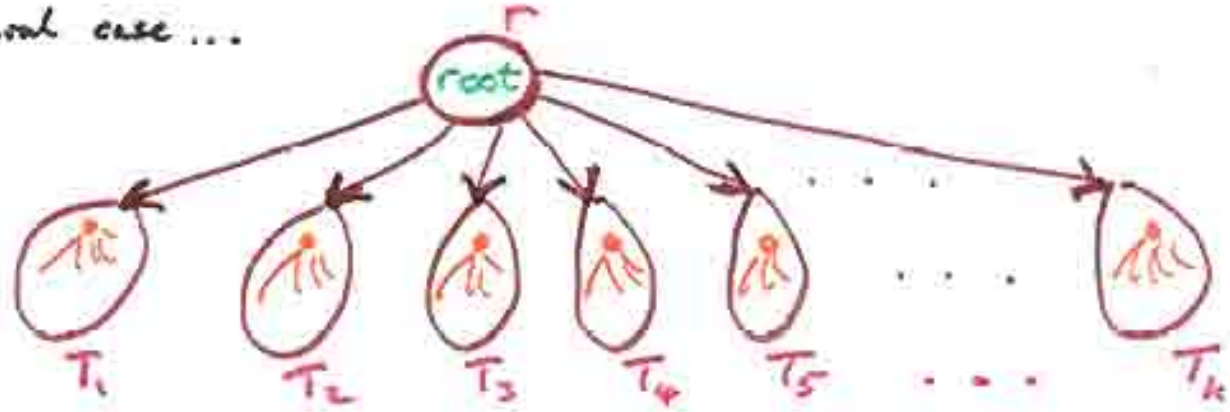
```
public void push (Object x)                    ← remember this is a Stack
   { topElt = new ListNode (x, topElt); }

public void pop () throws Underflow
   { if (isEmpty()) throw new Underflow ("Empty!");
     topElt = topElt.next;
   }

public Object top()
   { if (isEmpty()) return null;
     return topElt.data;
   }

public Object topAndPop()
   { if (isEmpty()) return null;

     Object temp = topElt.data;
     topElt = topElt.next;
     return temp;
   }

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

public void makeEmpty() { topElt = null; }
}
```

As is clear, this is a painless implementation, and all operations are O(1), although as is always the tradeoff, repeated calls to new incur their own associated expense. Notice that our implementation of a Stack had no use for a 'header' node (whose sole function is to point to the first 'real' node).

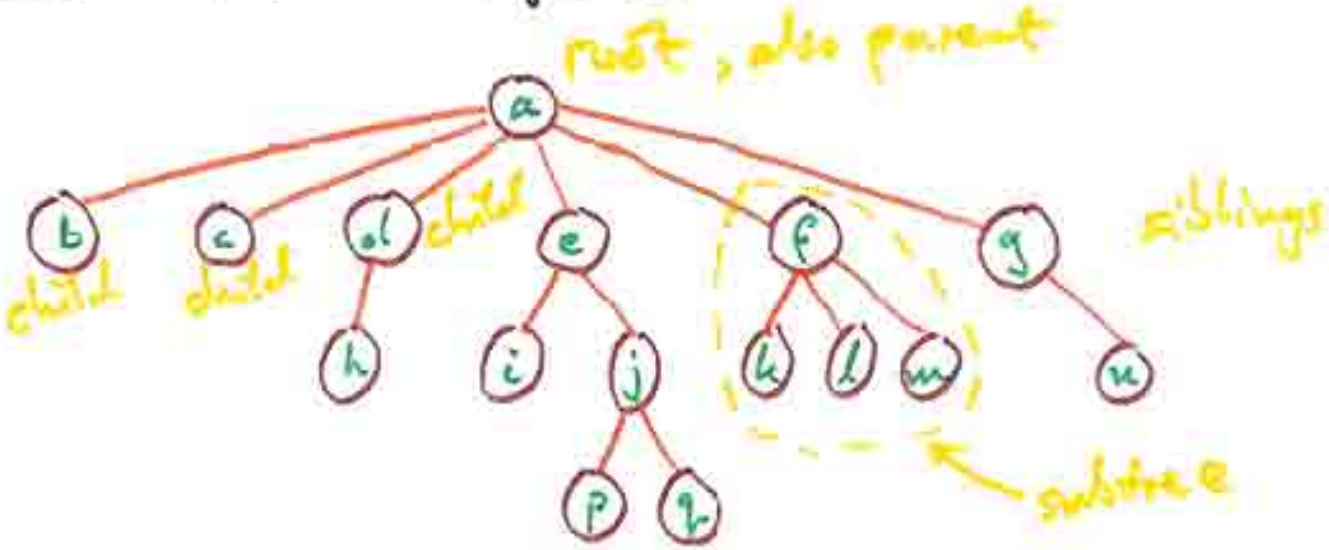
Trees

Read chapter 4 in Weiss

Let's look in a little more detail at trees. First the general case ...



A natural, recursive, definition has a tree with a root node r and subtrees $T_1, T_2, \ldots, T_k$ connected to r by directed edges from r. Then, of course, each of these subtrees has a root node r_i' with its own collection of subtrees $T_{i1}', T_{i2}', \ldots, T_{ik_i}'$, etc.! Playing this out, if a tree has a total of N nodes (including the root node), then it has $N-1$ edges ...



We can think of each subtree as starting with its root as a parent, and each node which is only one edge down from the root as being a child; the collection of these particular 'child' nodes being siblings.

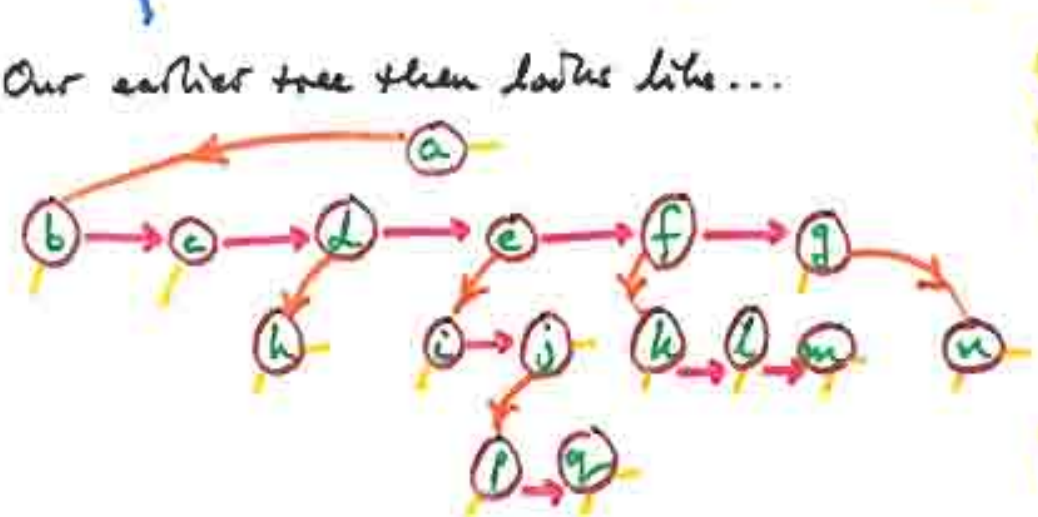
We define a path in a (directed) tree to be a sequence of nodes $n_1, n_2, \ldots, n_k$ where each n_i is the parent of n_{i+1}. Notice that this definition of a path doesn't allow any upwards travel, so there are no paths (per se) connecting seperate branches. The length of a path is the number of edges in that path. (Technically, we can say that there is a path of length 0 from a node to itself.)

More definitions. For any given node n, the depth of n is the length of the (unique) path from the root node to n. A leaf is a terminal node (i.e., all its children are null). The height of n is the length of the longest path from n to a leaf. Naturally, the height of a tree is the height of its root, and the depth of a tree is the depth of its deepest leaf (of course here depth of tree = height of tree).

Following our list example, we build ...

```
class TreeNode
  { Object data;
    TreeNode firstChild, nextSibling;
  }
```

Our earlier tree then looks like...



This gives one version of how we might traverse a tree. This approach can work well for general trees, where the number of children per node can vary greatly. However, for binary trees we can maintain direct links to both children of each node. As a more detailed example, we'll look at binary search trees ...

```
package DataStructures;
class BNode
   {
      Comparable data;
      BNode left;
      BNode right;

      BNode (Comparable thing, BNode l, BNode r)
         { data = thing; left = l; right = r; }
      BNode (Comparable thing)
         { this (thing, null, null); }
   }
```

```
package DataStructures;
public interface BSTree
   {
      void makeEmpty();
      boolean isEmpty();
      Comparable find (Comparable x);
      Comparable findMin();
      Comparable findMax();
      void insert (Comparable x);
      void remove (Comparable x);
      void printTree();
   }
```

This is then implemented as ...

```
package DataStructures;
public class LBSTree implements BSTree
  {
    private BNode root;

    public LBSTree () { root = null; }

    public void makeEmpty () { root = null; }
    public boolean isEmpty () { return root == null; }

    private Comparable dataAt ( BNode a )
      { return a == null ? null : a.data; }
    private BNode find ( Comparable x, BNode a )
      { if ( a == null ) return null;
        if ( x.compareTo(a.data) < 0 )
            return find ( x, a.left );
        else if ( x.compareTo (a.data) > 0 )
            return find (x, a.right);
        else return a; // found it!
      }
    private BNode findMin ( BNode a )
      { if (a == null) return null;
        else if ( a.left == null ) return a;
        return findMin ( a.left );
      }
    private BNode findMax ( BNode a )
      { if ( a != null )
          while ( a.right != null )
            a = a.right;
        return a;
      }
```

Some methods useful later

these private methods will be used when defining the public methods in our interface

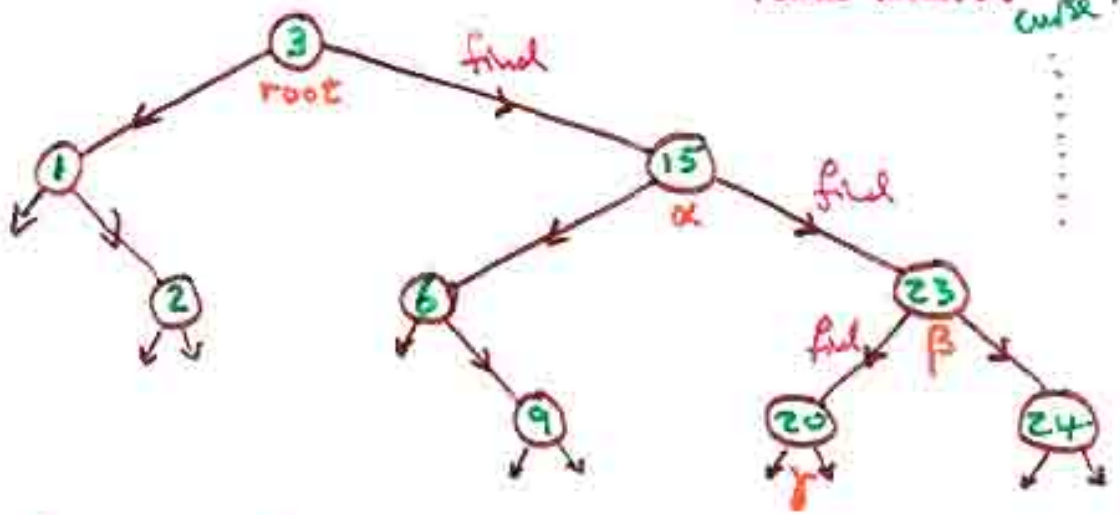
recursive flavour

non-recursive flavour

```
public Comparable find (Comparable x)
    { return dataAt ( find (x, root) ); }
public Comparable findMin ()
    { return dataAt ( findMin (root) ); }
public Comparable findMax ()
    { return dataAt ( findMax (root) ); }
```

Before we continue, let's see how 'find' works on a sample binary tree ...

find(20);
return dataAt(recurse)



returns 20

The recursion here is ...

return dataAt(find (20, root)) nothing returned yet — recursion opens find
↳ if ✱ if ✱
if ✓ return find (20, α) nothing returned yet - recursion opens find
returns γ
if ✱ if ✱
if ✓ return find (20, β) nothing returned yet - recursion opens find
↳ if ✱
if ✓ return find (20, γ) nothing returned yet
↳ if ✱ if ✱ if ✱
else return γ;
returns γ
returns γ
now we return γ

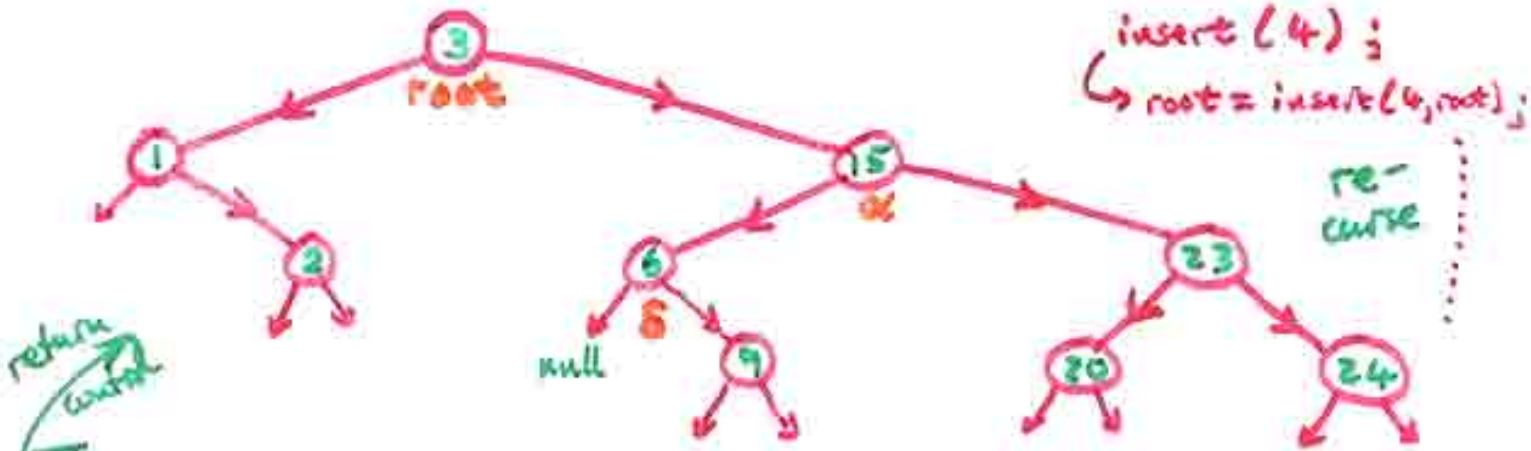
follow the arrow directions carefully — no 'returning' happens until we're at the bottom, and it's the bottom (successful) node which is returned all the way.

```
private BNode insert (Comparable x, BNode a)
{
    if (a == null)      a = new BNode(x);
    else if (x.lessThan(a.data))
                        a.left = insert(x, a.left);
    else if ((a.data).lessThan(x))
                        a.right = insert(x, a.right);
    else ;      // duplicate entry, so do nothing
    return a;
}
public void insert (Comparable x)
  { root = insert(x, root); }
```

as in the earlier 'find', would it be more efficient to evaluate 'compareTo' once and then use that value if there are multiple ?

the only reason we assign to 'root' here is to handle the case starting with an empty tree.

Let's see how the recursion for 'insert' works ...



The recursion here is ...

root = insert(4, root) no change yet to root — recursion opens insert

↳ if ✱ if ✱ if ✓ root.right = insert(4, x)

↳ if ✱ if ✓ x.left = insert(4, δ)

↳ if ✱ if ✓ δ.left = insert(4, null)

↳ if ✓ return new BNode(4);

return control

return control

return control

following the arrows here shows that only at the very bottom, when a new node is created, is there any real change due to =

real change

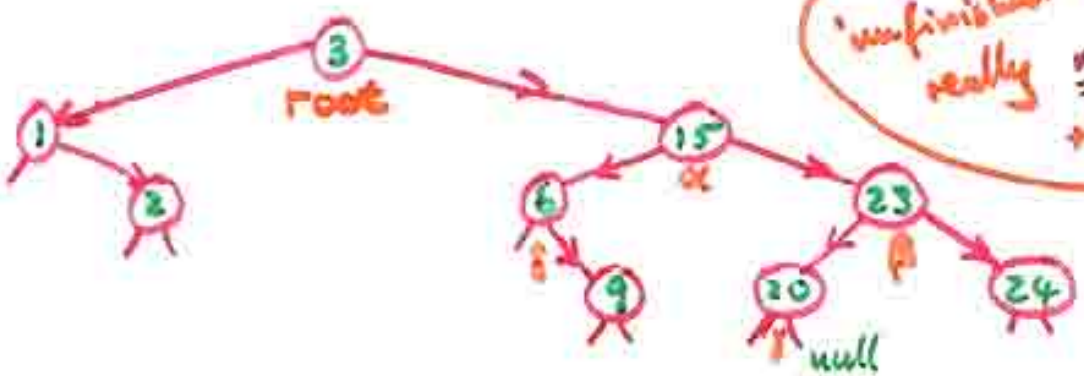
```
private BNode remove (Comparable x, BNode a)
{
  if (a == null) return a; // ast there to remove !!
  if (x.lessThan(a.data))
        a.left = remove (x, a.left);
  else if ( (a.data).lessThan(x) )
        a.right = remove (x, a.right);
  else if ( a.left != null && a.right != null)
  { a.data = findMin (a.right).data;
    a.right = remove (x, a.right);
  }
  else
    a = (a.left != null)? a.left : a.right;
  return a;
}
public void remove ( Comparable x)
  { root = remove (x, root); }
```

case where 'a' has two children →

case where 'a' has only 1 child or has 0 children →

if 0 children then a ← null
if 1 child then a ← that child

Let's see what's going on with 'remove'...



Note that all the 'unfinished' 'spots' are really null, not just the one marked.

If left remove null

Removing a leaf, such as 20, is easy...

remove (20) → root = remove (20, root) → if * if * if ✓

all these return control with no changes

root.right = remove (20, α) → if * if * if ✓

α.right = remove (20, β) → if * if ✓

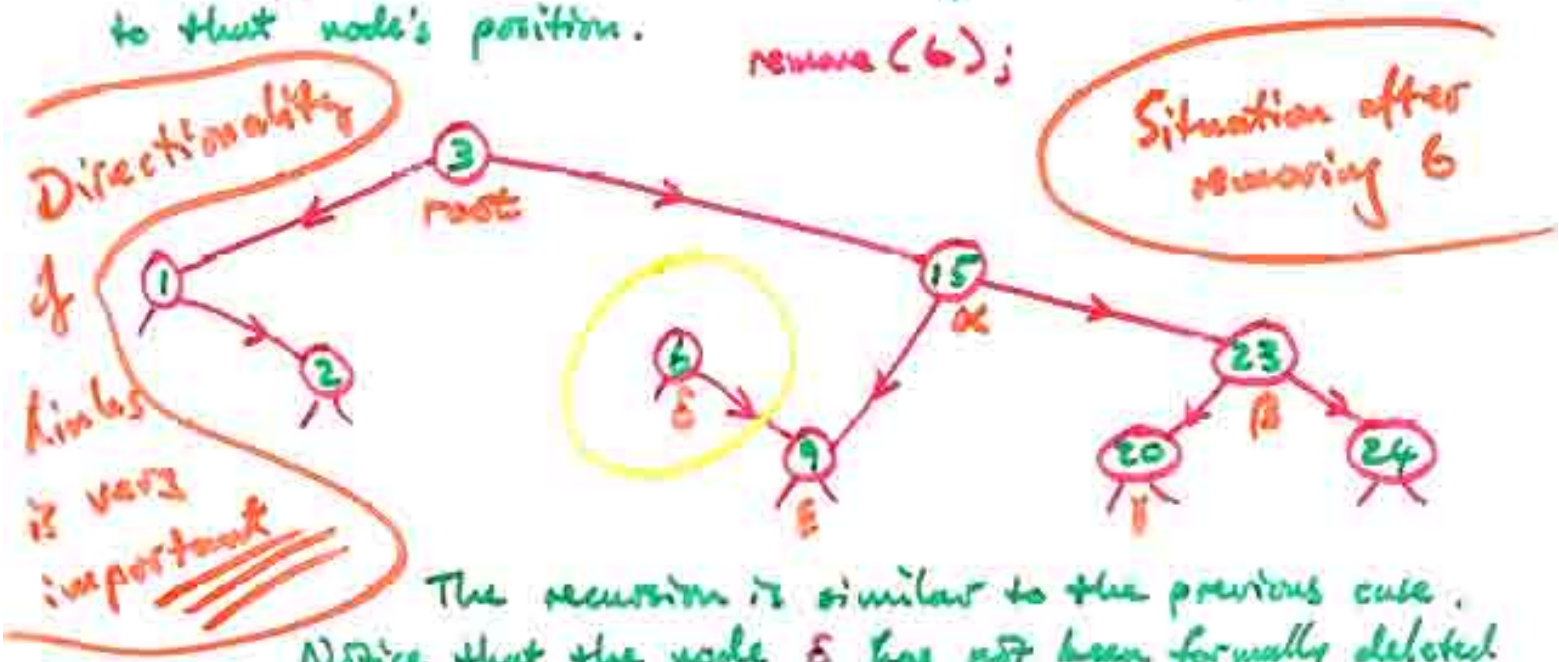
β.left = remove (20, γ) → if * if * if * if * else ✓

return γ

γ = null;

Removing a node which has only 1 child, such as 6, is also easy — the non-trivial subtree of that node gets promoted to that node's position.

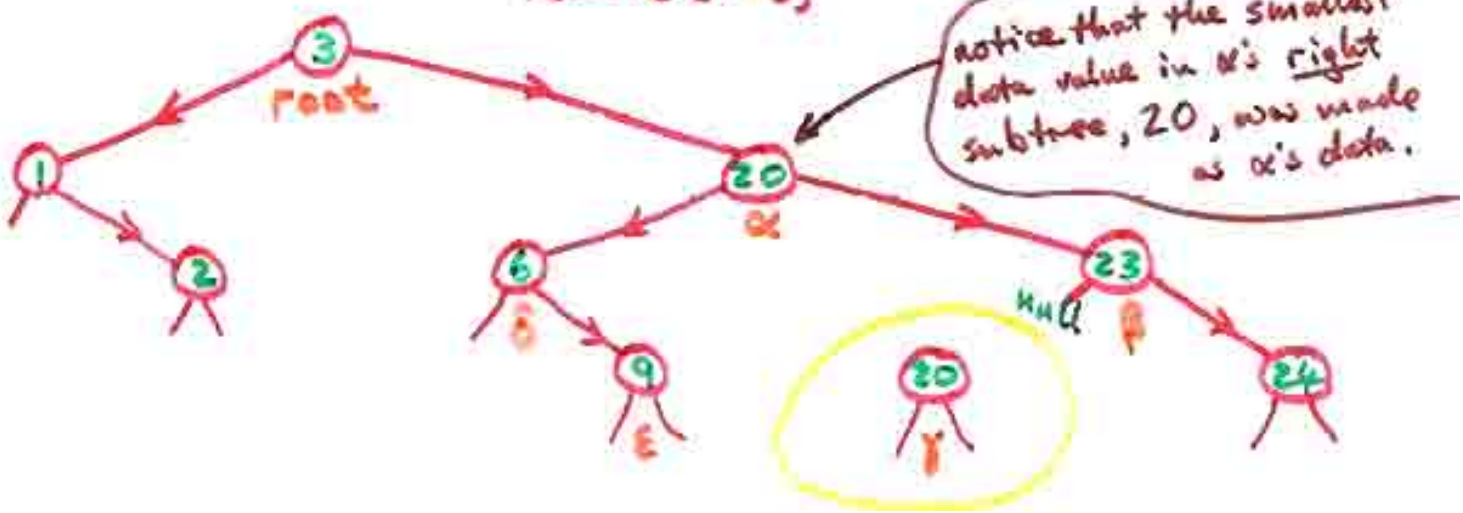
remove(6);



The recursion is similar to the previous case. Notice that the node δ has not been formally deleted (although 'garbage collection' will take care of this), but that α's 'left' (which previously was directed to δ) is now directed to ε. This means of course that δ is no longer being referenced, hence is subject to 'garbage collection'.

Finally, removing a node which has 2 children is a little more delicate, although the recursion is still fairly easy to follow (even though there are now 2 recursions going on!). This code could be made more efficient, but let's remove 15 for an illustration...

remove(15);

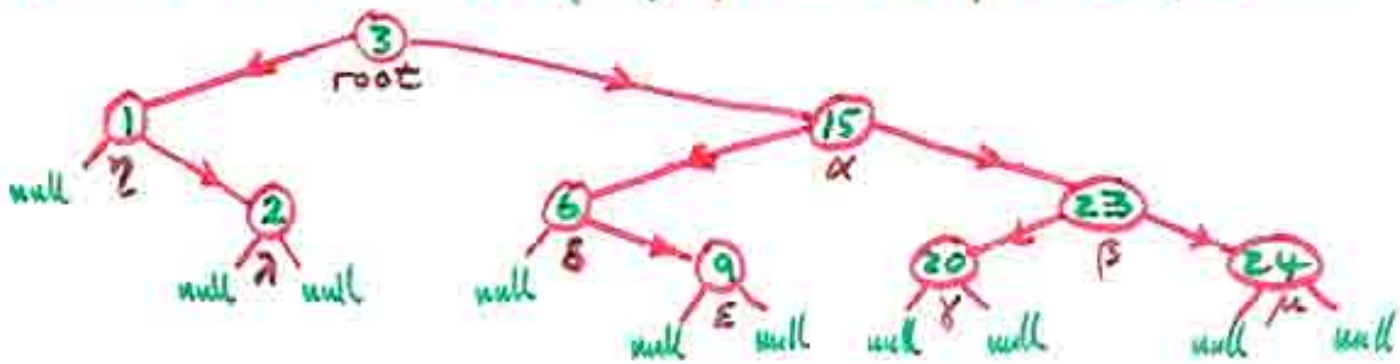


The last method we need to implement is ...

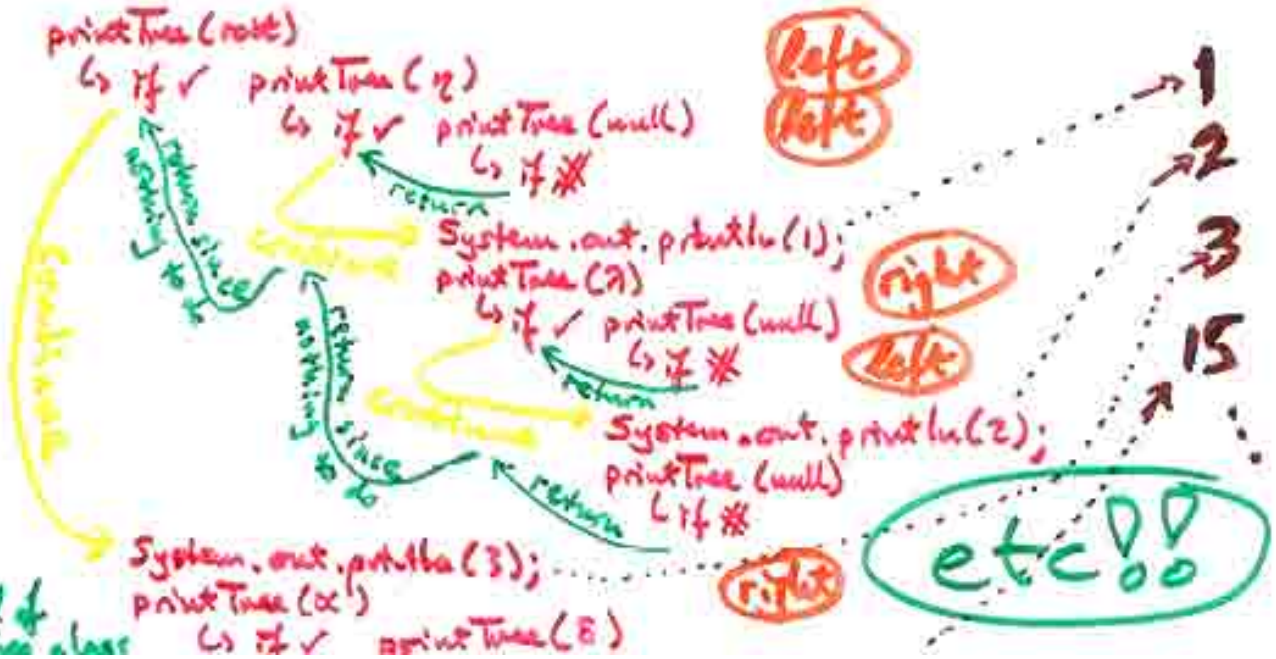
```
private void printTree(IBNode a)
    {
        if (a != null)
            {  printTree(a.left);
               System.out.println(a.data);
               printTree(a.right);
            }
    }

public void printTree()
    {
        if ( isEmpty() ) System.out.println("Empty");
        else printTree(root);
    }
```

This will print data from the tree in sorted order

With our standard example, printTree() produces from ...



the recursion ...



} // end of IBSTree class

Hash Tables

Read chapter 5 in Weiss

As a change from the structures we've been looking at so far, hash tables are almost refreshingly simple.

Think of a collection of labelled buckets, and as each fresh object comes along, we apply some clever hash function to the object which yields the label of the bucket we should put it in. Then putting stuff into this structure is easy, and finding an object merely involves looking in the correctly labelled bucket.

We assume that each Object has an associated key, so that if object e has key k, then e is stored in position $f(k)$ of the hash table (possibly merely an array). To find e, look in position $f(k)$.

As examples:

1. Store students in chairs, $f(k)$ = DNA code of student — lots of chairs!
2. Store students in chairs, $f(k)$ = day/month of birthday — 367 chairs & some collisions.
3. Store students in chairs, $f(k)$ = weight to nearest pound — lots of collisions!

Clearly we don't want to have lots of wasted storage, but we also want to spread the distribution of objects as evenly as possible, and may also have to resolve collisions.

Suppose we are working with a table which can hold M objects (treat this as an array), so the available results from applying the hash function f to keys associated with these objects is an integer in ...

$$0, 1, 2, \ldots, M-1$$

In an ideal sense, each answer would appear equally often as f is applied to all of our possible objects — that way, no one bucket is unfairly overloaded.

Example 1. Let the keys be doubles with $0 \le k < 1$, then define

$$f(k) := \text{Math.floor}(M * k).$$

If keys are doubles with $a \le k < b$, then define

$$f(k) := \text{Math.floor}(M * k / (b - a)).$$

Example 2. Let the keys be ints, then define

$$f(k) := k \,\%\, M.$$

Example 3. Let the keys be Strings, then convert to an array of chars, rewrite each char as an ASCII int, then assemble into a big integer and apply example 2. E.g.,

cat $\longmapsto$ | 99 | 97 | 116 | $\longmapsto$ $99(128)^2 + 97(128) + 116$

$\longmapsto$ $1634548 \,\%\, M$

A little thought shows that using the modular hash function of examples 2 or 3 with M having lots of factors will lead to a very uneven distribution over the table. Far better here is to choose M near the desired size where M is actually prime. (Rather than constructing primes on the fly, have some useful ones stored in a convenient array.

Actually, the String hash function of example 3 could easily get really nasty since a longer string would easily waste space before we even got to the %M part of the computation. Far better would be to continually reduce the running total mod M, for example ...

```
int hash (String s, int M)
  {                                   ← sneaky prime trick
    int h = 0, a = 127;
    for (int i=0; i<s.length(); i++)
      {
        int temp = Character.getNumericValue(charAt(i));
        h = (a*h + temp) % M;
      }
    return h;
  }
```

Given that we might have a decent hash function, we must still resolve the issue of collisions. There are many approaches to this, but the easiest (and the one of primary concern to us) is that of separate chaining.

This is a funny name for doing what is really the most obvious thing ...

if a collision occurs, build a linked list at the site.

As an example, consider the collection ...

Object	A	B	C	D	. . .	X	Y	Z
Key	0	1	2	3	. . .	23	24	25

upper and lower case identically keyed.

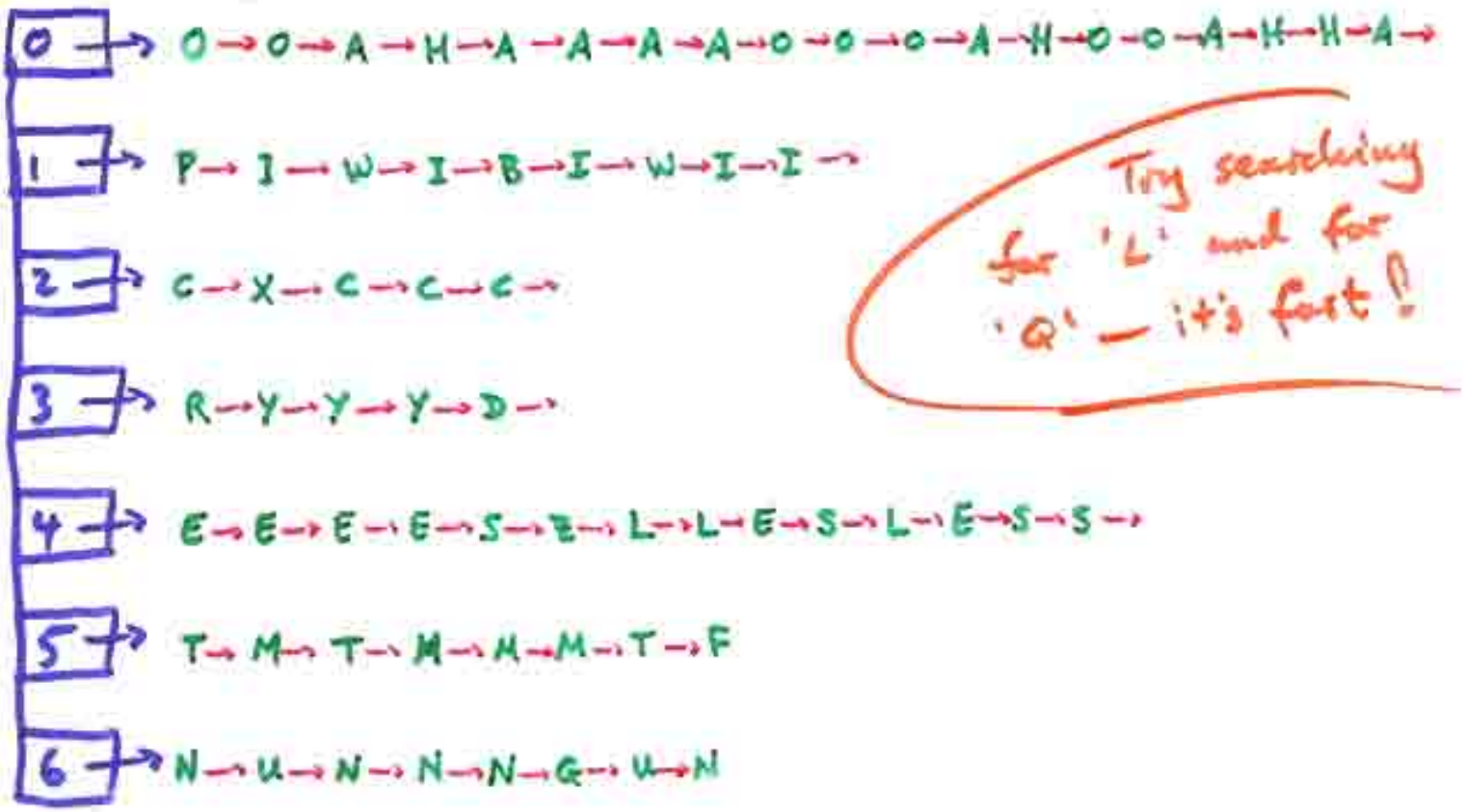
with hash function ...

$$f(k) = k \% 7$$

applied to the text (space-less for illustration) ...

Once upon a time there was an amazingly yummy box of Leonidas chocolates which as if

With a table of 7 'buckets', this produces ...



Try searching for 'L' and for 'Q' — it's fast!