Data Structures

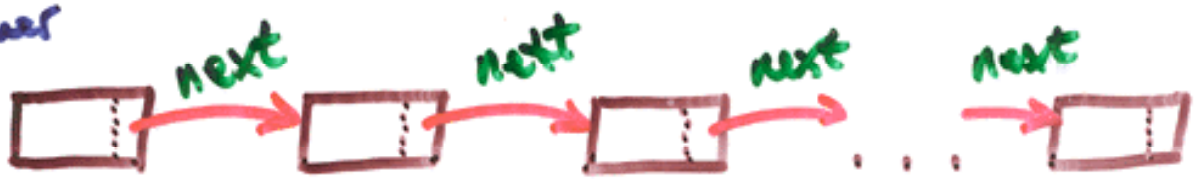
Read Chapter 3 of Weiss

Firstly, an overview.

1. Lists

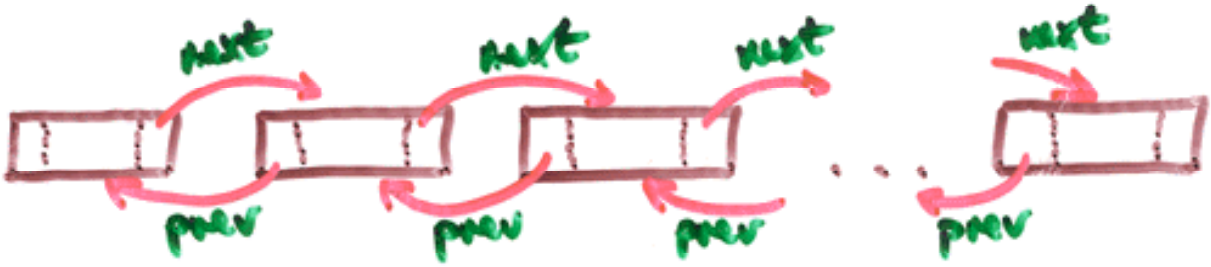
A list is a collection of things in order. We refer to a linked list where the 'link' points to the next object in the list.

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;
    boolean find(Object x)  throws ItemNotFound;
    void remove(Object x)   throws ItemNotFound;
    Object retrieve()       throws ItemNotFound;
    boolean isInList();
    void zeroth();
    void first();
    void advance();
  }
```

in case we're in a bad place

return item in current position

set current position prior to first!

set current position to first

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; }
    various constructors ...
  }
```

caveats about current being null should be here!

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 ...

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

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

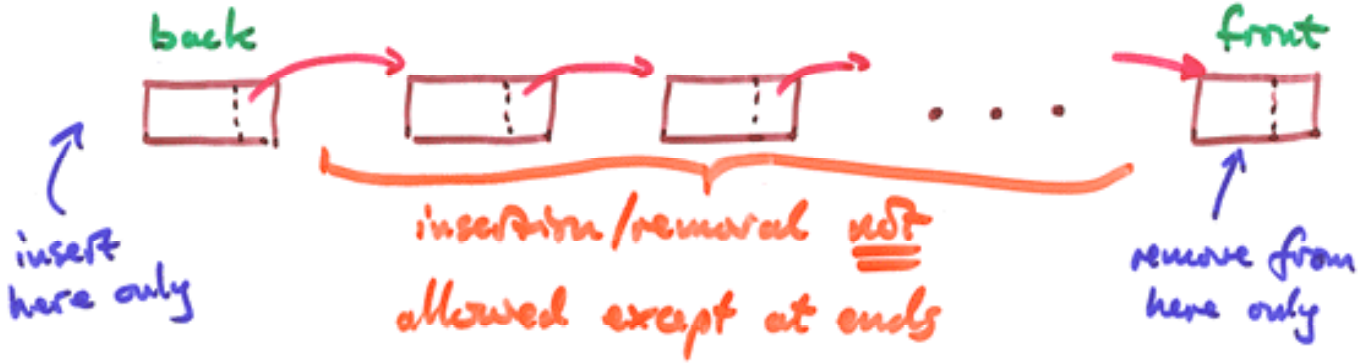
The obvious extra steps would be needed if working with a two-way linked list.

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

Three close relatives of linked lists are ...

2. Queues

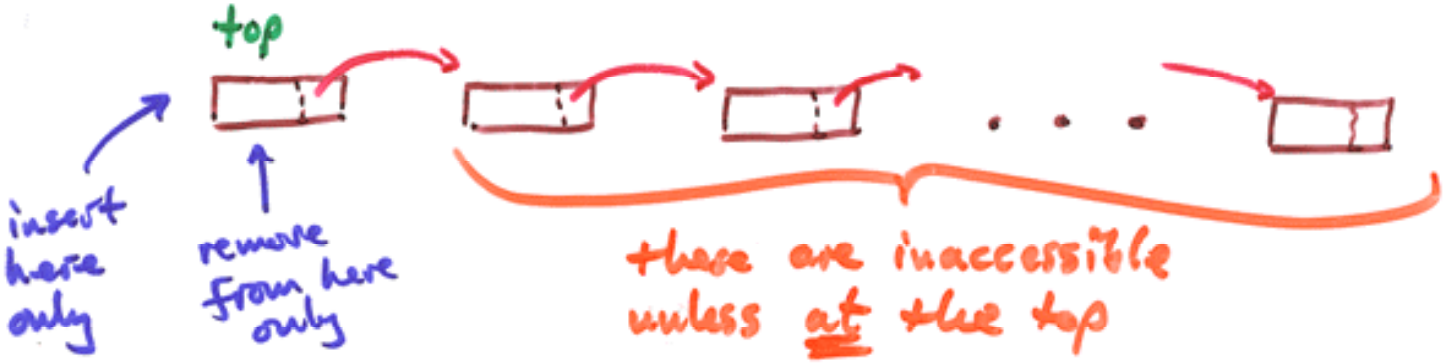
In a natural sense, a queue is an emaciated list which allows insertion at one end and removal from the other ...



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

3. Stacks

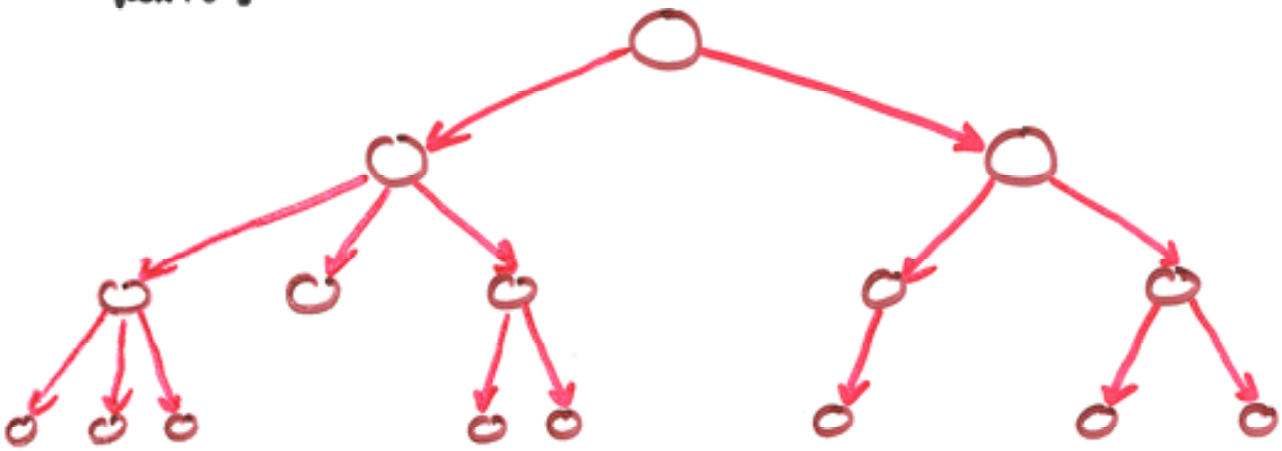
In a natural sense, a stack is a really emaciated list which only allows insertion and removal from one and the same end!



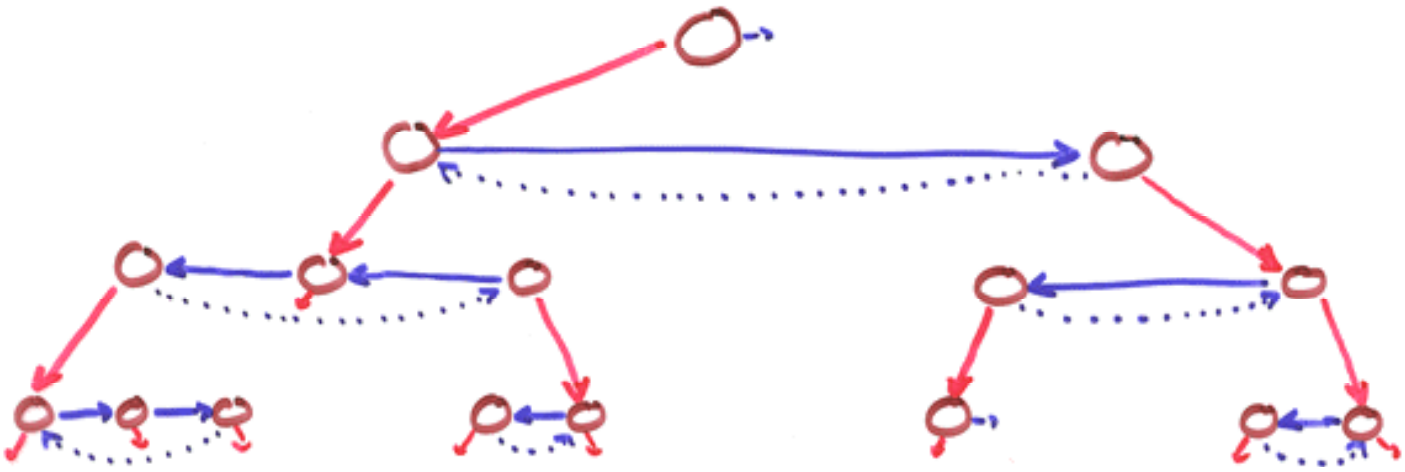
Stacks are used heavily in many areas of program compilation, use and design.

41. Trees

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

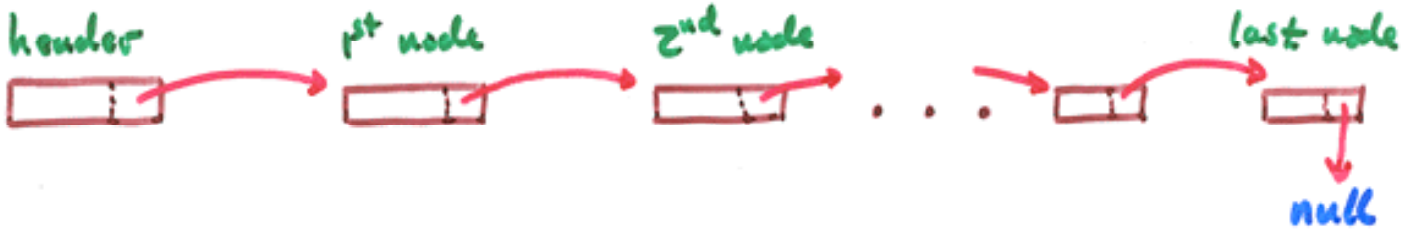


Although this approach can work well for binary trees, in the general case illustrated above it's often better not to think of each parent node having multiple children, but rather of each parent node having a favourite child and a favourite sibling (to the parent).



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 ListItr interfaces, however there is a philosophical and practical distinction between an actual "list" and the processes which move us around it 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, ListItr p ) throws
                                    ItemNotFound;
    ListItr find( Object x)
    void remove( Object x) throws ItemNotFound;

    ListItr zeroth();
    ListItr first();
  }
```

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

now 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 starting with a fresh list...

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

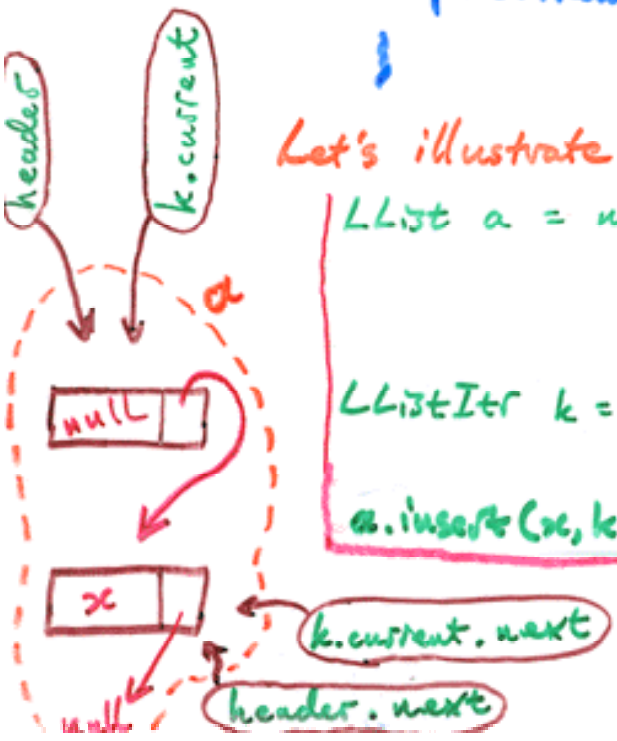

```
LListItr k = new LListItr(a.header);
```

```
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); }
```

So zeroth().current == 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;
  }
```

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

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 always!

```
public void push (Object x)
   { 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; }
}
```

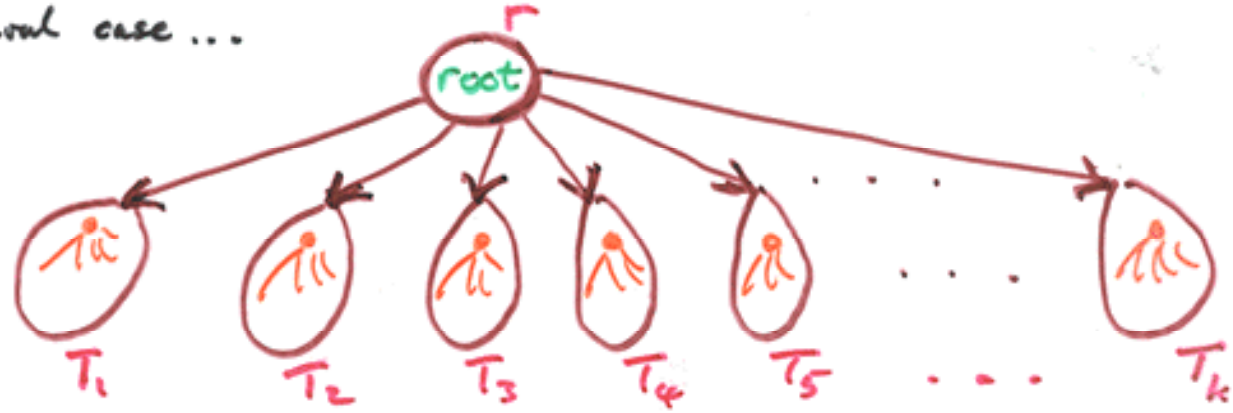
remember this is a Stack

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).

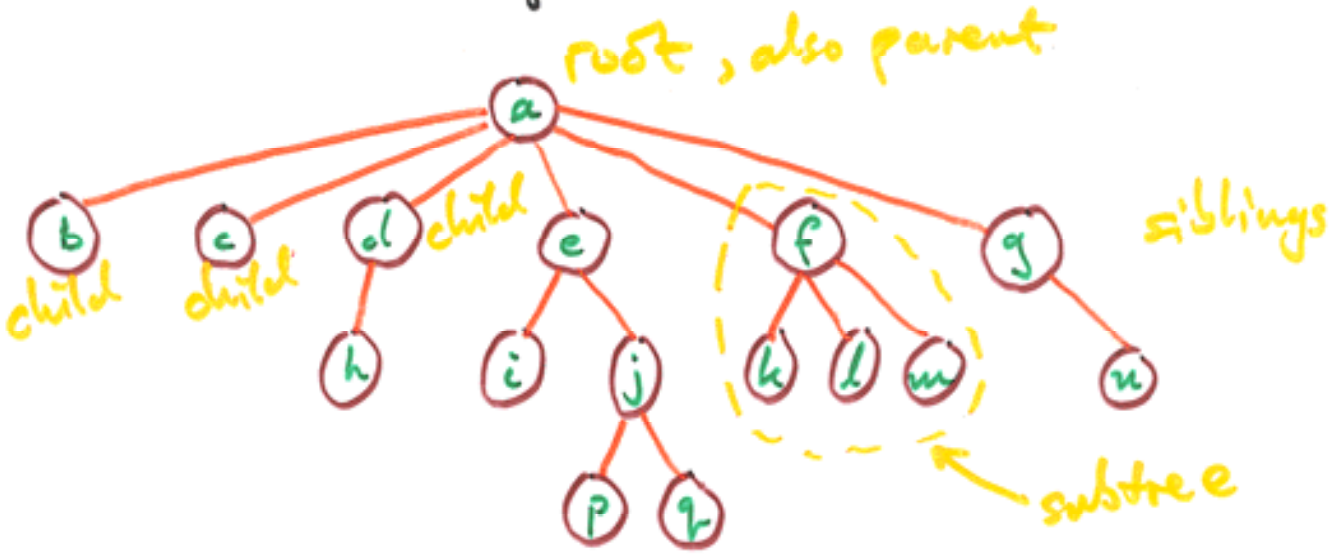
Trees

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 interfaces

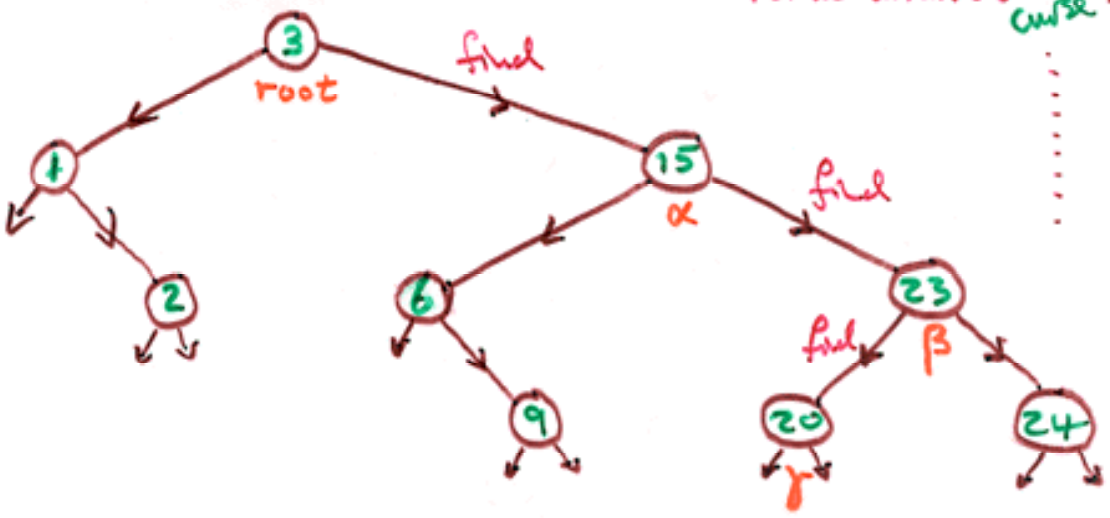
recursive flavour

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
returns γ
if ✱
if ✓ return find (20, γ) nothing returned yet
↳ if ✱ if ✱ if ✱
else return γ;
returns γ
now we return γ

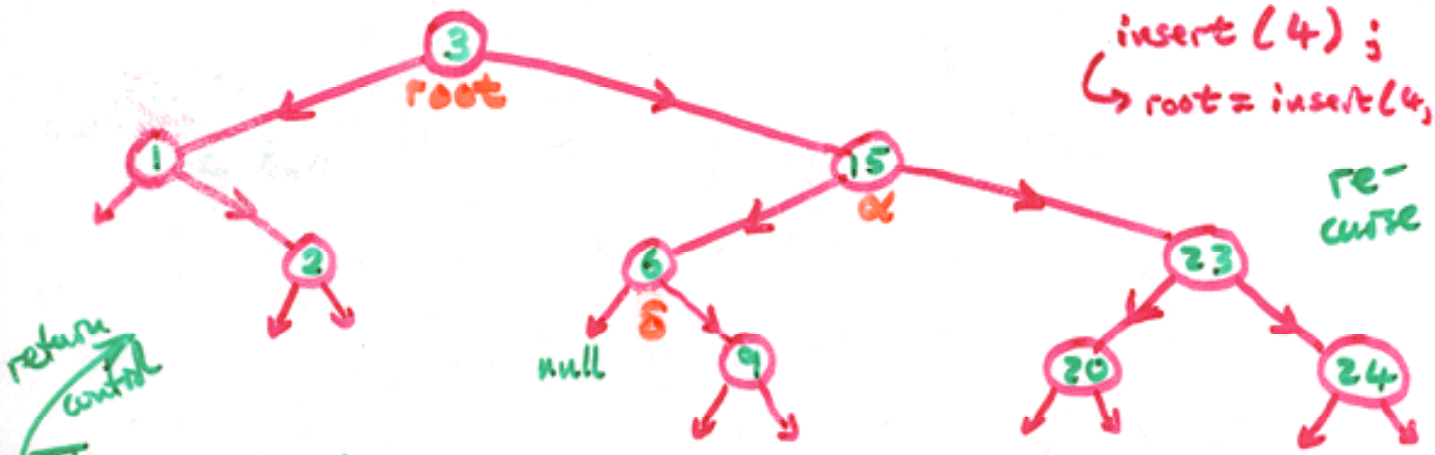
follow the arrow directions carefully — no 'returning' happens until we're at the bottom, and it's the bottom (successful) node which is returned all the way.

```
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 in the conditionals? If these

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 ...



insert(4);
↳ root = insert(4, root);

re-curse

return control

The recursion here is ...

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

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

↳ if ✱
if ✓ α.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;  // ain't 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 →

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' 'spurs' are really null, not just the one marked.

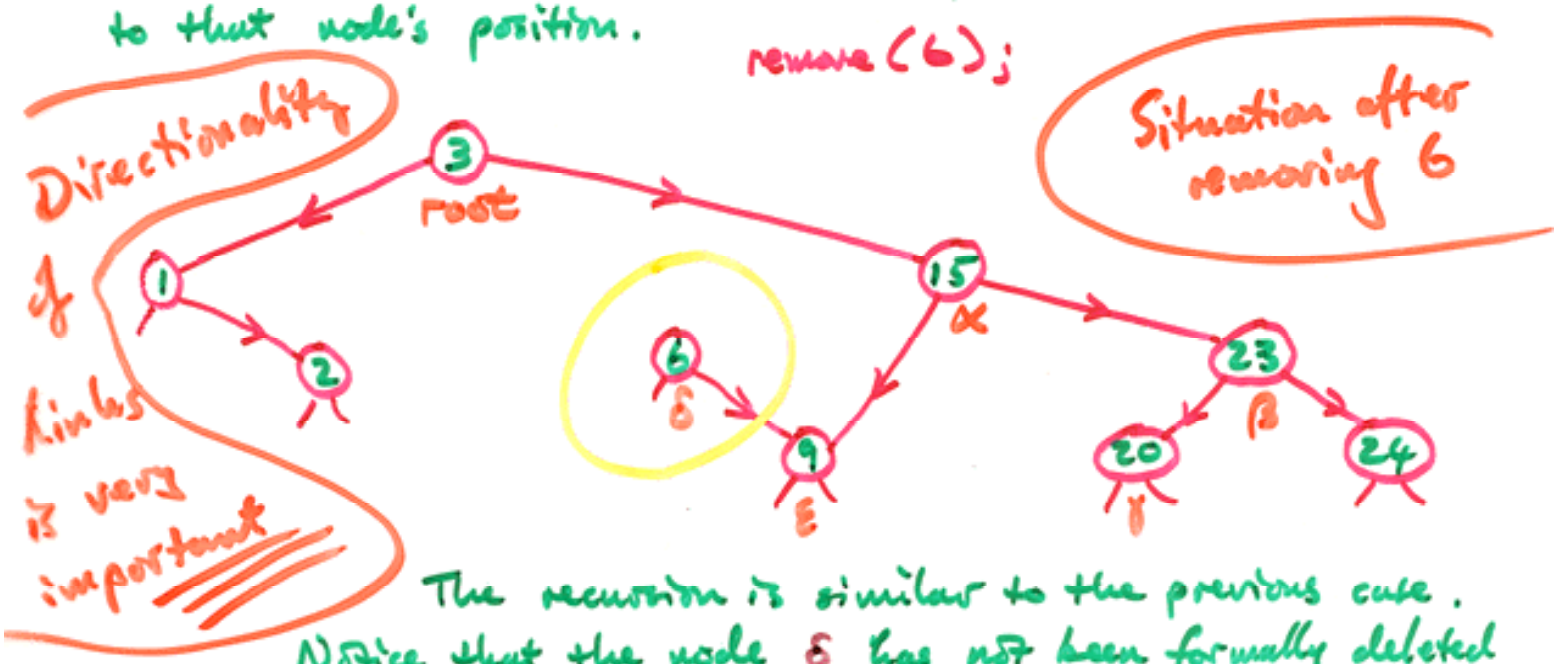
B's left becomes null

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

remove (20) → root = remove (20, root) → if * if * if ✓
all these return control with no change → 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.

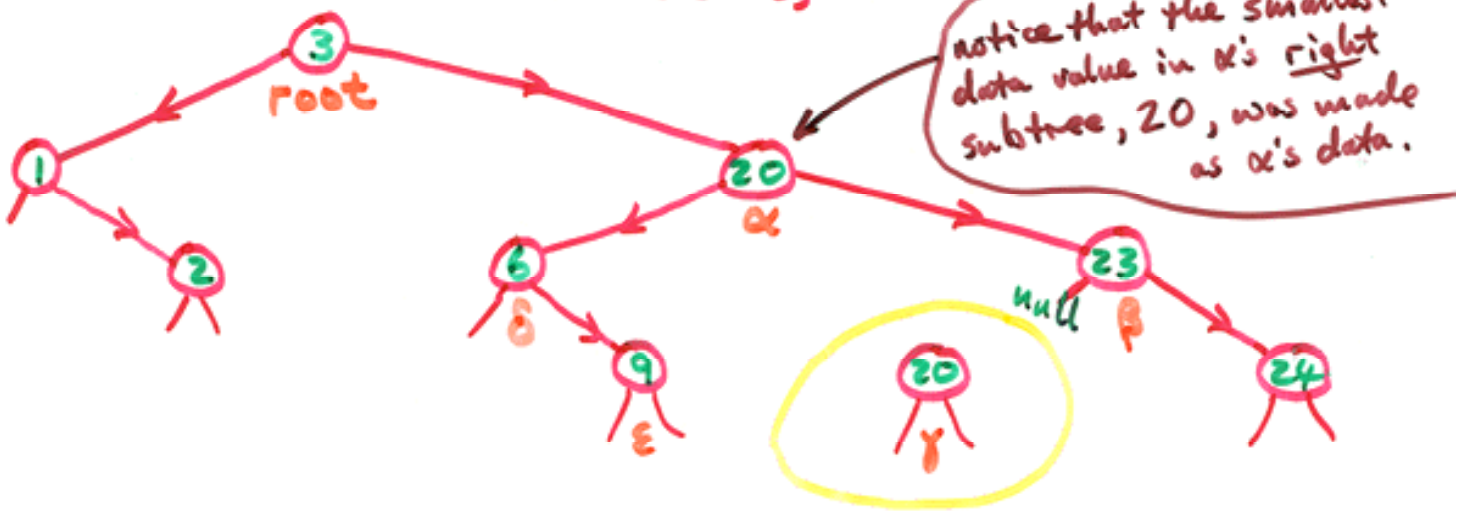
remove(6);



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);



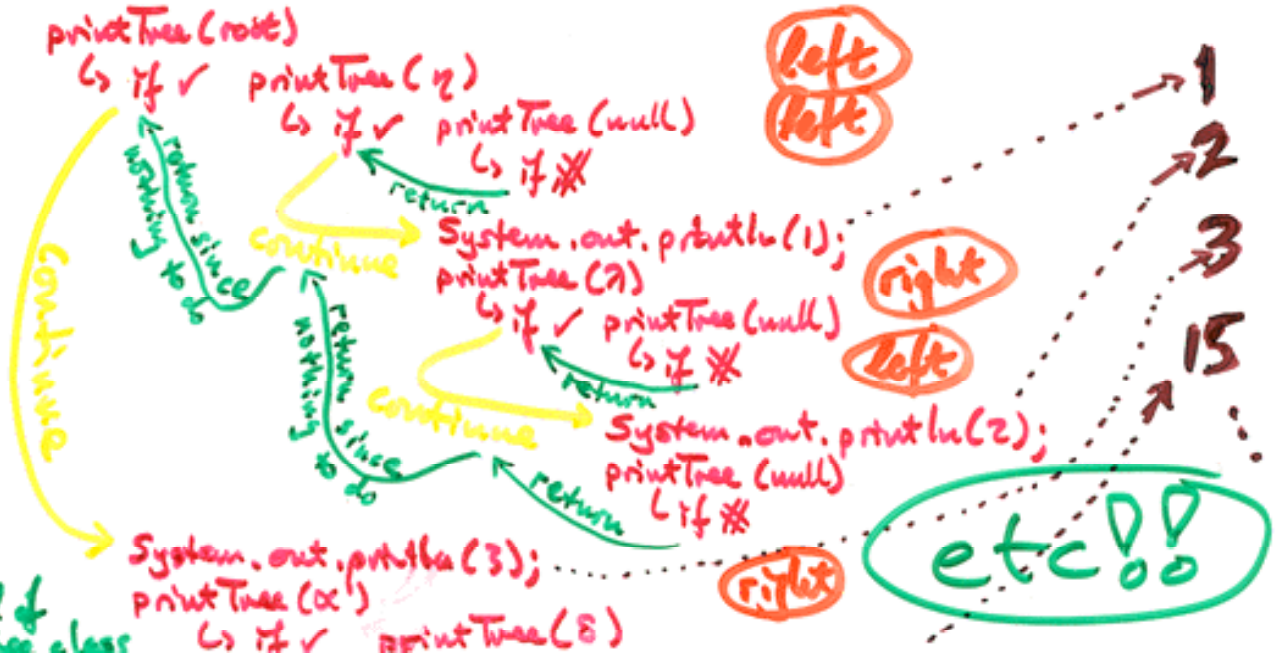
The last method we need to implement is ...

```
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 ⟼ | 99 | 97 | 116 | ⟼ $99(128)^2 + 97(128) + 116$

⟼ $1634548 \,\%\, M$

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)
  {
    int h = 0, a = 127;          <- sneaky prime trick
    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 fancy 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.

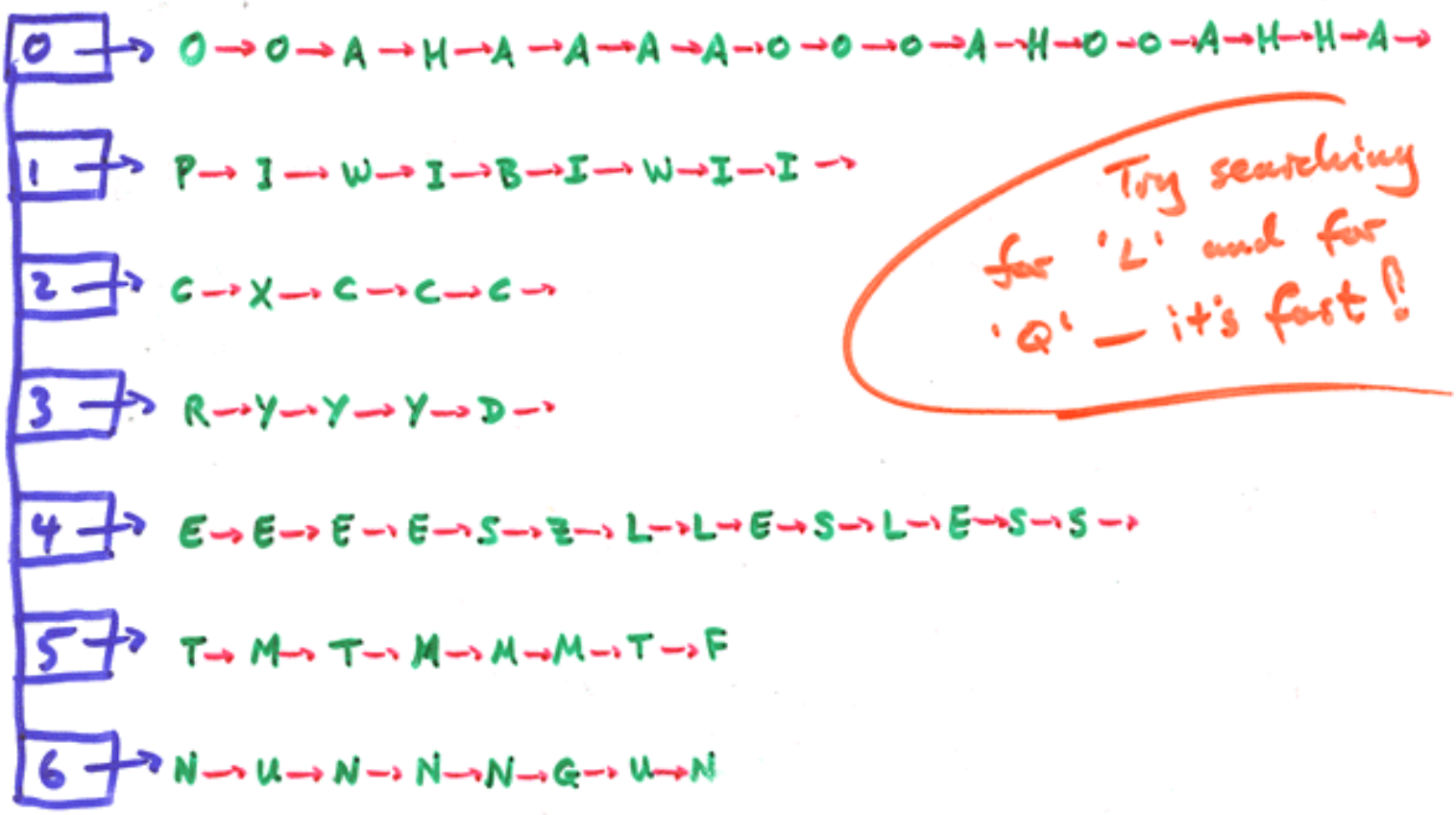
with hash function ...

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

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 ...

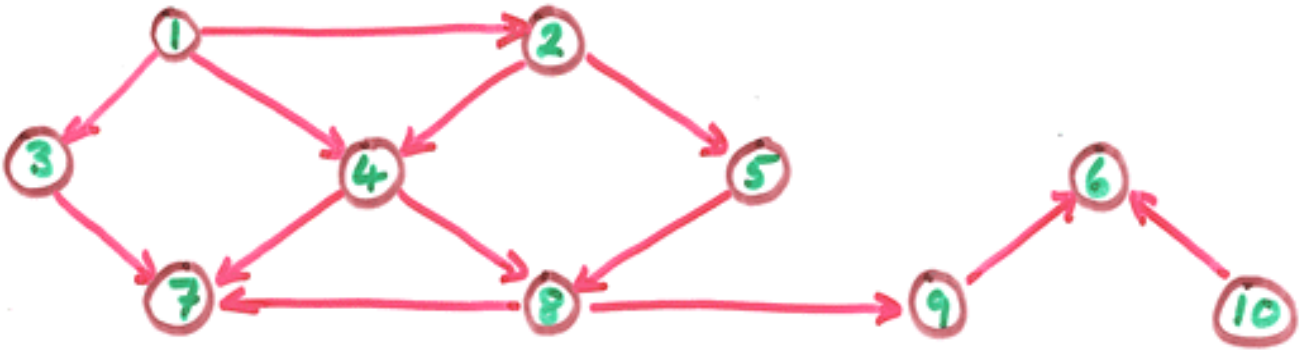


Try searching for 'L' and for 'Q' — it's fast!

Graphs

Read chapters 8 & 9 of Weiss

The trees we've been looking at are special cases of a more general design — graphs. These are general arrangements of nodes and edges ...



The edges will always be directed, some people call these graphs digraphs. Edges can carry weights or costs, a path in a graph is a sequence of vertices $v_1, v_2, \ldots, v_N$ such that (v_i, v_{i+1}) is an edge (with the correct direction), the length of a path is the number of edges on the path ($N-1$ in our example path), and the cost of a path is the sum of the weights along each edge of the path. A simple path never repeats vertices (except possibly the last may equal the first — called a loop or cycle if the length ≥ 1). As a formal statement, we allow paths of length 0, namely a path from a vertex to itself using no edges. An acyclic graph has no cycles.

A graph is complete if there is an edge between each pair of vertices. If a graph has paths from each vertex to each other vertex, then it is strongly connected; if it's not strongly connected, but the underlying undirected graph is connected, then the graph is weakly connected.

We could represent our graph by a two-dimensional adjacency matrix, where the weight of an edge from vertex k to vertex l is in position (k, l) of the matrix...

This time, assume each edge has weight 1, and we use . to represent the value ∞, for clarity. We use 0 weight to show absent connection.

Notice how wasteful such a sparsely connected graph is in matrix form, both for space and for the initialization cost — essentially $O(n^2)$ where n is the number of vertices. A better arrangement is to build lists of adjacent vertices for each vertex; the nodes in these lists could hold both the vertex 'name' and the edge weight ...

Notice that in this adjacency list, the space requirement is now just O(e), where e is the # edges. Initialization is now O(n), and building the graph is O(e).

Given an acyclic graph, we can write a simple routine to perform a topological sort. This is a non-unique list of vertices such that if there is a path from vertex v to vertex w, then w appears after v in the sorted list. (The reason for disallowing cycles is obvious!!) For example, a topological sort of our example graph could produce ...

1, 2, 4, 5, 8, 3, 7, 9, 10, 6

To do this in a more organized fashion, find any vertex having no incoming edges. Print and remove this vertex (together with its outgoing edges). Repeat until finished. Define the indegree of a vertex v as the # edges (u,v) coming in to v, and now assume that a graph has been read into an adjacency list where each vertex also stores its indegree ...

```
class Vertex
  {
    String name;
    LList adj;          // list of adjacent vertices
    int dist;           // for cost
    Vertex path;        // for previous vertex on shortest path
    boolean known;      // for possible later use
    int indegree; int topNum;

    public Vertex ( String appel)
       { name = appel; adj = new LList (); reset (); }
    public void reset ()
       { dist = Graph.INFINITY; path = null; }
  }
```

Integer.MAX_VALUE

Then our topological sort routine might be ...

```
void topSort() throws CycleFound
  {
    Vertex v, w;

    for (int i = 0; i < NUM_VERTICES; i++)
      {
        v = findZero();
        if (v == null)
          throw new CycleFound();
        v.topNum = i;
        adjustGraph();
      }
  }
```

a method which for each vertex w adjacent to v (i.e., one step out along an edge) performs w.indegree--;

a method to find any vertex of indegree 0 in the current version of the graph which has not yet had a topological number assigned to it.

This is a little wasteful, since findZero() is clearly O(n) and is applied n times, so our algorithm is $O(n^2)$. If the graph is sparse, then only a few vertices will have their indegrees updated in a given iteration. Better would be to store those (unassigned) vertices of indegree 0 separately, and whenever a vertex's indegree becomes 0, that vertex is stored there. We'll use a queue for this storage. This change now makes our algorithm $O(n + e)$.

```
void topSort() throws CycleFound   // O(n+e) version
  {
    Queue q;
    int i = 0;
    Vertex v, w;

    q = new Queue();
```

```
queueZero();
while (!q.isEmpty())
  {
    v = q.dequeue();
    v.topNum = ++i;
    adjGraph();
  }
if (i != NUM_VERTICES)
  throw new CycleFound();
}
```

a method which for each vertex v performs
if (v.indegree == 0)
q.enqueue(v);

a method which for each w adjacent to v performs
if (--w.indegree == 0)
q.enqueue(w);

A second question often raised in the context of graphs is finding the shortest path between two vertices in a graph. This is usually approached in two flavours; where the edges are unweighted, and then where various weights/costs are assigned to the various edges.

Taking our example graph again, we could ask for the lengths of the shortest paths from any given vertex s to every other vertex of the graph. If there is no path to a given vertex in the directed graph, then that 'path' has length 'infinity'. For example, if s is vertex #2, we can find all vertices of shortest path length 0 from 2, then all those of length 1 from these (path length 1), then all those of length 1 from these (path length 2), et similia.

however, it's better to use queues as we did for the topological sort to give an $O(n+e)$ algorithm. (We will assume that dist for each vertex has been reset to the value INFINITY before we start...)

```
void unweighted ( Vertex s )
  {
    Queue q;
    Vertex v, w;

    q = new Queue();
    q.enqueue(s);
    s.dist = 0;

    while (! q.isEmpty())
      {
        v = q.dequeue();
        for (each w adjacent to v)
          if (w.dist == INFINITY)
            {
              w.dist = v.dist + 1;
              w.path = v;
              q.enqueue(w);
            }
      }
  }
```

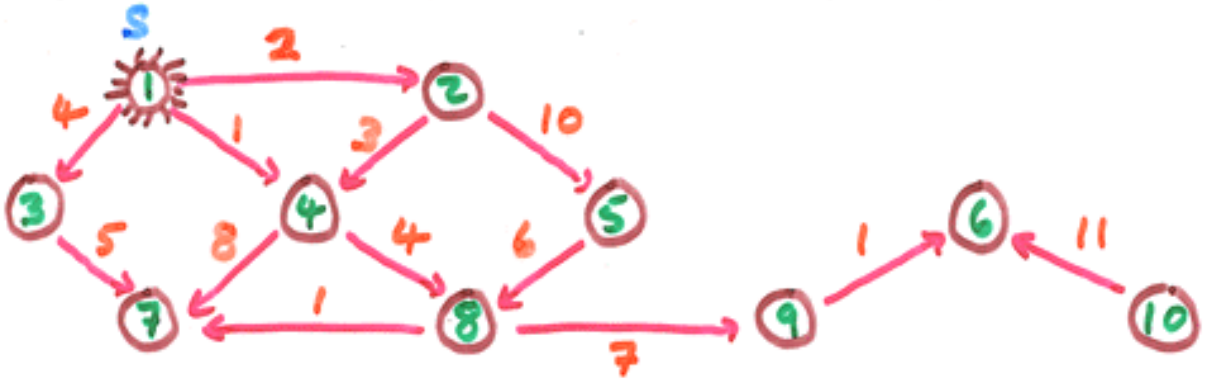
on first pass this sets v = s. Subsequent passes pull previously handled vertices from the queue to deal with their adjacent vertices...

this increments the w.dist for w adjacent to v provided w.dist hasn't previously been given a reasonable value

Now we attack the weighted edge flavour (initially assuming for sanity that all edge weights are non-negative).

The process we're going to describe is known as Dijkstra's algorithm; it's a particular example of a class of techniques called greedy algorithms which essentially proceed at each stage with what appears to be the best 'local' solution.

At each step we choose a vertex v having the smallest dist amongst the unknown vertices, and then set as known the shortest path from s to v. The various values of dist are then updated. Taking our standard example, and adding weights to the edges, and starting at vertex 1...



A table of vertex values changes as follows...

vertex:	1	2	3	4	5	6	7	8	9	10	
known:	F	F	F	F	F	F	F	F	F	F	initially
dist:	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	initially
path:	0	0	0	0	0	0	0	0	0	0	initially
known:	T	F	F	F	F	F	F	F	F	F	vertex 1 now known, so adjust adjacents
dist:	0	2	4	1	∞	∞	∞	∞	∞	∞	vertex 1 now known, so adjust adjacents
path:	0	1	1	1	0	0	0	0	0	0	vertex 1 now known, so adjust adjacents
known:	T	F	F	T	F	F	F	F	F	F	vertex 4 now known, so adjust adjacents summary dists
dist:	0	2	4	1	∞	∞	9	5	∞	∞	vertex 4 now known, so adjust adjacents summary dists
path:	0	1	1	1	0	0	4	4	0	0	vertex 4 now known, so adjust adjacents summary dists
known:	T	T	F	T	F	F	F	F	F	F	vertex 2 now known
dist:	0	2	4	1	12	∞	9	5	∞	∞	vertex 2 now known
path:	0	1	1	1	2	0	4	4	0	0	vertex 2 now known
known:	T	T	T	T	F	F	F	F	F	F	vertex 3 now known
dist:	0	2	4	1	12	∞	9	5	∞	∞	vertex 3 now known
path:	0	1	1	1	2	0	4	4	0	0	vertex 3 now known

known:	T	T	T	T	F	F	F	T	F	F	vertex 8
dist:	0	2	4	1	12	∞	6	5	12	∞	now known
path:	0	1	1	1	2	0	8	4	8	0	
known:	T	T	T	T	F	F	T	T	F	F	vertex 7
dist:	0	2	4	1	12	∞	6	5	12	∞	now known
path:	0	1	1	1	2	0	8	4	8	0	
known:	T	T	T	T	T	F	T	T	F	F	vertex 5
dist:	0	2	4	1	12	∞	6	5	12	∞	now known
path:	0	1	1	1	2	0	8	4	8	0	
known:	T	T	T	T	T	F	T	T	T	F	vertex 9
dist:	0	2	4	1	12	13	6	5	12	∞	now known
path:	0	1	1	1	2	9	8	4	8	0	
known:	T	T	T	T	T	T	T	T	T	*F	vertex 6
dist:	0	2	4	1	12	13	6	5	12	∞	now known
path:	0	1	1	1	2	9	8	4	8	0	

* this will then change to T at end

The code might then be ...

```
public Vertex [] createTable()
  {
     Vertex [] t = readGraph();
     for (int i=0; i< t.length; i++)
       {
          t[i].known = false;
          t[i].dist  = Graph.INFINITY;
          t[i].path  = null;
       }
     NUM_VERTICES = t.length;
  }
void printPath( Vertex v)
  { if (v.path != null)
     { printPath(v.path);
       System.out.print(" to ");
     }
    System.out.print(v.name);
  }
```

some method to grab the graph with its adjacency lists, etc..

we started this by 0 in the tables above + the example.

recursive path printing

```
void dijkstra ( Vertex s )
  {
     Vertex v, w;
     s.dist = 0;

     for ( ; ; )
       {
          v = unknown vertex with smallest dist
          if (v == null)  break;
          v.known = true;

          for each w adjacent to v
             if ( ! w.known )
                if ( v.dist + edge wt. from v to w < w.dist )
                   {
                      w.dist = v.dist + edge wt.;
                      w.path = v;
                   }
       }
  }
```

This is an $O(e + n^2)$ algorithm

There are many improvements we could make, especially in the case of a very sparse graph, but we save these embellishments for later courses.

Dijkstra's algorithm fails if we were to allow negative edge weights, since there could be 'cheaper' paths appearing later going back to previously marked 'known' vertices.

An $O(en)$ algorithm (!!) to deal with this could be ...

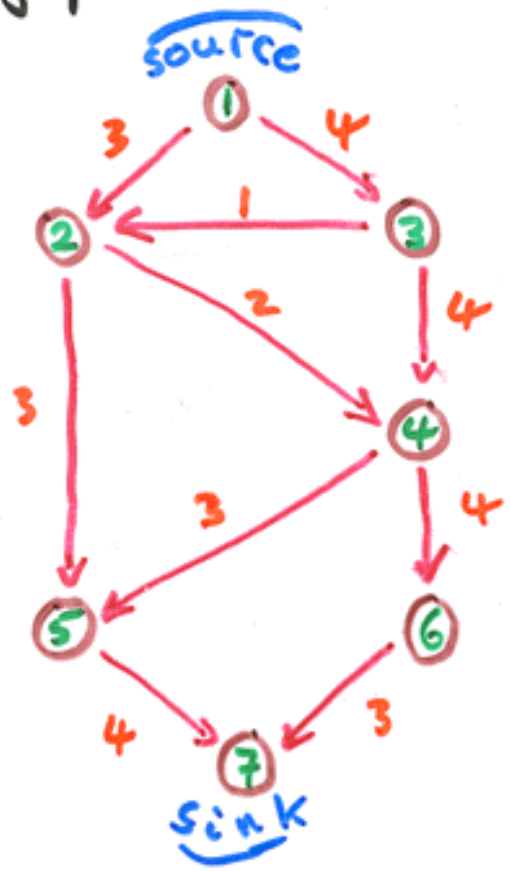
```
void negwts ( Vertex s )
  {
     Queue q;
     Vertex v, w;
     q = new Queue();
     q.enqueue(s);

     while (! q.isEmpty())
       {
          v = q.dequeue();
          for each w adjacent to v
            if (v.dist + edge cost from v to w < w.dist)
              {
                 w.dist = v.dist + edge cost;
                 w.path = v;
                 if (w is not already in q)
                   q.enqueue(w);
              }
       }
  }
```

This code doesn't quite work correctly - see comments following...

The above code is an amalgam of our unweighted and Dijkstra solutions. The idea is to start by putting s on a queue, then for each dequeued v we find all adjacent w with current dist larger than v.dist + edge cost, update w.dist and w.path, and ensure that w is on the queue. We could use the now unused known to indicate a vertex's presence on the queue. This is then repeated until the queue is empty. Notice that each vertex should only be able to dequeue at most n times, so to avoid infinite looping if there are negative costs we should ensure that no vertex gets dequeued more than (n+1) times!!

If we know that the graph is acyclic, then we can improve Dijkstra to an $O(e+n)$ algorithm by doing the selecting and updating within essentially a topological sort algorithm (using this order to declare vertices 'known'). This kind of graph appears frequently in practical applications.

Looking again at general weighted edge graphs, but with an almost reversed emphasis, leads to their use in analysing network flow problems. Here the edge-values refer to permitted flow rather than assessed cost. We'll use a fresh graph to illustrate this problem ... Whether you think of this as traffic flow, plumbing, or internet connectivity is up to you!



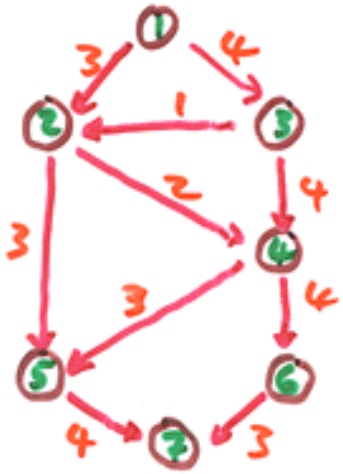
We'll start by considering an intuitively reasonable, but flawed approach, and then adjust it to work. Our assumptions are that at each vertex (apart from the source & sink), "what comes in must go out". Our approach will be to build two related graphs — one to display the flow choices, the other to display the residual flow available ...

flow graph

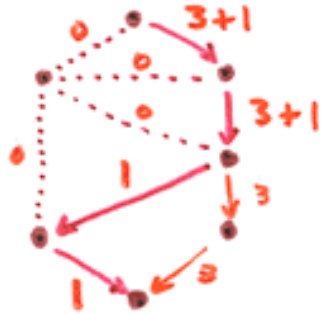
Initial set-up

We start by choosing any path from 1 to 7, labelling this in the flow graph with the maximum flow this whole path can accomodate.

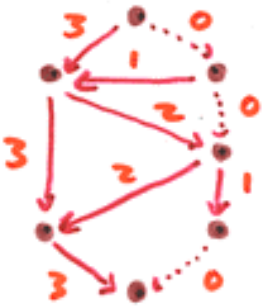
residual graph

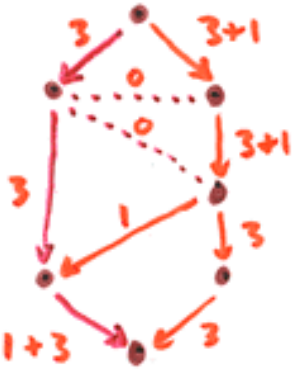


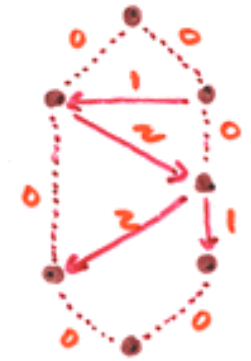
Having now reached this situation, we repeat the process (looking at the residual graph).



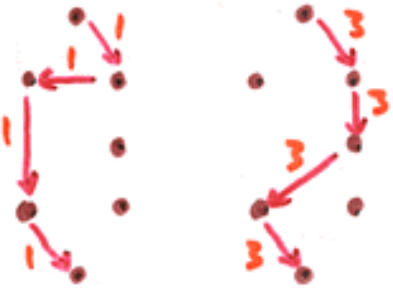
Again we chose our path somewhat at random, and a final flow arrangement can be found easily from here. (Note that for our graph, the solution is not unique.) The process clearly terminates since the residual graph now provides no way of getting from the top to the bottom.







The flow with this algorithm can be seen if our first two choices of paths were these following...

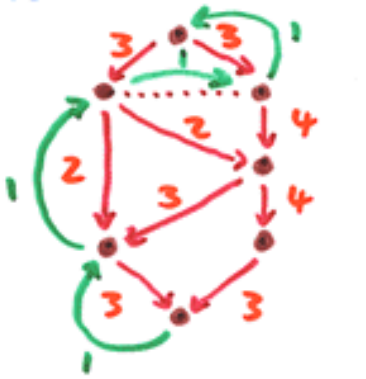


This leaves room 2 to enter right!

There's an easy fix for this — to allow our algorithm to "change its mind", each insertion in the flow graph will be accompanied by an equal and opposite insertion in the residual graph. Hence our previously 'bad' choice would appear as follows ...

flow graph

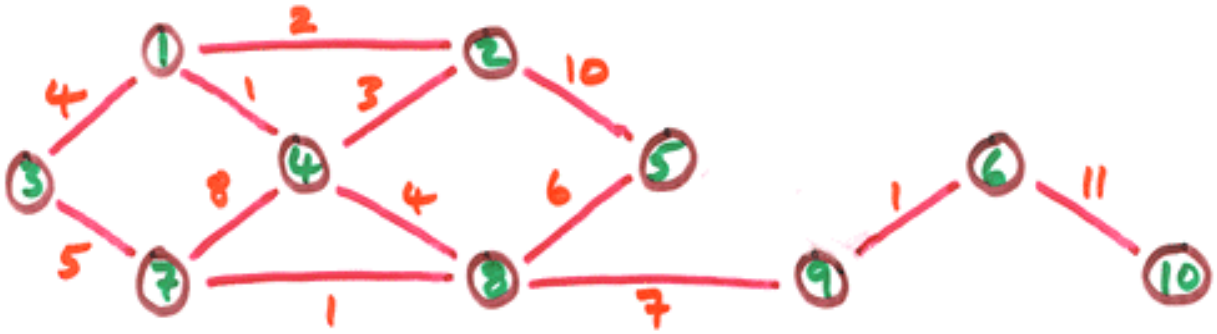
residual graph



We continue in this fashion, always augmenting the total flow. This is not necessarily a particularly efficient algorithm — given integer flows and a final max flow of F, this is an $O(e.F)$ algorithm at worst!!

Another thing we can ask about graphs is to find a minimum spanning tree. Since the solution is easier for undirected graphs, we'll restrict our attention to this situation. What we mean by such an animal is of course that it should be a tree (no circuits), that every vertex of the original graph be a vertex of this tree, and that 'minimal' means that the total edge costs in the tree are as low as possible. There is absolutely no guarantee that such an animal be unique. A few moments thought will convince you that if the graph has n vertices then the minimum spanning tree will have (n-1) edges.

Using our standard example (and removing the directedness) gives ...



with minimum spanning tree ...

having total 'cost' 37.

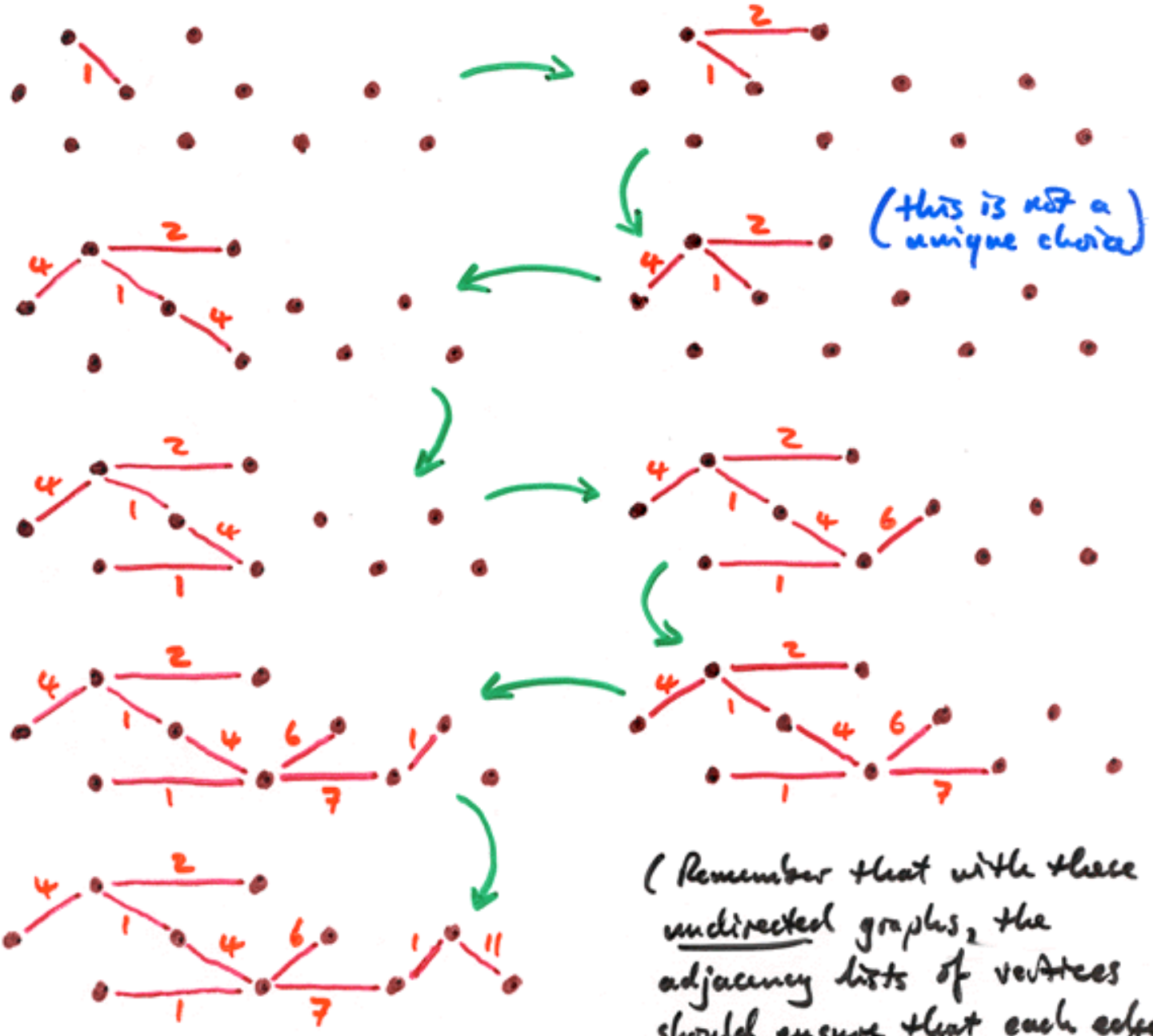
There are two natural approaches to this problem. The first (Prim's algorithm) builds the tree by picking any vertex as root, and then adds the vertex v by finding an edge u—v of minimum cost where u is in the tree and v is not yet in it. This is reminiscent of Dijkstra, so we would maintain a table of 'known' vertices, weights of 'cheapest' edge connecting a given vertex to a known vertex, and the vertex most recently used to change the dist. The starting table would be (if starting at vertex 4) ...

vertex #	1	2	3	4	5	6	7	8	9	10
known:	F	F	F	F	F	F	F	F	F	F
dist:	∞	∞	∞	0	∞	∞	∞	∞	∞	∞
path:	0	0	0	0	0	0	0	0	0	0

The update rule for this is simply that for each vertex v chosen, we reassign...

$$dist_w = \min(dist_w, cost_{v,w})$$

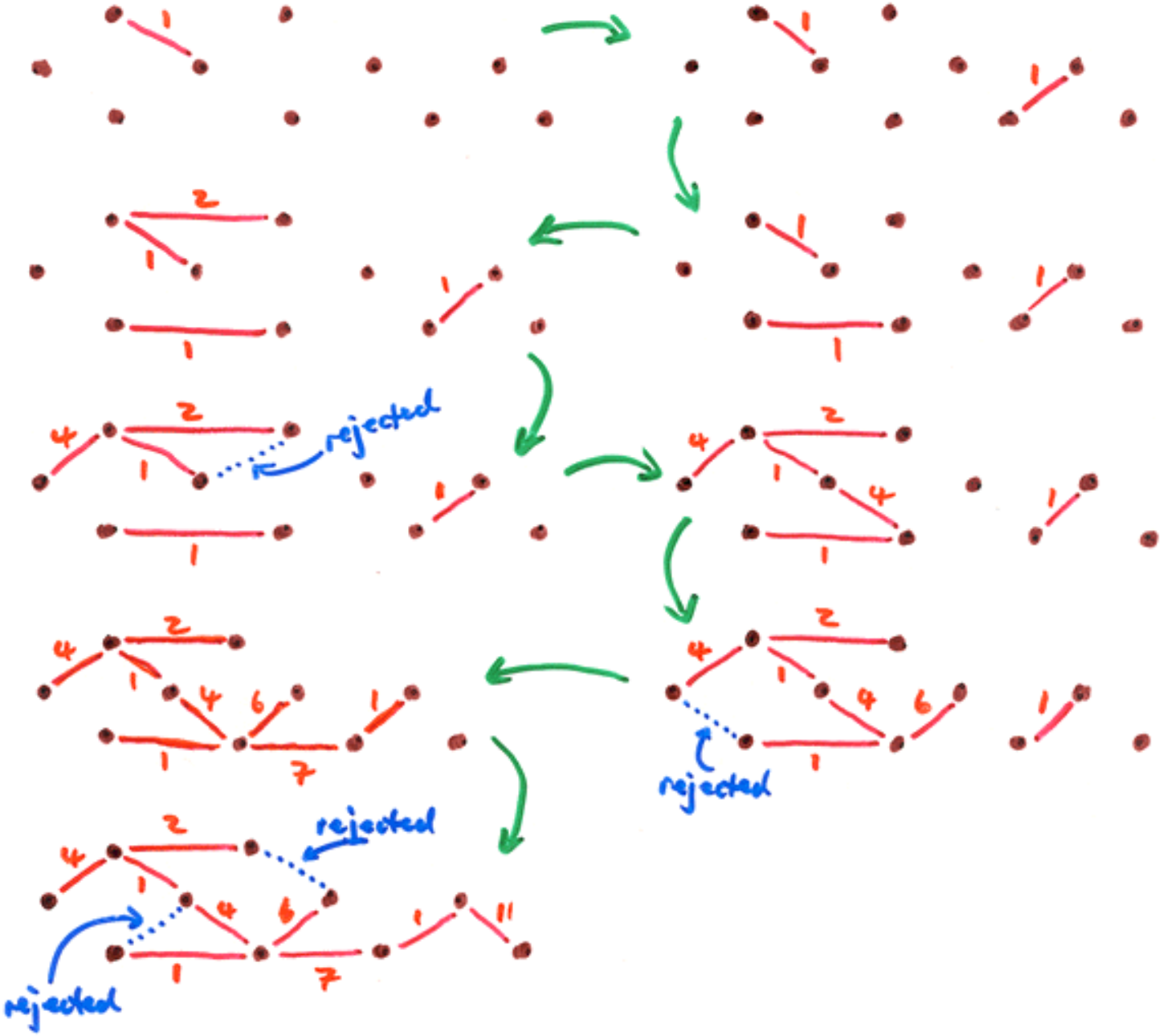
for each vertex w adjacent to v. Graphically, starting with vertex 4, this gives the following sequence of pictures...



(Remember that with these undirected graphs, the adjacency lists of vertices should ensure that each edge is in two lists!)

The running time of Prim is $O(n^2)$ — this is optimal for dense graphs. If the graph is sparse, then binary heaps should be used here, giving Prim a running time of $O(e \log n)$.

The other natural approach (Kruskal's algorithm) gathers edges in order of 'cheapness', rejecting any edges which don't add any new vertices to the collection. Again, a pictorial portrayal of this gives ...



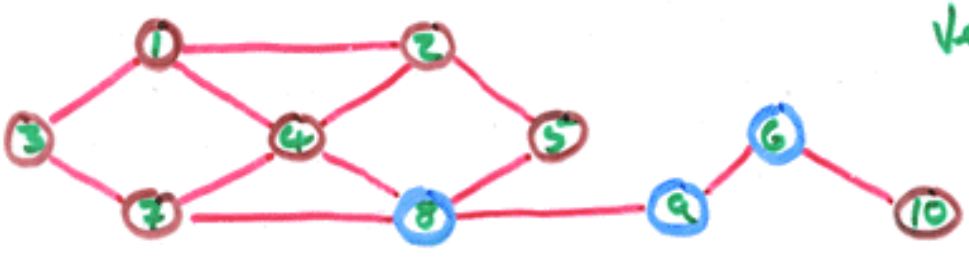
For our final graph problem, we'll look at depth-first searches. This involves starting at some vertex v and then recursively traversing all vertices adjacent to v. If the graph is a tree, this takes $O(e)$ time.

For general graphs, we need to avoid getting stuck in cycles! This is easily done by using known to mean visited. If we initialize v.known = false for all vertices, then a generic depth first search might be ...

```
void dfs ( Vertex v )
  {
    v.known = true;
    for each w adjacent to v
      if ( ! w.known )  dfs(w);
  }
```

This will work for connected undirected graphs; but if the graph is directed, but not strongly connected (i.e. does not have a path from every vertex to every other vertex), then this process will stop prematurely. This is easily fixed by restarting dfs, if necessary, at the 'next' unknown vertex. This gives an $O(n+e)$ traversal of all the vertices.

To illustrate how we might use this approach, we'll investigate how 'connected' a graph is. An undirected graph is biconnected if it has no vertices whose removal would disconnect the graph into two or more components — such vertices are called articulation points. Obvious applications could be the stability of computer networks, electricity power nets, or transportation studies. In our standard example...



Vertices 6, 8 and 9 are articulation points, so this graph isn't biconnected