DATA STRUCTURES

ADT

In this course, we won't delve into the full theory of object-oriented design. We'll concentrate on the pre-cursor of OO design: abstract data types (ADTs). A theory for the full object oriented approach is readily built on the ideas for abstract data types.

An **abstract data type** is a data structure and a collection of functions or procedures which operate on the data structure.

To align ourselves with OO theory, we'll call the functions and procedures **methods** and the data structure and its methods a **class**, *i.e.* we'll call our ADTs classes. However our classes do not have the full capabilities associated with classes in OO theory. An instance of the class is called an *object* . Objects represent objects in the real world and appear in programs as variables of a type defined by the class. These terms have exactly the same meaning in OO design methodologies, but they have additional properties such as inheritance that we will not discuss here.

It is important to note the object orientation is a *design methodology*. As a consequence, it is possible to write OO programs using languages such as C, Ada and Pascal. The so-called OO languages such as C++ and Eiffel simply provide some compiler support for OO design: this support must be provided by the programmer in non-OO languages.

Constructors and destructors

The create and destroy methods - often called constructors and destructors - are usually implemented for any abstract data type. Occasionally, the data type's use or semantics are such that there is only ever one object of that type in a program. In that case, it is possible to hide even the object's `handle' from the user. However, even in these cases, constructor and destructor methods are often provided.

Of course, specific applications may call for additional methods, e.g. we may need to join two collections (form a union in set terminology) - or may not need all of these.

One of the aims of good program design would be to ensure that additional requirements are easily handled.

Data Structure

To construct an abstract software model of a collection, we start by building the formal specification. The first component of this is the name of a data type - this is the type of objects that belong to the `collection` *class*. In C, we use `typedef` to define a new type which is a pointer to a structure:

```
typedef struct collection_struct *collection;
```

Note that we are defining a pointer to a structure only; we have not specified details of the *attributes* of the structure. We are deliberately deferring this - the details of the implementation are irrelevant at this stage. We are only concerned with the abstract behaviour of the collection. *In fact, as we will see later, we want to be able to substitute different data structures for the actual implementation of the collection, depending on our needs.*

The `typedef` declaration provides us with a C type (*class* in OO design parlance), `collection`. We can declare *objects* of type `collection` wherever needed. Although C forces us to reveal that the handle for objects of the class is a pointer, it is better to take an abstract view: we regard variables of type `collection` simply as *handles* to objects of the class and forget that the variables are actually C pointers.

abstract data type (ADT)

A data structure and a set of operations which can be performed on it. A class in object-oriented design is an ADT. However, classes have additional properties (inheritance and polymorphism) not normally associated with ADTs.

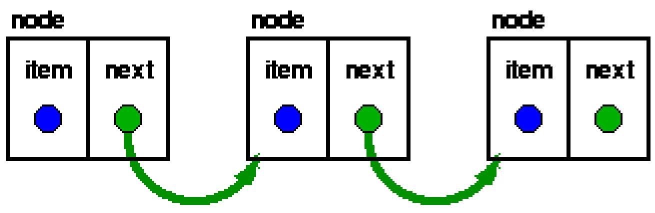
Lists

The array implementation of our collection has one serious drawback: you must know the maximum number of items in your collection when you create it. This presents problems in programs in which this maximum number cannot be predicted accurately when the program starts up. Fortunately, we can use a structure called a linked list to overcome this limitation.

Linked lists

The linked list is a very flexible **dynamic data structure**: items may be added to it or deleted from it at will. A programmer need not worry about how many items a program will have to accommodate: this allows us to write robust programs which require much less maintenance. A very common source of problems in program maintenance is the need to increase the capacity of a program to handle larger collections: even the most generous allowance for growth tends to prove inadequate over time!

In a linked list, each item is allocated space as it is added to the list. A link is kept with each item to the next item in the list.



Each node of the list has two elements

1. the item being stored in the list *and*
2. a pointer to the next item in the list

The last node in the list contains a NULL pointer to indicate that it is the end or *tail* of the list.

As items are added to a list, memory for a node is dynamically allocated. Thus the number of items that may be added to a list is limited only by the amount of memory available.

Handle for the list

The variable (or handle) which represents the list is simply a pointer to the node at the *head* of the list.

Adding to a list

The simplest strategy for adding an item to a list is to:

a. allocate space for a new node,
b. copy the item into it,
c. make the new node's `next` pointer point to the current head of the list *and*
d. make the head of the list point to the newly allocated node.

This strategy is fast and efficient, but each item is added to the head of the list.

An alternative is to create a structure for the list which contains both head and tail pointers:

```
struct fifo_list {
         struct node *head;
         struct node *tail;
         };
```

The code for `AddToCollection` is now trivially modified to make a list in which the item most recently added to the list is the list's tail.

The specification remains identical to that used for the array implementation: the `max_item` parameter to `ConsCollection` is simply ignored [7]

Thus we only need to change the implementation. As a consequence, applications which use this object will need no changes. The ramifications for the cost of software maintenance are significant.

The data structure is changed, but since the details (the attributes of the object or the elements of the structure) are hidden from the user, there is no impact on the user's program.

Points to note:

a. This implementation of our collection can be substituted for the first one with no changes to a client's program. With the exception of the added flexibility that any number of items may be added to our collection, this implementation provides exactly the same high level behaviour as the previous one.
b. The linked list implementation has exchanged flexibility for efficiency - on most systems, the system call to allocate memory is relatively expensive. Pre-allocation in the array-based implementation is generally more efficient. More examples of such trade-offs will be found later.

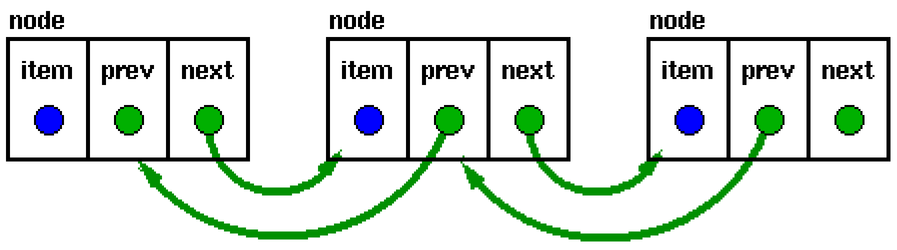
The study of data structures and algorithms will enable you to make the implementation decision which most closely matches your users' specifications.

List variants

Circularly Linked Lists

By ensuring that the tail of the list is always pointing to the head, we can build a circularly linked list. If the external pointer (the one in `struct t_node` in our implementation), points to the current "tail" of the list, then the "head" is found trivially via `tail->next`, permitting us to have either LIFO or FIFO lists with only one external pointer. In modern processors, the few bytes of memory saved in this way would probably not be regarded as significant. A circularly linked list would more likely be used in an application which required "round-robin" scheduling or processing.

Doubly Linked Lists



Doubly linked lists have a pointer to the preceding item as well as one to the next.

They permit scanning or searching of the list in both directions. (To go backwards in a simple list, it is necessary to go back to the start and scan forwards.) Many applications require searching backwards and forwards through sections of a list: for example, searching for a common name like "Kim" in a Korean telephone directory would probably need much scanning backwards and forwards through a small region of the whole list, so the backward links become very useful. In this case, the node structure is altered to have two links:

```
struct t_node {
     void *item;
     struct t_node *previous;
     struct t_node *next;
     } node;
```

Lists in arrays

Although this might seem pointless (Why impose a structure which has the overhead of the "next" pointers on an array?), this is just what memory allocators do to manage available space.

Memory is just an array of words. After a series of memory allocations and de-allocations, there are blocks of free memory scattered throughout the available heap space. In order to be able to re-use this memory, memory allocators will usually link freed blocks together in a *free list* by writing pointers to the next free block in the block itself. An external free list pointer pointer points to the first block in the free list. When a new block of memory is requested, the allocator will generally scan the free list looking for a freed block of suitable size and delete it from the free list (re-linking the free list around the deleted block). Many variations of memory allocators have been proposed: refer to a text on operating systems or implementation of functional languages for more details. The entry in the index under *garbage collection* will probably lead to a discussion of this topic.

STACKS

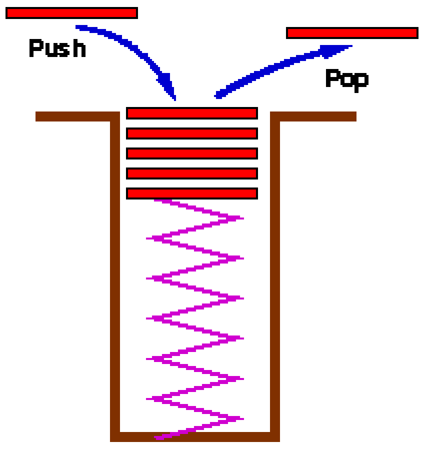
Another way of storing data is in a stack. A stack is generally implemented with only two principle operations (apart from a constructor and destructor methods):

`push`	adds an item to a stack
`pop`	extracts the most recently pushed item from the stack

Other methods such as

`top`	returns the item at the top *without removing it* [9]
`isempty`	determines whether the stack has anything in it

are sometimes added.



A common model of a stack is a plate or coin stacker. Plates are "pushed" onto to the top and "popped" off the top.

Stacks form Last-In-First-Out (LIFO) queues and have many applications from the parsing of algebraic expressions to ...

A formal specification of a stack class would look like:

```
typedef struct t_stack *stack;

stack ConsStack( int max_items, int item_size );
/* Construct a new stack
   Pre-condition: (max_items > 0) && (item_size > 0)
   Post-condition: returns a pointer to an empty stack
*/

void Push( stack s, void *item );
/* Push an item onto a stack
   Pre-condition: (s is a stack created by a call to ConsStack) &&
                  (existing item count < max_items) &&
                  (item != NULL)
   Post-condition: item has been added to the top of s
*/
```

```
void *Pop( stack s );
/* Pop an item of a stack
   Pre-condition: (s is a stack created by a call to
                  ConsStack) &&
                  (existing item count >= 1)
   Post-condition: top item has been removed from s
*/
```

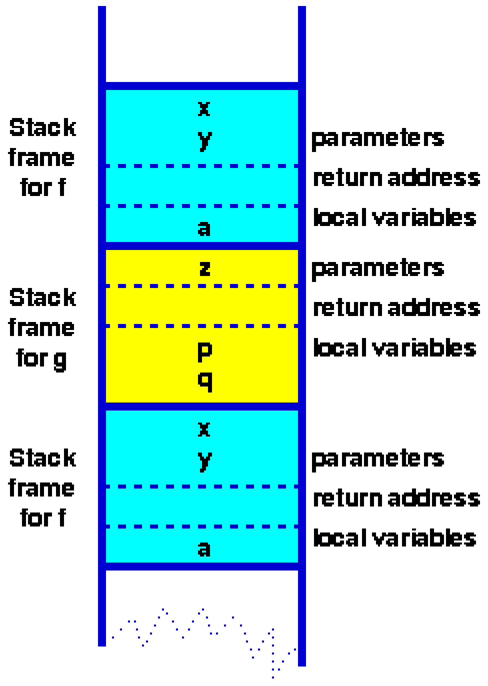
Points to note:

a. A stack is simply another collection of data items and thus it would be possible to use exactly the same specification as the one used for our general collection. However, collections with the LIFO semantics of stacks are so important in computer science that it is appropriate to set up a limited specification appropriate to stacks only.
b. Although a linked list implementation of a stack is possible (adding and deleting from the head of a linked list produces exactly the LIFO semantics of a stack), the most common applications for stacks have a space restraint so that using an array implementation is a natural and efficient one (In most operating systems, allocation and de-allocation of memory is a relatively expensive operation, there is a penalty for the flexibility of linked list implementations.).

Stack Frames

Almost invariably, programs compiled from modern high level languages (even C!) make use of a stack frame for the working memory of each procedure or function invocation. When any procedure or function is called, a number of words - the stack frame - is pushed onto a program stack. When the procedure or function returns, this frame of data is popped off the stack.

As a function calls another function, first its arguments, then the return address and finally space for local variables is pushed onto the stack. Since each function runs in its own "environment" or **context**, it becomes possible for a function to call itself - a technique known as *recursion*. This capability is extremely useful and extensively used - because many problems are elegantly specified or solved in a recursive way.



Program stack after executing a pair of mutually recursive functions:

```
function f(int x, int y) {
    int a;
    if ( term_cond ) return ...;
    a = .....;
    return g(a);
    }

function g(int z) {
    int p,q;
    p = ...; q = ...;
    return f(p,q);
    }
```

Note how all of function `f` and `g`'s environment (their parameters and local variables) are found in the stack frame. When `f` is called a second time from `g`, a new frame for the second invocation of `f` is created.

push, pop
Generic terms for adding something to, or removing something from a stack

context

The environment in which a function executes: includes argument values, local variables and global variables. All the context except the global variables is stored in a stack frame.

stack frames

The data structure containing all the data (arguments, local variables, return address, *etc*) needed each time a procedure or function is called.

Computer systems are often used to store large amounts of data from which individual records must be retrieved according to some search criterion. Thus the efficient storage of data to facilitate fast searching is an important issue. In this section, we shall investigate the performance of some searching algorithms and the data structures which they use.

Sequential Searches

Let's examine how long it will take to find an item matching a key in the collections we have discussed so far. We're interested in:

a. the average time
b. the worst-case time and
c. the best possible time.

However, we will generally be most concerned with the worst-case time as calculations based on worst-case times can lead to guaranteed performance predictions. Conveniently, the worst-case times are generally easier to calculate than average times.

If there are ***n*** items in our collection - whether it is stored as an array or as a linked list - then it is obvious that in the worst case, when there is no item in the collection with the desired key, then ***n*** comparisons of the key with keys of the items in the collection will have to be made.

To simplify analysis and comparison of algorithms, we look for a dominant operation and count the number of times that dominant operation has to be performed. In the case of searching, the dominant operation is the comparison, since the search requires n comparisons in the worst case, we say this is a ***O(n)*** (pronounce this "big-Oh-n" or "Oh-n") algorithm. The best case - in which the first comparison returns a match - requires a single comparison and is ***O(1)***. The average time depends on the probability that the key will be found in the collection - this is something that we would not expect to know in the majority of cases. Thus in this case, as in most others, estimation of the average time is of little utility. If the performance of the system is vital, i.e. it's part of a life-critical system, then we must use the worst case in our design calculations as it represents the best guaranteed performance.

Binary Search

However, if we place our items in an array and sort them in either ascending or descending order on the key first, then we can obtain much better performance with an algorithm called **binary search**.

In binary search, we first compare the key with the item in the middle position of the array. If there's a match, we can return immediately. If the key is less than the middle key, then the item sought must lie in the lower half of the array; if it's greater then the item sought must lie in the upper half of the array. So we repeat the procedure on the lower (or upper) half of the array.

Our `FindInCollection` function can now be implemented:

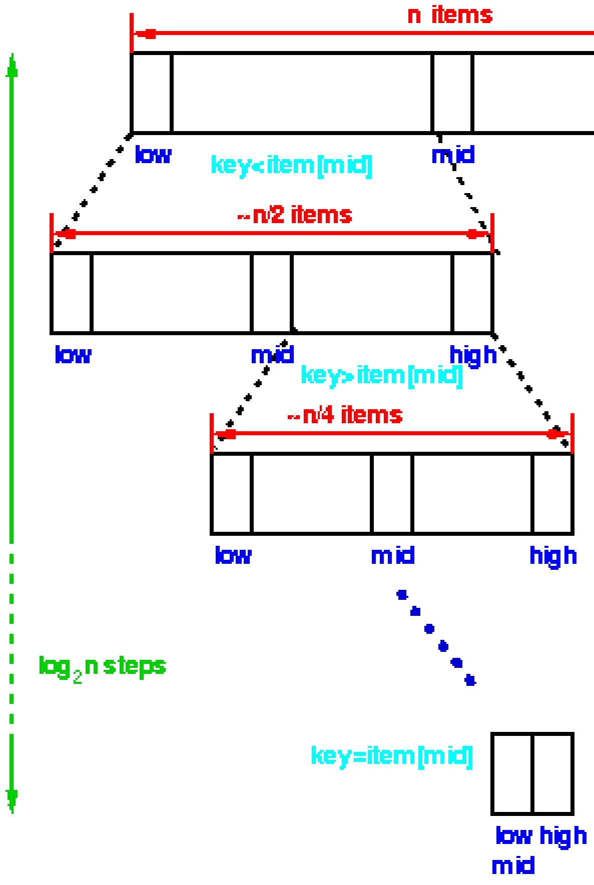
```
static void *bin_search( collection c, int low, int high, void *key ) {
          int mid;
          /* Termination check */
          if (low > high) return NULL;
          mid = (high+low)/2;
          switch (memcmp(ItemKey(c->items[mid]),key,c->size)) {
                    /* Match, return item found */
                    case 0: return c->items[mid];
                    /* key is less than mid, search lower half */
                    case -1: return bin_search( c, low, mid-1, key);
                    /* key is greater than mid, search upper half */
                    case 1: return bin_search( c, mid+1, high, key );
                    default : return NULL;
                    }
          }

void *FindInCollection( collection c, void *key ) {
/* Find an item in a collection
   Pre-condition:
          c is a collection created by ConsCollection
          c is sorted in ascending order of the key
          key != NULL
   Post-condition: returns an item identified by key if
   one exists, otherwise returns NULL
*/
          int low, high;
          low = 0; high = c->item_cnt-1;
          return bin_search( c, low, high, key );
}
```

Points to note:

a. `bin_search` is recursive: it determines whether the search key lies in the lower or upper half of the array, then calls itself on the appropriate half.
b. There is a termination condition (two of them in fact!)
   i. If `low > high` then the partition to be searched has no elements in it *and*
   ii. If there is a match with the element in the middle of the current partition, then we can return immediately.
c. `AddToCollection` will need to be modified to ensure that each item added is placed in its correct place in the array. The procedure is simple:
   i. Search the array until the correct spot to insert the new item is found,
   ii. Move all the following items up one position *and*
   iii. Insert the new item into the empty position thus created.
d. `bin_search` is declared `static`. It is a local function and is not used outside this class: if it were not declared static, it would be exported and be available to all parts of the program. The static declaration also allows other classes to use the same name internally.

`static` reduces the visibility of a function an should be used wherever possible to control access to functions!

Analysis



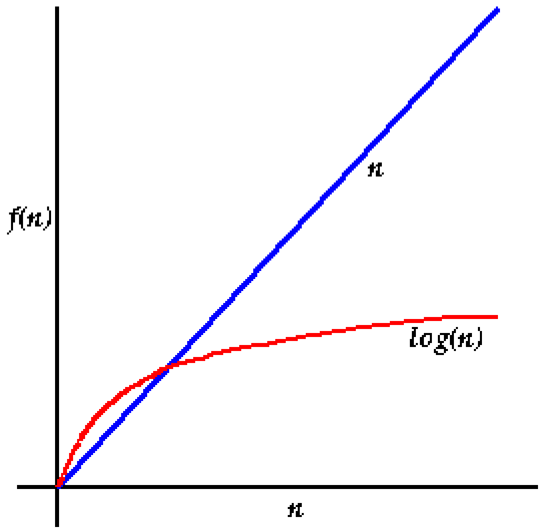
Each step of the algorithm divides the block of items being searched in half. We can divide a set of ***n*** items in half at most **log**$_2$ ***n*** times.

Thus the running time of a binary search is proportional to **log** ***n*** and we say this is a ***O*****(log** ***n)*** algorithm.

Binary search requires a more complex program than our original search and thus for *small* ***n*** it may run slower than the simple linear search. However, for large ***n***,

$$\lim_{n \to \infty} \frac{\log n}{n} = 0$$

Thus at large ***n***, **log** ***n*** is *much* smaller than ***n***, consequently an ***O*****(log** ***n)*** algorithm is *much* faster than an ***O(n)*** one.



Plot of ***n*** and **log** ***n*** *vs* ***n*** .

We will examine this behaviour more formally in a later section. First, let's see what we can do about the insertion (`AddToCollection`) operation.

In the worst case, insertion may require ***n*** operations to insert into a sorted list.

1. We can find the place in the list where the new item belongs using binary search in ***O*****(log** ***n)*** operations.
2. However, we have to shuffle all the following items up one place to make way for the new one. In the worst case, the new item is the first in the list, requiring ***n*** move operations for the shuffle!

A similar analysis will show that deletion is also an ***O(n)*** operation.

If our collection is static, *ie* it doesn't change very often - if at all - then we may not be concerned with the time required to change its contents: we may be prepared for the initial build of the collection and the occasional insertion and deletion to take some time. In return, we will be able to use a simple data structure (an array) which has little memory overhead.

However, if our collection is large and dynamic, *ie* items are being added and deleted continually, then we can obtain considerably better performance using a data structure called a tree.

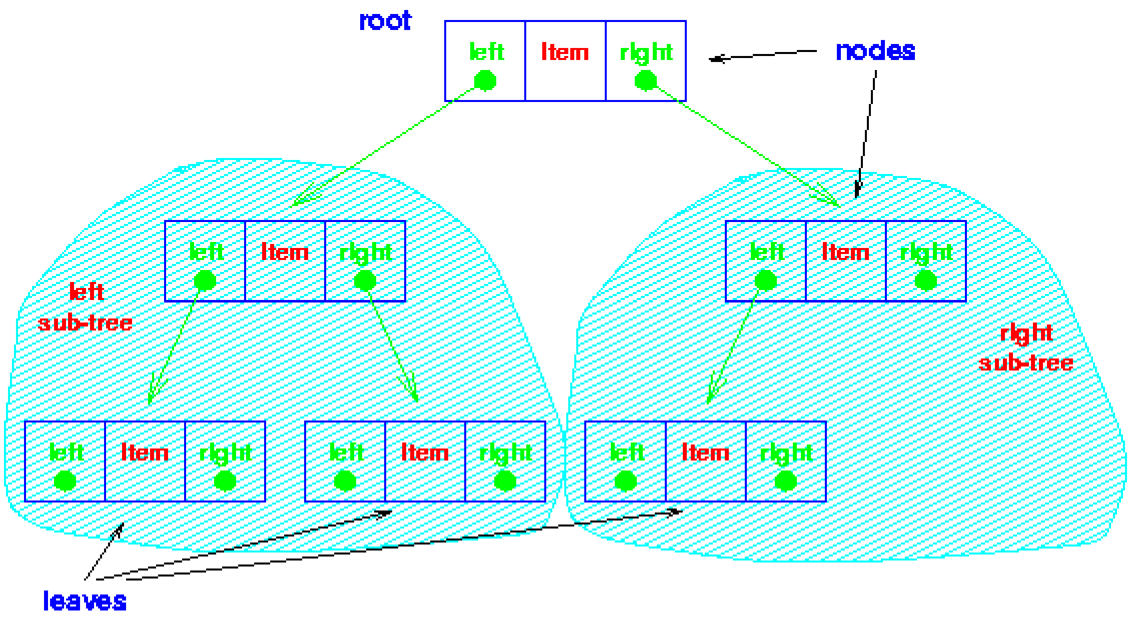
Big Oh
A notation formally describing the set of all functions which are bounded above by a nominated function.
Binary Search
A technique for searching an ordered list in which we first check the middle item and - based on that comparison - "discard" half the data. The same procedure is then applied to the remaining half until a match is found or there are no more items left.

Binary Trees

The simplest form of tree is a **binary tree**. A binary tree consists of

a. a *node* (called the **root** node) and
b. left and right sub-trees.
Both the sub-trees are themselves binary trees.

You now have a *recursively defined data structure*. (It is also possible to define a list recursively: can you see how?)



A binary tree

The nodes at the lowest levels of the tree (the ones with no sub-trees) are called **leaves**.

In an *ordered binary tree*,

1. the keys of all the nodes in the left sub-tree are less than that of the root,
2. the keys of all the nodes in the right sub-tree are greater than that of the root,
3. the left and right sub-trees are themselves ordered binary trees.

Data Structure

The data structure for the tree implementation simply adds left and right pointers in place of the next pointer of the linked list implementation. [Load the tree struct.]

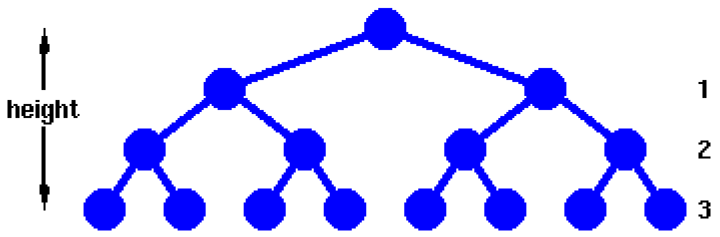
The `AddToCollection` method is, naturally, recursive. [Load the `AddToCollection` method.]

Similarly, the `FindInCollection` method is recursive. [Load the `FindInCollection` method.]

Analysis

Complete Trees

Before we look at more general cases, let's make the optimistic assumption that we've managed to fill our tree neatly, *ie* that each leaf is the same 'distance' from the root.



This forms a *complete tree*, whose *height* is defined as the number of links from the root to the deepest leaf.

A complete tree

First, we need to work out how many nodes, ***n***, we have in such a tree of height, ***h***.

Now,

$\boldsymbol{n} = 1 + 2^1 + 2^2 + ... + 2^{\boldsymbol{h}}$
From which we have,
$\boldsymbol{n} = 2^{\boldsymbol{h}+1} - 1$
and
$\boldsymbol{h} = \textbf{floor}(\ \textbf{log}_2 \boldsymbol{n}\ )$

Examination of the `Find` method shows that in the worst case, $\boldsymbol{h}+1$ or **ceiling(log₂*n*)** comparisons are needed to find an item. This is the same as for binary search.

However, `Add` also requires $\textbf{ceiling}(\ \textbf{log}_2 \boldsymbol{n}\ )$ comparisons to determine where to add an item. Actually adding the item takes a constant number of operations, so we say that a binary tree requires **O(log*n*)** operations for *both* adding and finding an item - a considerable improvement over binary search for a *dynamic* structure which often requires addition of new items.

Deletion is also an **O(log*n*)** operation.

General binary trees

However, in general addition of items to an ordered tree will not produce a complete tree. The worst case occurs if we add an ordered list of items to a tree.

What will happen? Think before you click here!

This problem is readily overcome: we use a structure known as a heap. However, before looking at heaps, we should formalise our ideas about the complexity of algorithms by defining carefully what **O(f(*n*))** means.

Root Node
: Node at the "top" of a tree - the one from which all operations on the tree commence. The root node may not exist (a NULL tree with no nodes in it) or have 0, 1 or 2 children in a binary tree.

Leaf Node
: Node at the "bottom" of a tree - farthest from the root. Leaf nodes have no children.

Complete Tree
: Tree in which each leaf is at the same distance from the root. A more precise and formal definition of a complete tree is set out later.

Height
: Number of nodes which must be traversed from the root to reach a leaf of a tree.

HEAPS

Heaps are based on the notion of a **complete tree**, for which we gave an informal definition earlier.

Formally:

A binary tree is **completely full** if it is of height, *h*, and has 2^{h+1}-1 nodes.

A binary tree of height, *h*, is **complete** *iff*

a. it is empty *or*
b. its left subtree is complete of height *h*-1 and its right subtree is completely full of height *h*-2 *or*
c. its left subtree is completely full of height *h*-1 and its right subtree is complete of height *h*-1.

A complete tree is filled from the left:

- all the leaves are on
    - the same level *or*
    - two adjacent ones *and*
- all nodes at the lowest level are as far to the left as possible.

Heaps

A binary tree has the **heap property** *iff*

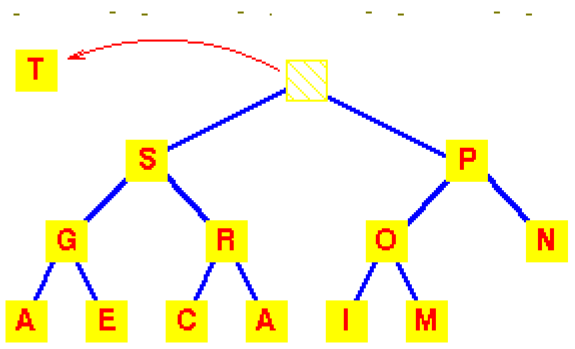
a. it is empty *or*
b. the key in the root is larger than that in either child and both subtrees have the heap property.

A heap can be used as a priority queue: the highest priority item is at the root and is trivially extracted. But if the root is deleted, we are left with two sub-trees and we must *efficiently* re-create a single tree with the heap property.
The value of the heap structure is that we can both extract the highest priority item and insert a new one in **O(log*n*)** time.

How do we do this?

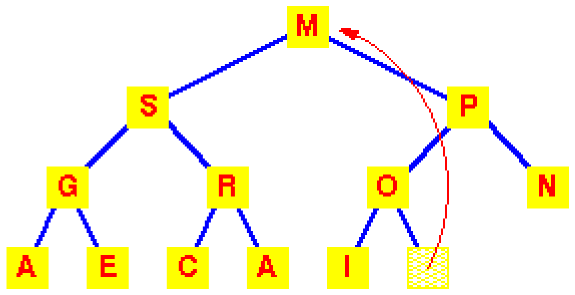
Let's start with this heap.

A deletion will remove the T at the root.



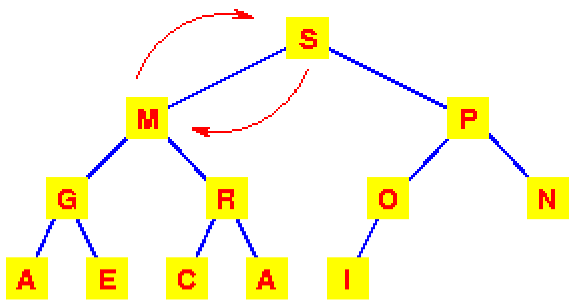
To work out how we're going to maintain the heap property, use the fact that a complete tree is filled from the left. So that the position which must become empty is the one occupied by the M.

Put it in the vacant root position.



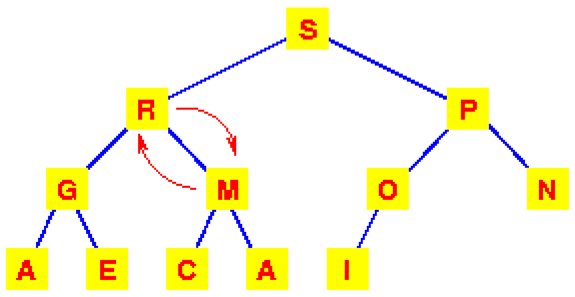
This has violated the condition that the root must be greater than each of its children.

So interchange the M with the larger of its children.



The left subtree has now lost the heap property.

So again interchange the M with the larger of its children.



This tree is now a heap again, so we're finished.

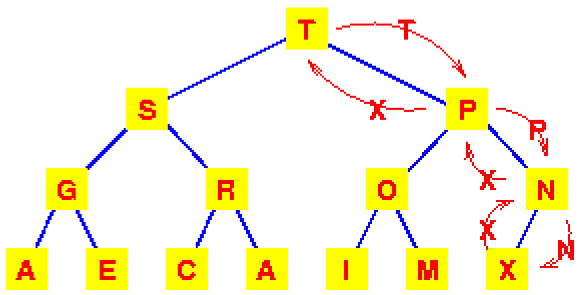
We need to make at most *h* interchanges of a root of a subtree with one of its children to fully restore the heap property. Thus deletion from a heap is **O(*h*)** or **O(log*n*)**.

Addition to a heap

To add an item to a heap, we follow the reverse procedure.

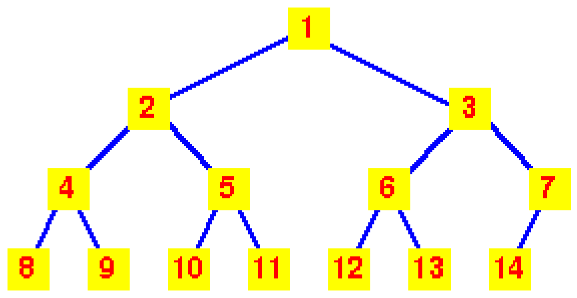
Place it in the next leaf position and move it **up**.

Again, we require **O(*h*)** or **O(log*n*)** exchanges.



Storage of complete trees

The properties of a complete tree lead to a very efficient storage mechanism using *n* sequential locations in an array.

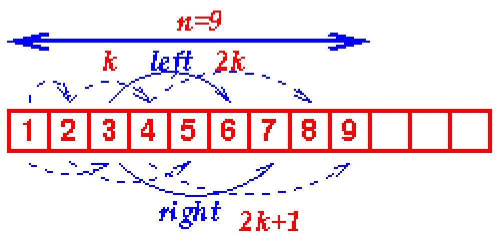


If we number the nodes from 1 at the root and place:

- the left child of node *k* at position 2*k*
- the right child of node *k* at position 2*k*+1

Then the 'fill from the left' nature of the complete tree ensures that the heap can be stored in consecutive locations in an array.

Viewed as an array, we can see that the *n*th node is always in index position *n*.



The code for extracting the highest priority item from a heap is, naturally, recursive. Once we've extracted the root (highest priority) item and swapped the last item into its place, we simply call `MoveDown` recursively until we get to the bottom of the tree.

Click here to load heap_delete.c

Note the macros `LEFT` and `RIGHT` which simply encode the relation between the index of a node and its left and right children. Similarly the `EMPTY` macro encodes the rule for determining whether a sub-tree is empty or not.

Inserting into a heap follows a similar strategy, except that we use a `MoveUp` function to move the newly added item to its correct place. (For the `MoveUp` function, a further macro which defines the `PARENT` of a node would normally be added.)

Heaps provide us with a method of sorting, known as heapsort. However, we will examine and analyse the simplest method of sorting first.

Hash Tables

Direct Address Tables

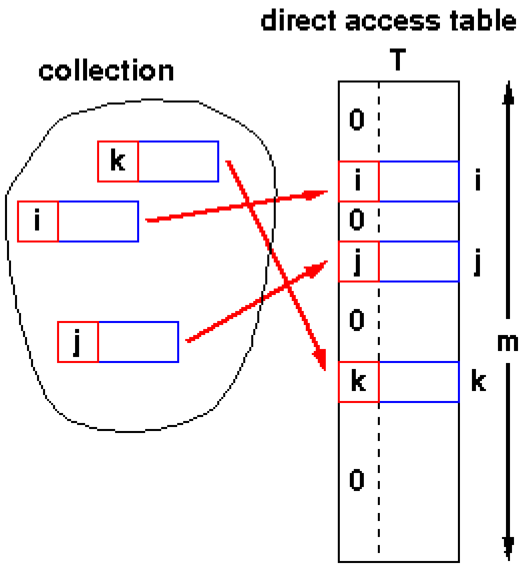
If we have a collection of **n** elements whose keys are unique integers in (1,**m**), where **m** >= **n**, then we can store the items in a *direct address* table, **T[m]**,
where $\mathbf{T_i}$ is either empty or contains one of the elements of our collection.

Searching a direct address table is clearly an **O(1)** operation:
for a key, **k**, we access $\mathbf{T_k}$,

- if it contains an element, return it,
- if it doesn't then return a NULL.

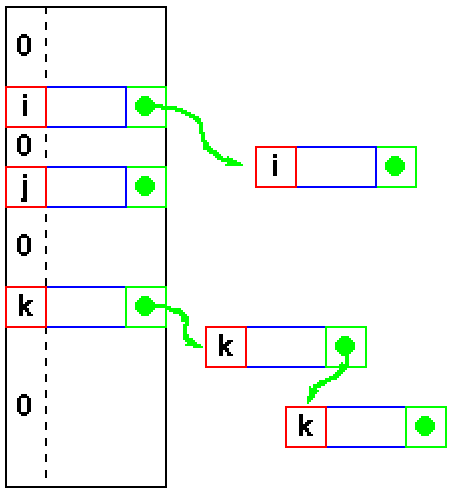
There are two constraints here:

1. the keys must be unique, and
2. the range of the key must be severely bounded.



If the keys are not unique, then we can simply construct a set of **m** lists and store the heads of these lists in the direct address table. The time to find an element matching an input key will still be **O(1)**.

However, if each element of the collection has some other distinguishing feature (other than its key), and if the maximum number of duplicates is $\mathbf{n_{dup}}^{max}$, then searching for a specific element is $\mathbf{O(n_{dup}}^{max}\mathbf{)}$. If duplicates are the exception rather than the rule, then $\mathbf{n_{dup}}^{max}$ is much smaller than **n** and a direct address table will provide good performance. But if $\mathbf{n_{dup}}^{max}$ approaches **n**, then the time to find a specific element is **O(n)** and a tree structure will be more efficient.



The range of the key determines the size of the direct address table and may be too large to be practical. For instance it's not likely that you'll be able to use a direct address table to store elements which have arbitrary 32-bit integers as their keys for a few years yet!

Direct addressing is easily generalised to the case where there is a function,

h(k) => (1,**m**)

which maps each value of the key, **k**, to the range (1,**m**). In this case, we place the element in **T[h(k)]** rather than **T[k]** and we can search in **O(1)** time as before.

Mapping functions

The direct address approach requires that the function, **h(k)**, is a one-to-one mapping from each **k** to integers in (1,**m**). Such a function is known as a **perfect hashing function**: it maps each key to a distinct integer within some manageable range and enables us to trivially build an **O(1)** search time table.

Unfortunately, finding a perfect hashing function is not always possible. Let's say that we can find a **hash function**, **h(k)**, which maps *most* of the keys onto unique integers, but maps a small number of keys on to the same integer. If the number of **collisions** (cases where multiple keys map onto the same integer), is sufficiently small, then *hash tables* work quite well and give **O(1)** search times.

Handling the collisions

In the small number of cases, where multiple keys map to the same integer, then elements with different keys may be stored in the same "slot" of the hash table. It is clear that when the hash function is used to locate a potential match, it will be necessary to compare the key of that element with the search key. But there may be more than one element which should be stored in a single slot of the table. Various techniques are used to manage this problem:

1. chaining,
2. overflow areas,
3. re-hashing,
4. using neighbouring slots (linear probing),
5. quadratic probing,
6. random probing, ...

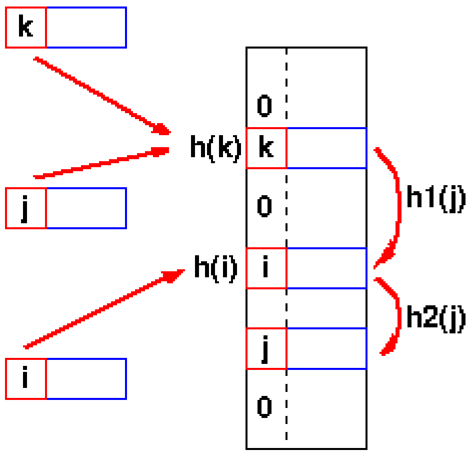
Chaining

One simple scheme is to chain all collisions in lists attached to the appropriate slot. This allows an unlimited number of collisions to be handled and doesn't require *a priori* knowledge of how many elements are contained in the collection. The tradeoff is the same as with linked lists versus array implementations of collections: linked list overhead in space and, to a lesser extent, in time.

Re-hashing

Re-hashing schemes use a second hashing operation when there is a collision. If there is a further collision, we *re-hash* until an empty "slot" in the table is found.

The re-hashing function can either be a new function or a re-application of the original one. As long as the functions are applied to a key in the same order, then a sought key can always be located.



h(j)=h(k), so the next hash function, **h1** is used. A second collision occurs, so **h2** is used.

Linear probing

One of the simplest re-hashing functions is +1 (or -1), *ie* on a collision, look in the neighbouring slot in the table. It calculates the new address extremely quickly and may be extremely efficient on a modern RISC processor due to efficient cache utilisation (*cf.* the discussion of linked list efficiency).

The animation gives you a practical demonstration of the effect of linear probing: it also implements a quadratic re-hash function so that you can compare the difference.

Clustering

Linear probing is subject to a **clustering** phenomenon. Re-hashes from one location occupy a block of slots in the table which "grows" towards slots to which other keys hash. This exacerbates the collision problem and the number of re-hashed can become large.

Quadratic Probing

Better behaviour is usually obtained with **quadratic probing**, where the secondary hash function depends on the re-hash index:

address = h(key) + c $\boldsymbol{i^2}$

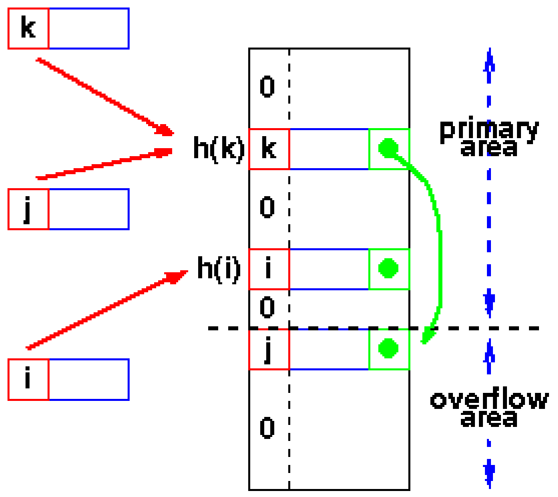
on the $\boldsymbol{t^{th}}$ re-hash. (A more complex function of ***i*** may also be used.) Since keys which are mapped to the same value by the primary hash function follow the same sequence of addresses, quadratic probing shows **secondary clustering**. However, secondary clustering is not nearly as severe as the clustering shown by linear probes.

Re-hashing schemes use the originally allocated table space and thus avoid linked list overhead, but require advance knowledge of the number of items to be stored.

However, the collision elements are stored in slots to which other key values map directly, thus the potential for multiple collisions increases as the table becomes full.

Overflow area

Another scheme will divide the pre-allocated table into two sections: the *primary area* to which keys are mapped and an area for collisions, normally termed the *overflow area*.



When a collision occurs, a slot in the overflow area is used for the new element and a link from the primary slot established as in a chained system. This is essentially the same as chaining, except that the overflow area is pre-allocated and thus possibly faster to access. As with re-hashing, the maximum number of elements must be known in advance, but in this case, two parameters must be estimated: the optimum size of the primary and overflow areas.

Of course, it is possible to design systems with multiple overflow tables, or with a mechanism for handling overflow out of the overflow area, which provide flexibility without losing the advantages of the overflow scheme.

Summary: Hash Table Organization

Organization	**Advantages**	**Disadvantages**

Chaining	• Unlimited number of elements • Unlimited number of collisions	• Overhead of multiple linked lists
Re-hashing	• Fast re-hashing • Fast access through use of main table space	• Maximum number of elements must be known • Multiple collisions may become probable
Overflow area	• Fast access • Collisions don't use primary table space	• Two parameters which govern performance need to be estimated •

Animation

Hash table
: Tables which can be searched for an item in **O(1)** time using a hash function to form an address from the key.

hash function
: Function which, when applied to the key, produces a integer which can be used as an address in a hash table.

collision
: When a hash function maps two different keys to the same table address, a collision is said to occur.

linear probing
: A simple re-hashing scheme in which the next slot in the table is checked on a collision.

quadratic probing
: A re-hashing scheme in which a higher (usually 2nd) order function of the hash index is used to calculate the address.

clustering.
: Tendency for clusters of adjacent slots to be filled when linear probing is used.

secondary clustering.
: Collision sequences generated by addresses calculated with quadratic probing.

perfect hash function
: Function which, when applied to all the members of the set of items to be stored in a hash table, produces a unique set of integers within some suitable range.

Red-Black Trees

A *red-black tree* is a binary search tree with one extra attribute for each node: the *colour*, which is either red or black. We also need to keep track of the parent of each node, so that a red-black tree's node structure would be:

```
struct t_red_black_node {
    enum { red, black } colour;
    void *item;
    struct t_red_black_node *left,
                     *right,
                     *parent;
    }
```

For the purpose of this discussion, the NULL nodes which terminate the tree are considered to be the leaves and are coloured black.

Definition of a red-black tree

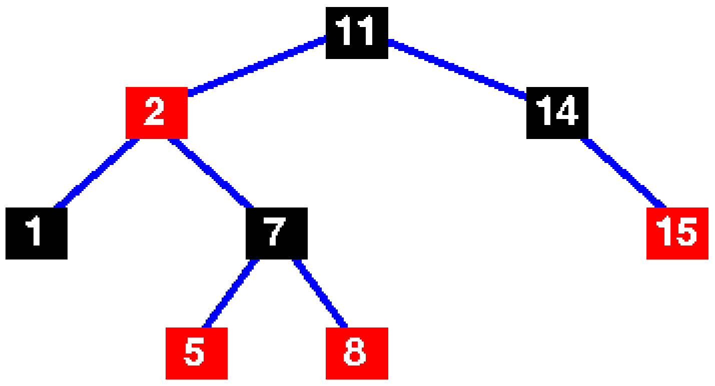
A red-black tree is a binary search tree which has the following *red-black properties*:

1. Every node is either red or black.
2. Every leaf (NULL) is black.

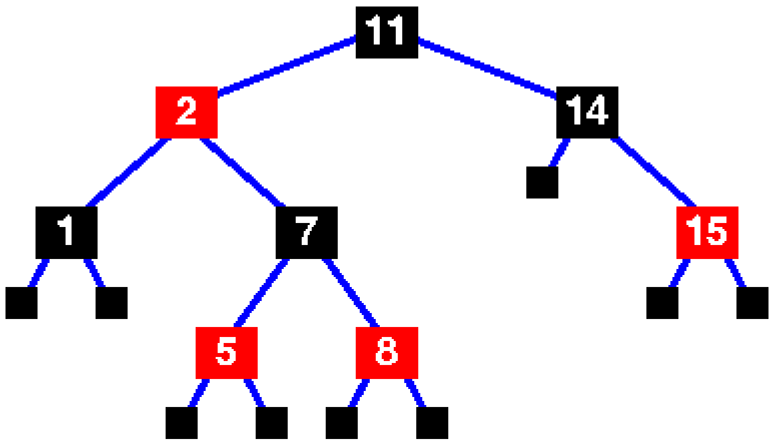
3. implies that on any path from the root to a leaf, red nodes must not be adjacent.

3. If a node is red, then both its children are black.
4. Every simple path from a node to a descendant leaf contains the same number of black nodes.

However, any number of black nodes may appear in a sequence.



A basic red-black tree



Basic red-black tree with the **sentinel** nodes added. Implementations of the red-black tree algorithms will usually include the sentinel nodes as a convenient means of flagging that you have reached a leaf node.

They are the NULL black nodes of property 2.

The number of black nodes on any path from, but not including, a node ***x*** to a leaf is called the *black-height* of a node, denoted **bh(x)**. We can prove the following lemma:

Lemma

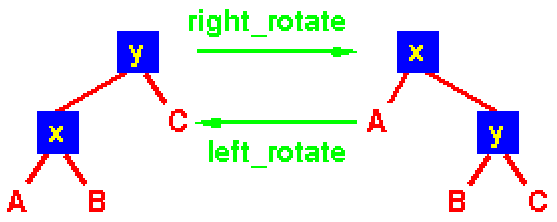
A red-black tree with ***n*** internal nodes has height at most **2log(*n*+1)**. *(For a proof, see Cormen, p 264)*

This demonstrates why the red-black tree is a good search tree: it can always be searched in **O(log n)** time.

As with heaps, additions and deletions from red-black trees destroy the red-black property, so we need to restore it. To do this we need to look at some operations on red-black trees.

Rotations

A rotation is a local operation in a search tree that preserves *in-order* traversal key ordering.



Note that in both trees, an in-order traversal yields:

```
A x B y C
```

The left_rotate operation may be encoded:

```
left_rotate( Tree T, node x ) {
    node y;
    y = x->right;
    /* Turn y's left sub-tree into x's right sub-tree */
    x->right = y->left;
    if ( y->left != NULL )
        y->left->parent = x;
    /* y's new parent was x's parent */
    y->parent = x->parent;
    /* Set the parent to point to y instead of x */
    /* First see whether we're at the root */
    if ( x->parent == NULL ) T->root = y;
    else
        if ( x == (x->parent)->left )
            /* x was on the left of its parent */
            x->parent->left = y;
        else
            /* x must have been on the right */
            x->parent->right = y;
    /* Finally, put x on y's left */
    y->left = x;
    x->parent = y;
    }
```

Insertion

Insertion is somewhat complex and involves a number of cases. Note that we start by inserting the new node, `x`, in the tree just as we would for any other binary tree, using the `tree_insert` function. This new node is labelled red, and possibly destroys the red-black property. The main loop moves up the tree, restoring the red-black property.

```
rb_insert( Tree T, node x ) {
    /* Insert in the tree in the usual way */
    tree_insert( T, x );
    /* Now restore the red-black property */
    x->colour = red;
    while ( (x != T->root) && (x->parent->colour == red) ) {
       if ( x->parent == x->parent->parent->left ) {
           /* If x's parent is a left, y is x's right 'uncle' */
           y = x->parent->parent->right;
           if ( y->colour == red ) {
               /* case 1 - change the colours */
               x->parent->colour = black;
               y->colour = black;
               x->parent->parent->colour = red;
               /* Move x up the tree */
               x = x->parent->parent;
               }
           else {
               /* y is a black node */
               if ( x == x->parent->right ) {
                   /* and x is to the right */
                   /* case 2 - move x up and rotate */
                   x = x->parent;
                   left_rotate( T, x );
                   }
               /* case 3 */
               x->parent->colour = black;
               x->parent->parent->colour = red;
               right_rotate( T, x->parent->parent );
               }
           }
       else {
           /* repeat the "if" part with right and left
              exchanged */
       }
    /* Colour the root black */
    T->root->colour = black;
    }
```

Here's an example of the insertion operation.

Animation

Examination of the code reveals only one loop. In that loop, the node at the root of the sub-tree whose red-black property we are trying to restore, `x`, may be moved up the tree *at least one level* in each iteration of the loop. Since the tree originally has **O(log n)** height, there are **O(log n)** iterations. The `tree_insert` routine also has **O(log n)** complexity, so overall the `rb_insert` routine also has **O(log n)** complexity.

Red-black trees

Trees which remain **balanced** - and thus guarantee **O(logn)** search times - in a dynamic environment. Or more importantly, since any tree can be re-balanced - but at considerable cost - can be re-balanced in **O(logn)** time.

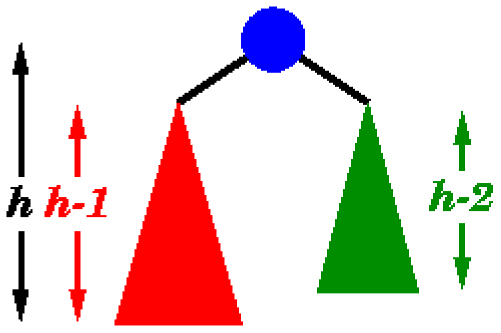
AVL tree

An **AVL tree** is another balanced binary search tree. Named after their inventors, **A**delson-**V**elskii and **L**andis, they were the first dynamically balanced trees to be proposed. Like red-black trees, they are not perfectly balanced, but pairs of sub-trees differ in height by at most 1, maintaining an ***O*(logn)** search time. Addition and deletion operations also take ***O*(logn)** time.

Definition of an AVL tree

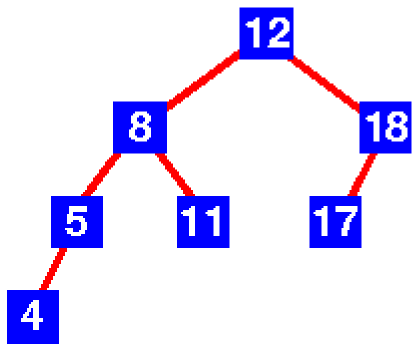
An AVL tree is a binary search tree which has the following properties:

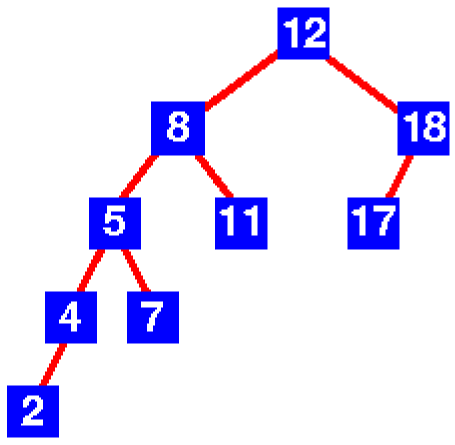
1. The sub-trees of every node differ in height by at most one.
2. Every sub-tree is an AVL tree.



Balance requirement for an AVL tree: the left and right sub-trees differ by at most 1 in height.

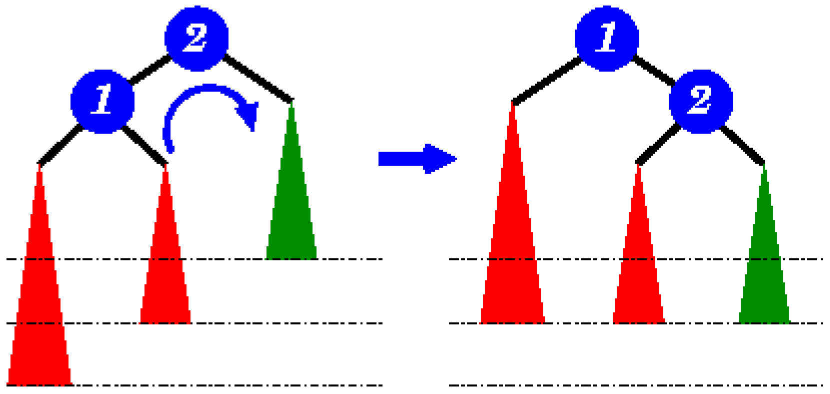
You need to be careful with this definition: it permits some apparently unbalanced trees! For example, here are some trees:

Tree	**AVL tree?**
	Yes Examination shows that *each* left sub-tree has a height 1 greater than each right sub-tree.

	No Sub-tree with root 8 has height 4 and sub-tree with root 18 has height 2

Insertion

As with the red-black tree, insertion is somewhat complex and involves a number of cases. Implementations of AVL tree insertion may be found in many textbooks: they rely on adding an extra attribute, the **balance factor** to each node. This factor indicates whether the tree is *left-heavy* (the height of the left sub-tree is 1 greater than the right sub-tree), *balanced* (both sub-trees are the same height) or *right-heavy* (the height of the right sub-tree is 1 greater than the left sub-tree). If the balance would be destroyed by an insertion, a rotation is performed to correct the balance.



A new item has been added to the left subtree of node 1, causing its height to become 2 greater than 2's right sub-tree (shown in green). A right-rotation is performed to correct the imbalance.

AVL trees

Trees which remain **balanced** - and thus guarantee **O(logn)** search times - in a dynamic environment. Or more importantly, since any tree can be re-balanced - but at considerable cost - can be re-balanced in **O(logn)** time.

Graphs

Minimum Spanning Trees

Greedy Algorithms

Many algorithms can be formulated as a finite series of guesses, *eg* in the Travelling Salesman Problem, we try (guess) each possible tour in turn and determine its cost. When we have tried them **all**, we know which one is the optimum (least cost) one. However, we must try them all before we can be certain that we know which is the optimum one, leading to an ***O(n!)*** algorithm.

Intuitive strategies, such as building up the salesman's tour by adding the city which is closest to the current city, can readily be shown to produce sub-optimal tours. As another example, an experienced chess player will not take an opponent's pawn with his queen - because that move produced the maximal gain, the capture of a piece - if his opponent is guarding that pawn with another pawn. In such games, you must look at *all* the moves ahead to ensure that the one you choose is in fact the optimal one. All chess players know that short-sighted strategies are good recipes for disaster!

There is a class of algorithms, the *greedy algorithms*, in which we can find a solution by using only knowledge available at the time the next choice (or guess) must be made. The problem of finding the Minimum Spanning Tree is a good example of this class.

The Minimum Spanning Tree Problem

Suppose we have a group of islands that we wish to link with bridges so that it is possible to travel from one island to any other in the group. Further suppose that (as usual) our government wishes to spend the absolute minimum amount on this project (because other factors like the cost of using, maintaining, etc, these bridges will probably be the responsibility of some future government ☺). The engineers are able to produce a cost for a bridge linking each possible pair of islands. The set of bridges which will enable one to travel from any island to any other at minimum capital cost to the government is the *minimum spanning tree*.

We will need some definitions first:

Graphs

A graph is a set of *vertices* and *edges* which connect them. We write:

G = (V,E)

where **V** is the set of vertices and the set of edges,

E = { (v_i,v_j) }
where v_i and v_j are in **V**.

Paths

A *path*, **p**, of length, **k**, through a graph is a sequence of connected vertices:
p = <v_0,v_1,...,v_k>
where, for all **i** in (0,**k**-1:
(v_i,v_{i+1}) is in **E**.

Cycles

A graph contains no *cycles* if there is no path of non-zero length through the graph, **p = <v_0,v_1,...,v_k>** such that **v_0 = v_k**.

Spanning Trees

A *spanning tree* of a graph, G, is a set of **|V|**-1 edges that connect all vertices of the graph.

Minimum Spanning Tree

In general, it is possible to construct multiple spanning trees for a graph, **G**. If a cost, c_{ij}, is associated with each edge, $e_{ij} = (v_i, v_j)$, then the minimum spanning tree is the set of edges, E_{span}, forming a spanning tree, such that:
$C = \text{sum}(\ c_{ij}\ |\ \text{all } e_{ij} \text{ in } E_{span}\ )$

is a minimum.

Kruskal's Algorithm

This algorithm creates a *forest* of trees. Initially the forest consists of **n** single node trees (and no edges). At each step, we add one (the cheapest one) edge so that it joins two trees together. If it were to form a cycle, it would simply link two nodes that were already part of a single connected tree, so that this edge would not be needed.

The basic algorithm looks like this:

```
Forest MinimumSpanningTree( Graph g, int n, double **costs ) {
   Forest T;
   Queue q;
   Edge e;
   T = ConsForest( g );
   q = ConsEdgeQueue( g, costs );
   for(i=0;i<(n-1);i++) {
       do {
          e = ExtractCheapestEdge( q );
       } while ( !Cycle( e, T ) );
       AddEdge( T, e );
   }
   return T;
}
```

The steps are:

1. The forest is constructed - with each node in a separate tree.
2. The edges are placed in a priority queue.
3. Until we've added **n**-1 edges,
    1. Extract the cheapest edge from the queue,
    2. If it forms a cycle, reject it,
    3. Else add it to the forest. Adding it to the forest will join two trees together.

Every step will have joined two trees in the forest together, so that at the end, there will only be one tree in T.

We can use a heap for the priority queue. The trick here is to detect cycles. For this, we need a *union-find* structure.

Union-find structure

To understand the union-find structure, we need to look at a *partition* of a set.

Partitions

A partitions is a set of sets of elements of a set.

- Every element of the set belong to one of the sets in the partition.
- No element of the set belong to more than one of the sub-sets.

or

- Every element of a set belongs to one *and only one* of the sets of a partition.

The forest of trees is a partition of the original set of nodes. Initially all the sub-sets have exactly one node in them. As the algorithm progresses, we form a union of two of the trees (sub-sets), until eventually the partition has only one sub-set containing all the nodes.

A partition of a set may be thought of as a set of *equivalence classes*. Each sub-set of the partition contains a set of equivalent elements (the nodes connected into one of the trees of the forest). This notion is the key to the cycle detection algorithm. For each sub-set, we denote one element as the *representative* of that sub-set or equivalence class. Each element in the sub-set is, somehow, equivalent and represented by the nominated representative.

As we add elements to a tree, we arrange that all the elements point to their representative. As we form a union of two sets, we simply arrange that the representative of one of the sets now points to any one of the elements of the other set.

So the test for a cycle reduces to: for the two nodes at the ends of the candidate edge, find their representatives. If the two representatives are the same, the two nodes are already in a connected tree and adding this edge would form a cycle. The search for the representative simply follows a chain of links.

Each node will need a representative pointer. Initially, each node is its own representative, so the pointer is set to NULL. As the initial pairs of nodes are joined to form a tree, the representative pointer of one of the nodes is made to point to the other, which becomes the representative of the tree. As trees are joined, the representative pointer of the representative of one of them is set to point to any element of the other. (Obviously, representative searches will be somewhat faster if one of the representatives is made to point directly to the other.)

Greedy operation

At no stage did we try to look ahead more than one edge - we simply chose the best one at any stage. Naturally, in some situations, this myopic view would lead to disaster! The simplistic approach often makes it difficult to prove that a greedy algorithm leads to the *optimal* solution. proof by contradiction is a common proof technique used: we demonstrate that if we didn't make the greedy choice now, a non-optimal solution would result. Proving the MST algorithm is, happily, one of the simpler proofs by contradiction!

Data structures for graphs

You should note that we have discussed graphs in an abstract way: specifying that they contain nodes and edges and using operations like `AddEdge`, `Cycle`, *etc*. This enables us to define an abstract data type *without* considering implementation details, such as how we will store the attributes of a graph! This means that a complete solution to, for example, the MST problem can be specified before we've even decided how to store the graph in the computer. However, representation issues can't be deferred forever, so we need to examine ways of representing graphs in a machine. As before, the performance of the algorithm will be determined by the data structure chosen.

Greedy algorithms
Algorithms which solve a problem by making the next step based on local knowledge alone - without looking ahead to determine whether the next step is the optimal one.

Equivalence Classes
The set of equivalence classes of a set is a **partition** of a set such that all the elements in each subset (or **equivalence class**) is related to every other element in the subset by an **equivalence relation**.

Union Find Structure
A structure which enables us to determine whether two sets are in fact the same set or not.

Kruskal's Algorithm
One of the two algorithms commonly used for finding a minimum spanning tree - the other is **Prim's algorithm**.

Dijkstra's Algorithm

Djikstra's algorithm (named after its discover, E.W. Dijkstra) solves the problem of finding the shortest path from a point in a graph (the *source*) to a destination. It turns out that one can find the shortest paths from a given source to *all* points in a graph in the same time, hence this problem is sometimes called the **single-source shortest paths** problem.

The somewhat unexpected result that *all* the paths can be found as easily as one further demonstrates the value of reading the literature on algorithms!

This problem is related to the spanning tree one. The graph representing all the paths from one vertex to all the others must be a spanning tree - it must include all vertices. There will also be no cycles as a cycle would define more than one path from the selected vertex to at least one other vertex. For a graph,

G = **(V,E)** where

- **V** is a set of vertices and
- **E** is a set of edges.

Dijkstra's algorithm keeps two sets of vertices:

S	the set of vertices whose shortest paths from the source have already been determined *and*
V-S	the remaining vertices.

The other data structures needed are:

d	array of best estimates of shortest path to each vertex
pi	an array of predecessors for each vertex

The basic mode of operation is:

1. Initialise **d** and **pi**,
2. Set **S** to empty,
3. While there are still vertices in **V-S**,
   i. Sort the vertices in **V-S** according to the current best estimate of their distance from the source,
   ii. Add **u**, the closest vertex in **V-S**, to **S**,
   iii. **Relax** all the vertices still in **V-S** connected to **u**

Relaxation

The **relaxation** process updates the costs of all the vertices, **v**, connected to a vertex, **u**, if we could improve the best estimate of the shortest path to **v** by including **(u,v)** in the path to **v**.

The relaxation procedure proceeds as follows:

```
initialise_single_source( Graph g, Node s )
   for each vertex v in Vertices( g )
       g.d[v] := infinity
       g.pi[v] := nil
   g.d[s] := 0;
```

This sets up the graph so that each node has no predecessor (**pi[v] = nil**) and the estimates of the cost (distance) of each node from the source (**d[v]**) are infinite, except for the source node itself (**d[s] = 0**).

Note that we have also introduced a further way to store a graph (or part of a graph - as this structure can only store a spanning tree), the **predecessor sub-graph** - the list of predecessors of each node,
pi[j], 1 <= j <= |V|

The edges in the predecessor sub-graph are **(pi[v],v)**.

The relaxation procedure checks whether the current best estimate of the shortest distance to **v** (**d[v]**) can be improved by going through **u** (*i.e.* by making **u** the predecessor of **v**):

```
relax( Node u, Node v, double w[][] )
    if d[v] > d[u] + w[u,v] then
       d[v] := d[u] + w[u,v]
       pi[v] := u
```

The algorithm itself is now:

```
shortest_paths( Graph g, Node s )
    initialise_single_source( g, s )
    S := { 0 }          /* Make S empty */
    Q := Vertices( g ) /* Put the vertices in a PQ */
    while not Empty(Q)
        u := ExtractCheapest( Q );
        AddNode( S, u ); /* Add u to S */
        for each vertex v in Adjacent( u )
            relax( u, v, w )
```

Operation of Dijkstra's algorithm

As usual, proof of a greedy algorithm is the trickiest part.

Animation

In this animation, a number of cases have been selected to show all aspects of the operation of Dijkstra's algorithm. Start by selecting the data set (or you can just work through the first one - which appears by default). Then select either step or run to execute the algorithm. Note that it starts by assigning a weight of infinity to all nodes, and then selecting a source and assigning a weight of zero to it. As nodes are added to the set for which shortest paths are known, their colour is changed to red. When a node is selected, the weights of its neighbours are relaxed .. nodes turn green and flash as they are being relaxed. Once all nodes are relaxed, their predecessors are updated, arcs are turned green when this happens. The cycle of selection, weight relaxation and predecessor update repeats itself until all the shortest path to all nodes has been found.