Algorithm Analysis

When discussing whether a given algorithm is "efficient," we must decide whether we mean space efficiency or time efficiency, or both. These attributes often present trade-offs between various implementations; given an algorithm, we may be able to find another algorithm for the same task that improves upon the time efficiency but at the expense of space efficiency, or vice versa.

By space efficiency, we mean the amount of memory consumed over and above the original data. Does the algorithm require just a few extra variables to process this data, or does it, for example, recursively copy potentially large arrays over and over? By time efficiency, it would be reasonable to assume that we mean the wall-clock time to execute an algorithm. The problem with running your stopwatch while the program executes is that there are too many variables somewhat independent of the algorithm itself: the speed of the machine, the size of its internal memory, the programming language used, the skill of the programmer, etc. Consequently we will generally concentrate on identifying a more fundamental "unit of work" that the algorithm performs, regardless of these other circumstances, and then counting how many times that unit of work is done.

The fundamental unit of work in a search algorithm is comparison of the target value against values from the list. Regardless of the algorithm details, and no matter how clever someone may be in finding a new search algorithm, it is hard to imagine how such comparisons could be avoided. In analyzing our four search algorithms (Sequential Search, Short Sequential Search, Binary1 and Binary2), we will count the number of comparisons done in searching a list for a target value, and use this to compare the relative efficiency of the four algorithms.

It is true that each of these algorithms also does housekeeping tasks such as incrementing an array index or computing the midpoint of an array, but these operations at worst just result in some small constant multiple to the number of comparisons. For example, if we have to change one array index for every comparison, then if there are n comparisons done, there will be 2n comparisons plus index manipulations done. We'll seldom be concerned with this constant multiple. If algorithm A takes 10 steps and algorithm B takes 100 steps, or A takes c*10 steps, where c is some constant, and B takes c*100 steps, it is still true that B does ten times the work in either case.

Once we have decided on the unit of work, there are still two ways we can analyze the algorithm's efficiency.

Worst case	What kind of input will make the algorithm work hardest? How much work will it do in this case?
Average case	What constitutes "typical" input for this algorithm? How much work will it do in this case?

A worst-case analysis is almost always easier to do, for two reasons:

1. It's very hard to identify what "typical input" is.
2. The mathematics of the analysis to compute the amount of work done in the average case is usually more difficult than in the worst case.

Analysis

Hashing is a search method. As in all our other (internal) search algorithms, the unit of work is comparisons of the target value against a key value. We want the average amount of work for successful and unsuccessful searches.

Assume, then, that the hash table is already built, and we are now searching for a target. If collisions never occurred, the work would always be 1 comparison, a perfect situation. But of course, collisions do occur. Because chaining and open addressing handle collision resolution quite differently, we will do two separate analyses.

In either case, however, the likelihood of a collision is going to depend on how full the hash table is. Some notation:

n = the number of items stored in the hash table

(this is consistent with the previous use of n to denote the number of items in the set being searched - n items in a list, n items in a tree, etc.)

t = hash table size (size of the array)

l = n/t

l is the **load factor** and is an indication of how full the hash table is. For open addressing, this ratio is easy to understand. If t = 10 and n = 4, then 4 of the 10 slots in the array are filled, and the load factor is 0.4. It is clear for open addressing that

n £ t (you can't have more keys than there are array slots)

so

l £ 1.

For chaining, although there are t linked lists, each list can be arbitrarily long, so it is possible to have n > t and therefore l > 1.

Chaining

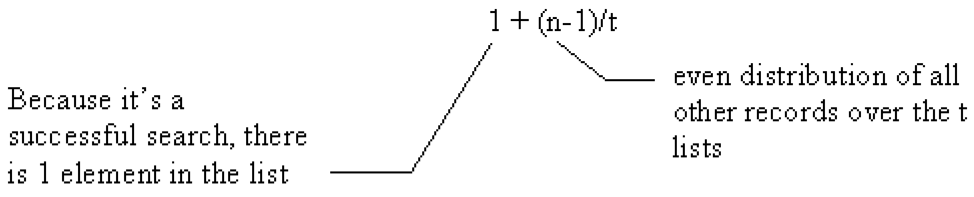
Assumptions:

1. Items are inserted at the head of the list, so the lists are unordered. Consequently, a regular sequential search must be used to search the list. Because there are t lists, the hope is that each is relatively short - see the remark after assumption #2 - so there shouldn't be much penalty for using regular sequential search.
2. Data in the hash table have been uniformly distributed among all the lists. Note: this is a BIG assumption. Unlike assumption #1, which we can make happen by the way we write the code, we have no guarantees of uniform distribution. We're using an approximation in order to make the mathematics tractable.

Assumption #2 says that each list is the same length, namely that each list is approximately n/t = l in length. (If 20 items are stored evenly in 4 lists, there are 5 items per list.)

Unsuccessful search: The target gets hashed to an array index, hence to a linked list that must be searched for the target. This is an unsuccessful sequential search on a list of length l . By our results on sequential search, this will require l compares. (Our analysis of sequential search assumed that the elements were stored in an array and we had to move from index to index in the array, but the same number of comparisons are done if you traverse from node to node along a linked list.)

Successful search: The target gets hashed to an array index, hence to a linked list that must be searched for the target. However, since this is a successful search, the list contains at least one element, namely the target. We can approximate the length of the list as follows:


1 + (n-1)/t
Because it's a successful search, there is 1 element in the list
even distribution of all other records over the t lists

Here the 1 is important because l could be quite small, so 1 is significant. We are doing a sequential search. From our earlier results, the average work for a successful search using sequential search is

(k + 1)/2

where k is the number of elements. So here the work is

$$\frac{(1+\frac{n-1}{t})+1}{2} = \frac{2+\frac{n-1}{t}}{2} = 1+\frac{n-1}{2t} \approx 1+\frac{n}{2t} = 1+\frac{\lambda}{2}$$

This is the 1 comparison we must make when we find the target, plus about half the list.

Open Addressing

Assumption:

1. We will equate the amount of work with the number of probes into the hash table (this includes the initial probe at index h(K) supplied by the hash function). This is an approximation because when an empty cell is encountered (terminating an unsuccessful search), this constitutes a probe, but not a comparison of the target against a key value.
2. All probes into the hash table are independent events. This means that where one element got inserted into the table had no effect on where others got inserted, and it also ignores the details of the collision resolution scheme used for probing. Basically, we are assuming that each time the table is probed is just some random access independent of a previous probe. Note that this is also a BIG assumption, again to make the mathematics tractable, and the answer will be an approximation.

Unsuccessful search: We must probe until we find an empty cell. The percentage of full cells is l , so the percentage of empty cells is 1 - l . The number of cells we expect to probe to hit an empty cell is

$$\frac{1}{1-\lambda}$$

(i.e., if there are 4/20 cells empty, 4/20 = 1/5, then we expect to do ~5 probes to hit an empty cell). This is where assumption #1 is used.

Note that if l = 0, work = 1; the array is empty and the first cell we try is empty, so only 1 probe is required. As l increases toward 1, its maximum value, the denominator decreases and the work increases.

Successful search:

Fact 1: The number of probes for a successful search for a given target equals the number of probes that were required to insert that target (in one case the last probe is where you find the target, in the other case the last probe is where you found the empty cell to insert the target).

Fact 2: When the load factor is l , the number of probes to insert a value into the hash table is the same as for an unsuccessful search for that value - you go until you find an empty cell. Hence, from above, the work is 1/(1 - l).

Even though there are now n elements in the table (l = n/t), the early ones were easy to insert and will be easy to find. We need some sort of average of the work to insert over the range l = 0 to now.

When i keys have been inserted, the load factor is i/t. The work to insert key i + 1 is the work to do an unsuccessful search for that key, which is

$$\frac{1}{1-\frac{i}{t}}$$

The total work to insert the n keys is therefore

$$\sum_{i=0}^{n-1}\frac{1}{1-\frac{i}{t}}=\sum_{i=0}^{n-1}\frac{t}{t-i}$$

We can approximate the summation using the area under a curve:

$$\int_0^{n-1}\frac{t}{t-x}dx=t\int_0^{n-1}\frac{1}{t-x}dx=-t\ln(t-x)\Big|_0^{n-1}=-t\ln(t-(n-1))+t\ln t$$

$$=t\left[\ln t-\ln(t-n+1)\right]=t\left[\ln\frac{t}{t-n+1}\right]\approx t\ln\frac{t}{t-n}=t\ln\frac{1}{1-(n/t)}=t\ln\frac{1}{1-\lambda}$$

Each of the n inserted keys has probability 1/n of being the target key, so the average work is

$$\frac{t}{n}\ln\frac{1}{1-\lambda}=\frac{1}{\lambda}\ln\frac{1}{1-\lambda}$$

Note that this value is less than the average unsuccessful search - you might find your target right away.

Summary:

	Unsuccess	Success
Chaining	» l	» 1 + l /2
Open addressing		

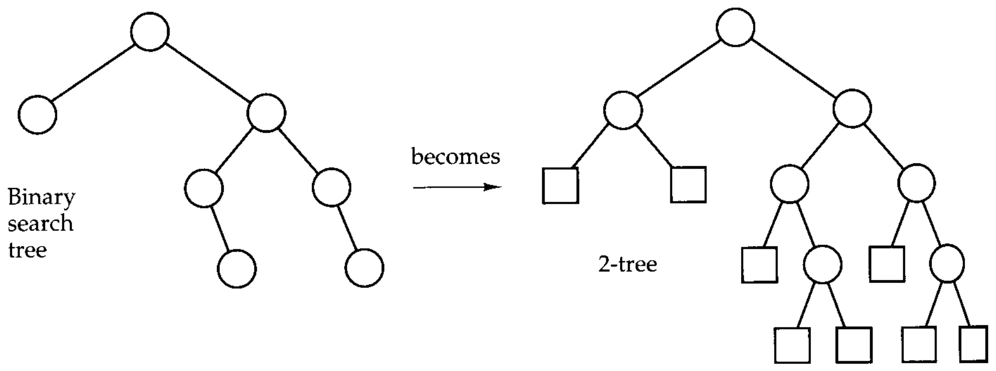
	$\approx \frac{1}{1-\lambda}$	$\approx \frac{1}{\lambda}\ln\frac{1}{1-\lambda}$

Note that, unlike all our previous search methods, the work required for hashing depends on the load factor, not on the absolute size of n. More records can be searched just as efficiently as fewer records provided the hash table size grows accordingly.

Average BST

Is there any estimate we can make for the AVERAGE work to search an AVERAGE binary search tree (one where we make no assumptions about the shape)?

Recall that in doing a binary tree search, unsuccessful searches occur when a null pointer is encountered, i.e., the null pointer indicates that the path you would have followed to find this node terminates, so you know the node isn't there. Step 1 in our analysis is to replace (on paper) any empty subtree of a BST (what would be a null pointer in the implementation) with square leaves.



As a result:

- the tree is a 2-tree (all nodes are leaves or have two children)
- two comparisons are done at each interior node except 1 comparison at a success node
- all successful searches will terminate at interior nodes
- all unsuccessful cases will terminate at leaves

These are also characteristics of the decision tree for the Binary2 search algorithm, but this time we assume nothing about the shape of the tree.

Assume there are n data elements, which will be the n internal nodes in the 2-tree. Let

S(n) = the average number of comparisons for successful search
U(n) = the average number of comparisons in unsuccessful search
E = external path length (sum of path lengths to all leaves)
I = internal path length (sum of path lengths to all internal vertices)

Here S(n) and U(n) are just notations, we don't yet have a formula for these expressions.

Then

$$U(n) = \frac{2 * E}{n+1}$$

because failures occur at the leaves. The total path length to all leaves is E, with 2 comparisons at each node along the way, so 2*E total comparisons. There are n + 1 failure modes, so divide by n + 1 to get the average. Rewriting this equation gives

$$E = \frac{U(n)(n+1)}{2} \quad (1)$$

Also

$$S(n) = \frac{2*I}{n} + 1$$

because success occurs at interior nodes. The total path length TO all interior nodes is I, with 2 comparisons at each node along the way. There are n successes, so divide by n to get the average. But every node has one final comparison for success.

From earlier work on 2-trees, we know E = I + 2n. So I = E - 2n. Substituting in the formula for S(n), we get

$$S(n) = \frac{2*I}{n} + 1 = \frac{2(E-2n)}{n} + 1 = \text{(using Equation(1))}\ \frac{2\left(\frac{U(n)(n+1)}{2} - 2n\right)}{\text{n}} + 1$$
$$= \frac{U(n)(n+1) - 4n}{n} + 1 = \frac{U(n)(n+1) - 4n + n}{n} = \frac{U(n)(n+1) - 3n}{n}$$

or

$$nS(n) = (n+1)U(n) - 3n \qquad (2)$$

Since S(n) is the average number of comparisons for successful search over all n data elements, we'll use the usual formula for average work

$$\text{Average work} = \sum_{\text{all cases}} (\text{work for this case}) * (\text{probability of this case})$$

As usual, we are assuming that each key value is equally likely to be in the tree, so the probability for each case is 1/n.

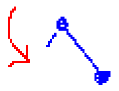
Then

$$S(n) = \frac{\text{successnode}(1) + \text{successnode}(2) + \ldots + \text{successnode}(\text{n})}{\text{n}}$$

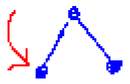
where

- success node (1) means the work to search for the first node that was inserted in the tree
- success node (2) means the work to search for the second node that was inserted in the tree
- etc.

Consider success node (i). A successful search for the key value stored at the ith node inserted into the tree takes 1 more comparison than the number of comparisons needed to do an unsuccessful search of the tree for this key value just before node i was inserted. The unsuccessful search comes to the place where the node would be and says " pointer = NULL, not found"; this does not count as a comparison.



The successful search comes to the same point and does a comparison for equality against that node's key value.



We can estimate the amount of work for that unsuccessful search by using the average work to do an unsuccessful search on a tree with i - 1 elements, which, according to our notation, would be U(i - 1).

So success search (i) = 1 + U(i - 1).

Therefore

$$S(n) = \frac{\text{success node}(1) + \text{success node }(2) + \ldots + \text{success node }(n)}{\text{n}}$$
$$= \frac{(1+U(0)) + (1+U(1)) + \ldots + (1+U(n-1))}{n}$$
$$= \frac{n + (U(0) + U(1) + \ldots + U(n-1))}{n}$$

and therefore

$$nS(n) = n + U(0) + U(1) + \ldots + U(n-1)$$

Combining this with our previous equation (2),

$$nS(n) = (n+1)U(n) - 3n \qquad (2)$$

we get

$$(n+1)U(n) - 3n = n + U(0) + U(1) + \ldots + U(n-1)$$

or

$$(n+1)U(n) = 4n + U(0) + U(1) + \ldots + U(n-1) \qquad (3)$$

Equation (3) holds for any n, so we can rewrite it using n - 1 instead of n.

$$nU(n-1) = 4(n-1) + U(0) + U(1) + \ldots + U(n-2) \qquad (4)$$

Now subtract equation (4) from equation (3) - most terms on the right will cancel out:

$$(n+1)U(n) - nU(n-1) = 4 + U(n-1)$$

or

$$(n+1)U(n) = 4 + (n+1)U(n-1)$$

$$U(n) = \frac{4}{n+1} + U(n-1)$$

$$U(n) = U(n-1) + \frac{4}{n+1}$$

Aha, what have we here? A linear, first-order recurrence relation with constant coefficients. Sound familiar from discrete math? The equation we have is of the form

$$U(n) = c\,U(n-1) + g(n)$$

where c = 1 and g(n) = 4/(n + 1).

We have a formula to solve such recurrence relations in general - see Recurrence Relations under the posted Mathematics Background pages. The only change we need to make is that the base case of our recurrence relation occurs when n = 0. We know how many comparisons it take to do an unsuccessful search of an empty tree - none!

The general form of the solution is therefore

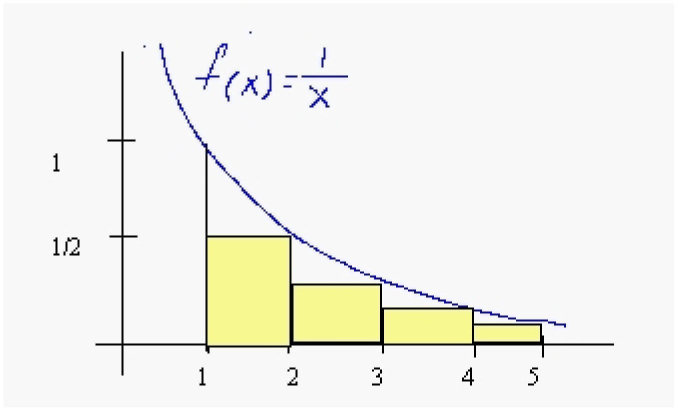
$$U(n) = c^n U(0) + \sum_{i=1}^{n} c^{n-i} g(i)$$
$$= 0 + \sum_{i=1}^{n} \frac{4}{i+1}$$
$$= 4\left[\frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n+1}\right]$$

The series

$$\frac{1}{2} + \frac{1}{3} + \ldots$$

is a **divergent infinite series**, meaning it does not approach a finite limit. But we only need the sum of a finite number of terms of this series. We can approximate it by using calculus to find the area under a curve.

Consider the function f(x) = 1/x. Its graph is shown below.



The area beneath this curve from 1 to 5 is an approximation (a little too big) of the area represented by the colored blocks,

$$\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5}$$

Similarly,

$$\int_1^{n+1} \frac{1}{x}\,dx$$

is an approximation to the sum

$$\left[\frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n+1}\right]$$

So

$$U(n) = 4\left[\frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n+1}\right] \approx 4\int_{1-1}^{n+1} \frac{1}{x}dx = 4\ln\, x\Big|_1^{n+1}$$
$$= 4\ln(n+1) - 0 = 4\ln(n+1)$$

For large n, we can approximate this value by 4 ln n.

Going back to Equation (2) above,

$$nS(n) = (n+1)U(n) - 3n$$

we again assume that n + 1 ~ n, so that

$$nS(n) \approx nU(n) - 3n$$

or, dividing by n and then dropping the 3,

$$S(n) \approx U(n) - 3 \approx U(n)$$

Finally, we have an approximation for the average work to search for a key value in an n-element tree:

$$S(n) \approx 4\ln n$$

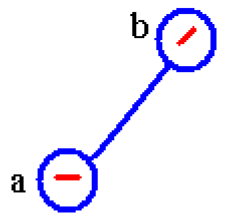
In an ideally-shaped tree (like the decision tree for Binary 2), the average work would be about 2 lg n. Note the different bases for the logarithms. To perform a change of base,

$$4\ln n = 4(\ln 2)(\lg n) = (2\ln 2)(2\lg n) \approx 1.39(2\lg n)$$

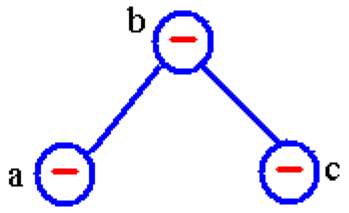
Final conclusion: the average work to search for an element in an n-element **average** BST is about 1.39 times the work to search for an element in an ideally-shaped BST.

AVL Tree Insertions

The basic idea of inserting a node into an AVL tree works just as it did for a BST. If you get to a NULL pointer, put the new node there; if your node is already there, return a "duplicate error" message. If your node is smaller, go left, if it's larger, go right. However, an additional signal called *taller* is maintained (in the implementation, this will be a boolean variable). When you reach a NULL pointer and insert a new node, *taller* becomes true - you turned an empty subtree into a (taller) subtree with one node. So you send this *taller* signal back out from this invocation of the recursive insertion function. At the next level up in the tree, however, that is at the previous invocation of the recursive insertion function, *taller* may become false. For example, consider the AVL tree



in which node b is left_higher. Suppose c is the next node to be inserted.

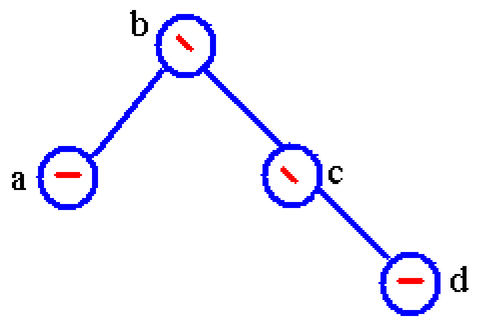


In the new tree, the invocation of the recursive insertion function that inserted c sent a *taller* signal from the (formerly empty) right subtree back up to node b, but node b, upon receiving that signal, merely resets its balance factor to equal_height. If b is the root of a subtree of some larger tree, the overall height of the subtree rooted at b has not changed, and the *taller* signal would be set to false, i.e., not propagated further back up the tree.

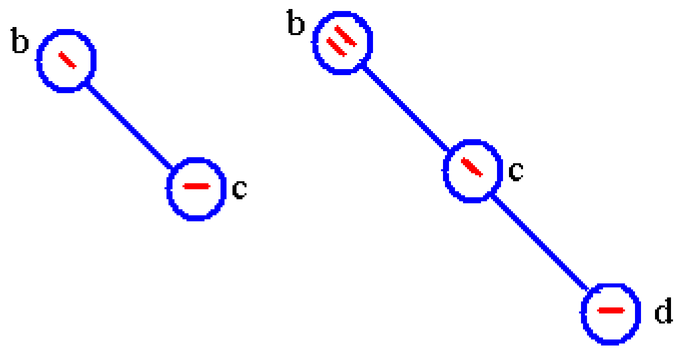
Here are the three cases for insertion into the right subtree of a node (the situation above where we inserted into b's right subtree).

Insert into the right subtree and check the taller signal. If taller is true, look at the balance factor at the root.

- Case 1: If the balance factor was left_higher, set it to equal_height and stop the taller signal by setting it to false [the case pictured above]
- Case 2: If the balance factor was equal, set it to right_higher [don't reset the *taller* signal, let it propagate up the tree because the subtree is taller than it was before] This happens both at node c and then at node b below when d is inserted.



- Case 3: If the balance factor was already right_higher, then something must be done. This is the case below, where node c goes from equal_height to right_higher, which is OK, and propagates the *taller* signal back up to b, which was already right_higher. Some rebalancing must be done at node b.

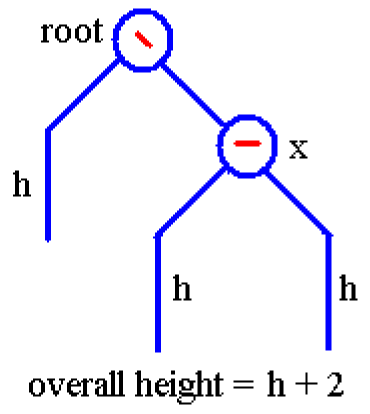


A similar set of three cases holds for insertion into a left subtree.

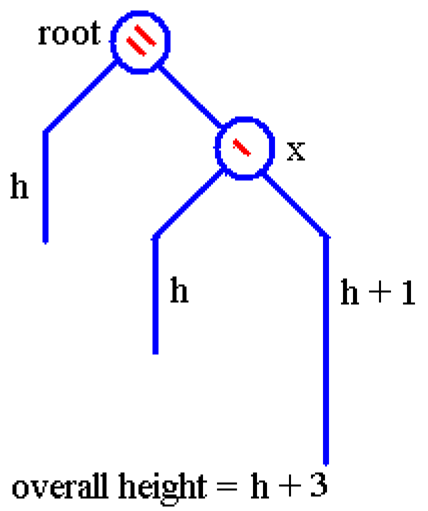
Now, how to rebalance? Again, let's just consider insertion into the right subtree; balancing after insertion into the left subtree is similar.

Originally, the root was right_higher because the right subtree had height h + 1 and the left subtree had height h. Now we insert into the right subtree, giving it height h +2. The root at which we need to rebalance is now doubly right_higher. The key to how to rebalance is to look at the balance factor at the root of the right subtree, the node that just sent the *taller* signal back up to the root.

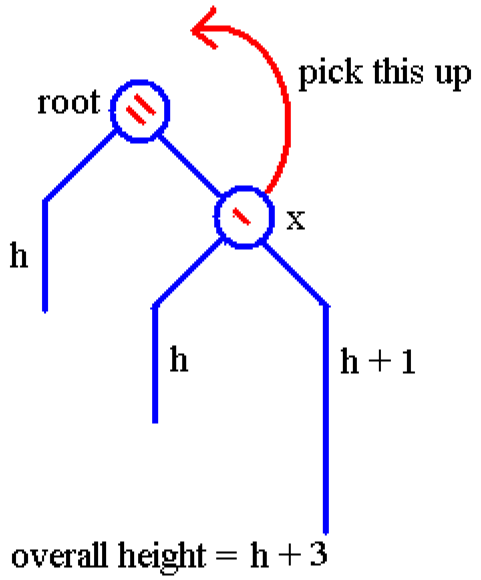
SubCase 1: Right_Higher. Here's the original picture. The right subtree root, x, is balanced, but we are about to insert into x's right subtree, which will make x right_higher.

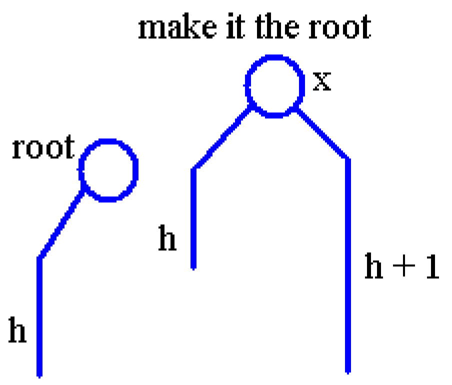


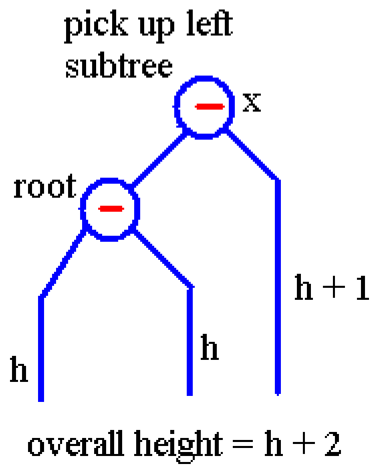
After insertion into x's right subtree:



Solution: Perform a "left rotation" around the root node that's now out of balance. Grab the subroot x and lift it up to become the new root; let the original root drop down to become the root of the left subtree of x. The original root keeps its original left subtree, and also picks up x's left subtree as its right subtree.







Why does this work? Node x was in the root's right subtree so root < x. Now root is in x's left subtree, which agrees with root < x. Also, the subtree that we switched from being x's left subtree to root's right subtree contains values between root and x. After the switch, they are still in the correct place for being between root and x.

This is all accomplished (in a slightly different order) in the code below with a couple of pointer redirections. The only difference is that where we have talked about "root" above, this code uses "sub_root" because it can be called at any node in the tree.

```
//make a temporary pointer that points to x, the root of the
//right subtree of the node where the problem occurs

Binary_node<Entry> *right_tree = sub_root->right;

//redirect the right pointer from the node where the
//problem occurs to point to the left subtree
//of x. That's the last step we did above.
// Note that it's safe to redirect this pointer
// because we've first made a temporary pointer to x

sub_root->right = right_tree->left;

//redirect the left pointer from x to the original root

right_tree->left = sub_root;
```

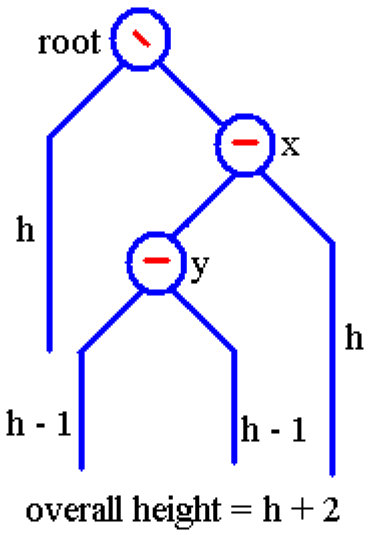
```
//take the original pointer and point to x,
//thereby making it the root of the subtree

sub_root = right_tree;
```

Of course, there is a similar situation for a right rotation.

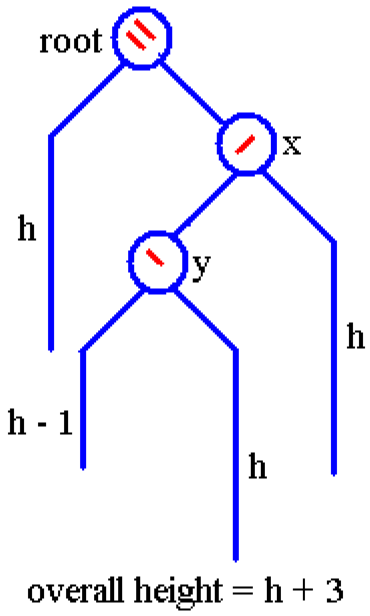
OK, this was the simple case!

SubCase 2: Left Higher

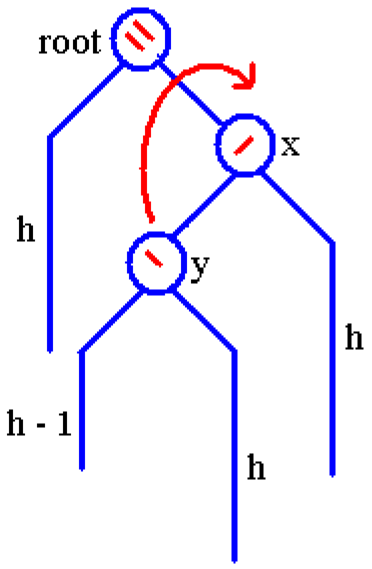
Here's the original picture. The right subtree root, x, is balanced, but we are about to insert into x's left subtree, which will make x left_higher. (Actually, we don't know - or need to know - whether y is equal_height at this point.)



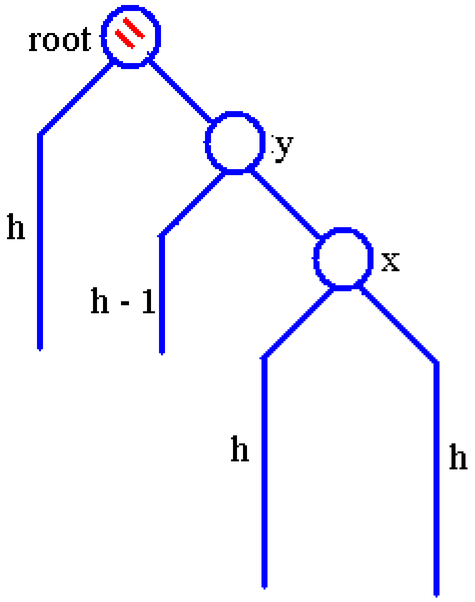
After insertion into x's left subtree:



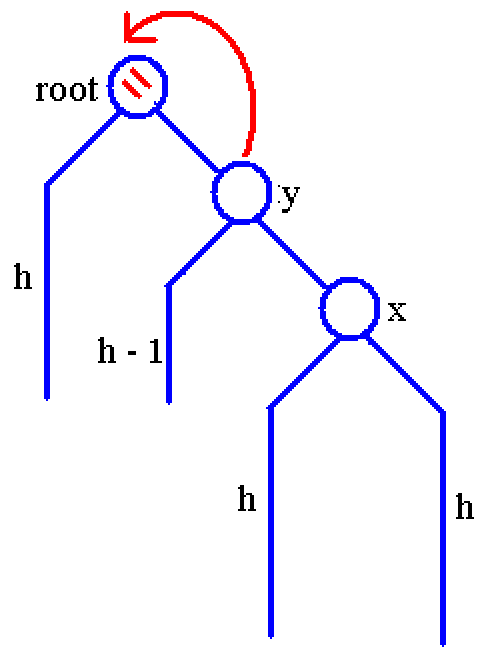
Solution: this requires two rotations. First, pick up node y and do a right rotation around node x.



Node y will be promoted to x's old position, x will fall to be y's right child, and will pick up y's right subtree as its left subtree, so the result is

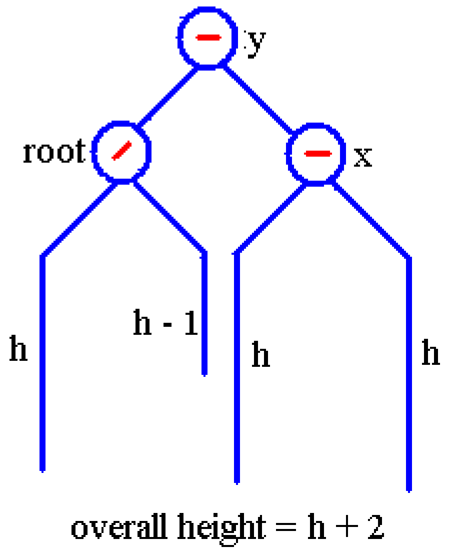


Now do a left rotation of y around the root.



Node y becomes the new root, the original root becomes y's left child and picks up y's left subtree as its right subtree.

Here's the result:



SubCase 3: Equal_height. After an insertion into the right subtree of the root, that is, into the subtree rooted at x, can we end up with x having equal height at that point? The insertion went into one of x's subtrees. If x has equal height after the insertion, then the *taller* signal would not have been propagated up to the root, and the root would not now be out of balance.

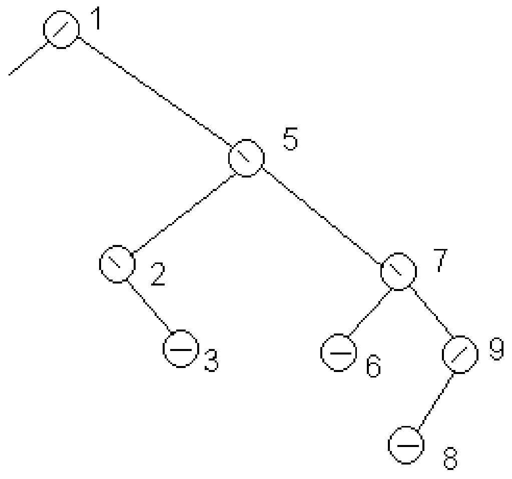
The code rebalances at the first node from the leaf back to the root at which a problem occurs. In the case of insertions, rebalancing always shortens the tree so that the "taller" flag is stopped. This means that any node above the problem node that may also have been out of balance will be OK once the rebalancing is done - no further rebalancing farther up the tree is necessary.

Click here for an animation of building an AVL tree.

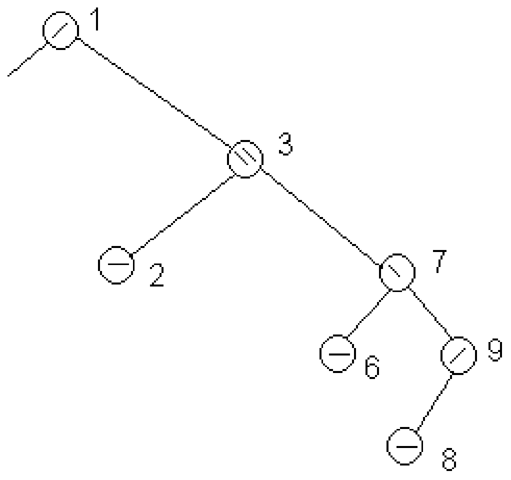
AVL Tree Removals

Removals from an AVL tree can also require rebalancing to maintain the AVL property. The removal function uses a boolean flag of *shorter* that propagates up the tree until balance is restored. The same rotations as in insertion are used, but it's a bit more complicated as to when to use them. What is somewhat confusing is that a removal is not precisely the opposite of an insertion, that is, you can insert a node into an AVL tree, then immediately remove it, and you won't get back to the original tree.

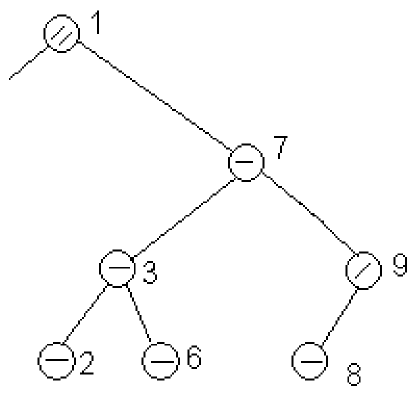
Just as in the case of removals, rebalancing occurs at the first problem node from the newly-inserted leaf back up to the root. However, rebalancing at this node may not stop the "shorter" flag, and nodes farther up in the tree may still have a problem. Below is part of a tree where the root, node 1, is left_higher. We want to remove node 5.



Because node 5 has two children, we follow the BST removal algorithm and replace node 5 with its immediate predecessor node 3, then remove node 3, giving



Node 3 is out of balance and will be fixed by a single left rotation of node 7 over node 3, giving



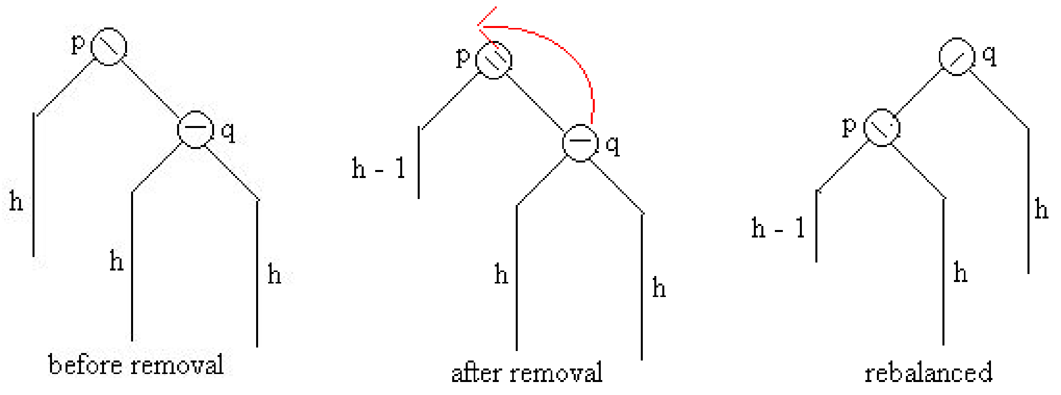
Because the right subtree of 1 has now been shortened, the "shorter" signal is propagated up to node 1, which is now out of balance.

So what exactly is the removal algorithm?

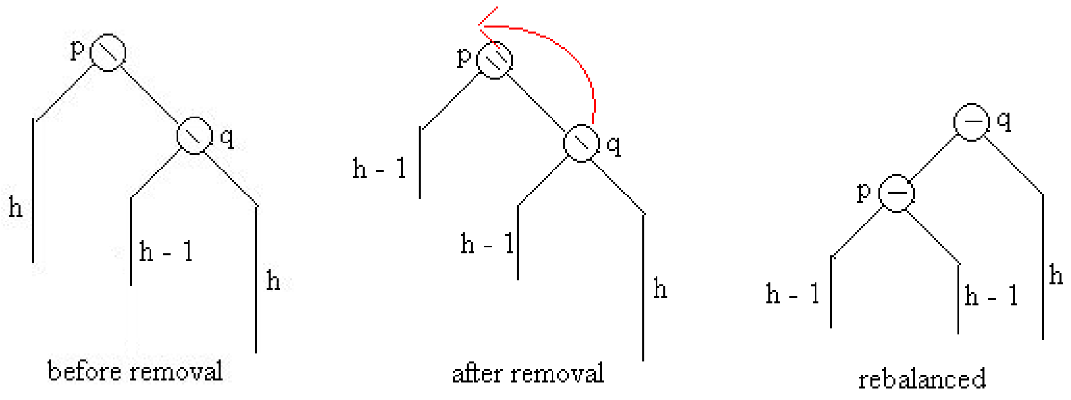
Begin by following the BST removal process. If the node x to be removed is a leaf, set the parent pointer to that node to NULL. If x has a single child, connect the parent pointer to that child. If x has two children, find the immediate predecessor of x and put it in place of x, then remove the immediate predecessor (which has only one child). In any of these actions, the subtree rooted at the removed node has been shortened, so set the boolean *shorter* signal to true. This signal first reaches the parent node of the removed node, but as we saw in the above example, the *shorter* signal can propagate all along the path from the parent back to the root. So consider any node p along this path that has just received a true *shorter* signal. (Once the *shorter* signal is false, the process of removing the node is complete.) As with insertions, the key is the balance factor at p, and there are several cases.

- Case 1: If the balance factor at p is equal, then set it to left higher or right higher, depending on whether the removal occurred in the right or left subtree, respectively, and stop the *shorter* signal by setting it to false.
- Case 2: If the balance factor at p is not equal and the removal occurred in the taller subtree, then set p's balance factor to equal. But the tree rooted at p has become shorter so do not reset the *shorter* signal, let it propagate to p's parent node.
- Case 3: The balance factor at p is not equal and the removal occurred in the shorter subtree. Node p is now double-weighted and some rebalancing needs to be done. As was true with insertions, what to do depends on the balance factor of a child q of p, but this time q is the root of the tree where the activity (removal) did NOT take place.

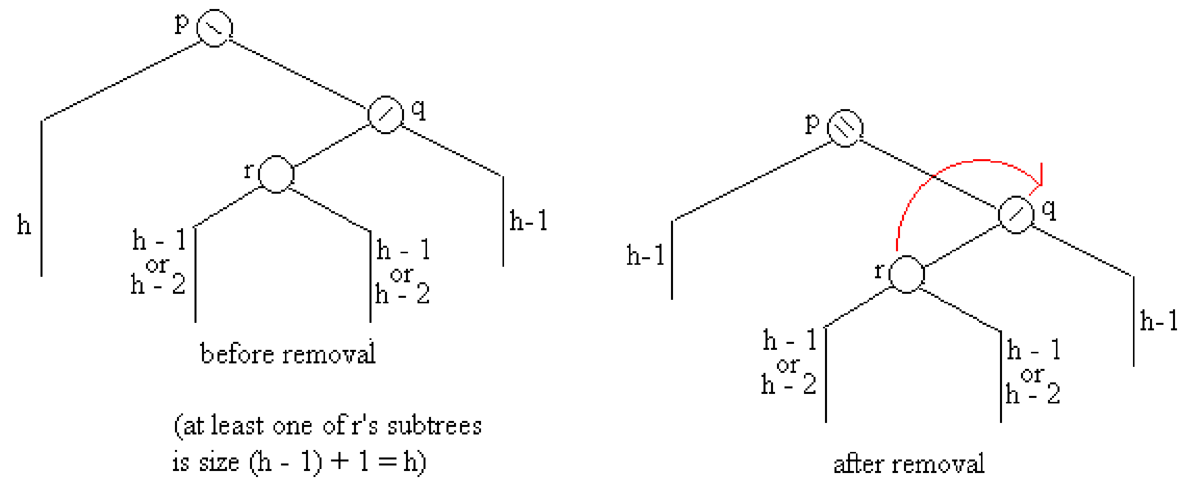
  Subcase 1: q has balance factor of equal. Do a single rotation (left or right, depending on whether q is the right or left child of p) and adjust the balance factors of p and q. All nodes are now OK and the overall height of the tree has not been changed, so set the *shorter* signal to false.

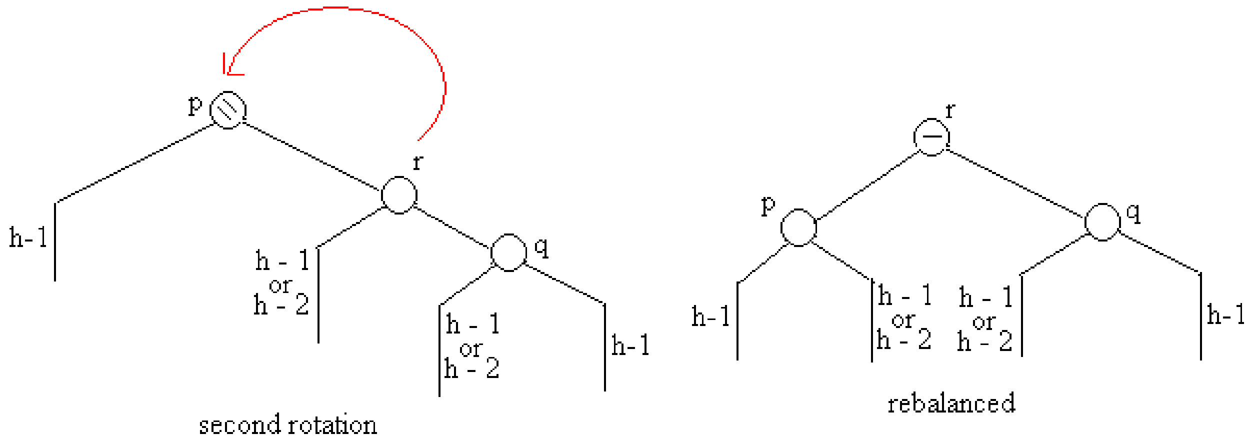


Subcase 2: q leans in the same direction as p. As in Subcase 1, a single rotation of q over p, with adjustments to the balance factors of p and q, will solve the rebalancing problem, but in this case the overall tree height has been shortened, so leave the *shorter* signal as true.



Subcase 3: q leans in the opposite direction from p. A double rotation is required. Take the root r of q's taller subtree and rotate r first over q and then over p. Adjust the balance factors of p and q; r will have equal balance. The overall tree height has been shortened, so leave the *shorter* signal as true.





Summary: Now we can insert nodes into an AVL tree and remove nodes from an AVL tree while maintaining the AVL property. Searching an AVL tree is just the same as searching a BST, nothing new happens because it's an AVL tree. But the AVL tree should be a nearly perfectly balanced BST, so the work to perform a search should be optimal.

Finally, here's a link to an online applet that lets you play with an AVL tree:

http://www.qmatica.com/DataStructures/Trees/AVL/AVLTree.html

AVL Trees

As we have seen, the ideally-shaped BST may save us 39% in the work (number of comparisons) to search for an element (either successfully or unsuccessfully) over that of a randomly-generated BST. We also know that the shape of the tree, which affects the work to search, is determined by the order in which data elements are inserted into the tree.

One approach is to read the data to be searched into an array, not a tree, then sort the array in increasing order. Of course we know that if we used these elements in order to build a BST we would get the worst case, a BST that is basically a chain. But we can be more clever about how we build the tree. Pick the midpoint element of the array as the root, the midpoint of the left half of the array as the root's left child, the midpoint of the right half of the array as the root's right child, the midpoint of the first quarter of the array as the left child of the left child, etc. The result will be a nearly perfect BST. As just described, the tree would be built in level order. BUT - sorting is expensive. Another option is to build a BST from the original data, read it into an array using in-order traversal, which results in a sorted array without having to use a sorting algorithm, then use the above approach. BUT - still pretty wasteful because you have had to build two entire BST's plus use array storage.

Instead of gathering all the data and then creating a balanced BST, we can build such a tree as we go by modifying the algorithm for insertions into a BST so that after each such operation, if the tree gets out of whack, it can be corrected then and there. The resulting tree is nearly perfectly balanced. Likewise, when we do deletions from a BST we make corrections if the deletion destroys the balance. These "nearly perfectly balanced search trees" are called AVL trees in honor of the two Russian mathematicians who devised these algorithms in 1962.

An **AVL tree** (also called a **height-balanced tree**) is a binary search tree where

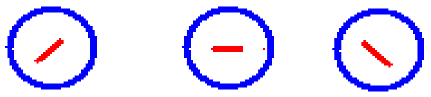
1. the heights of the left and right subtrees of the root differ by at most 1
2. the left and right subtrees are again AVL trees

Once again we have a recursive definition, so that for each and every node of the tree, the left and right subtrees of that node will differ in height by at most 1.

In order to keep track of the height of the subtrees of a node, each node has a balance factor that always has one of the three values

- -1, left_higher (the left subtree has a larger height than the right subtree)
- 0, equal_height (both left and right subtrees are the same height)
- 1, right_higher (the right subtree has a larger height than the left subtree)

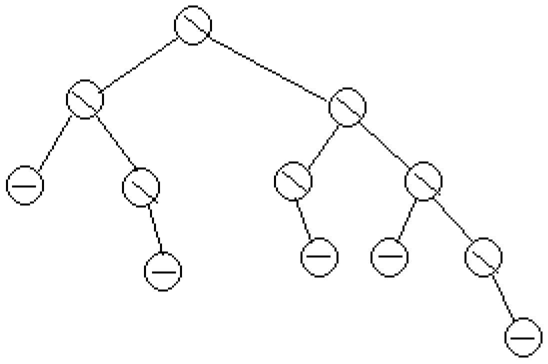
The sketches in these pages use a graphical notation, marking each node as



for left_higher, equal_height, right_higher, respectively.

[Think ahead to implementation. If an AVL tree is a special kind of binary search tree, then we will probably implement it as a derived class of the Search_tree class. This will make the AVL class the "grandchild" of the Binary_tree class. Furthermore, nodes in AVL trees are special kinds of Binary_nodes, because they have this balance factor, so AVL_node will be a derived struct of Binary_node. In a derived struct (or class), you can add additional member variables, so we'll add a fourth member variable to the three already used in Binary_node. It turns out we will have to modify the Binary_node struct a little to make things work.]

Note that the restrictions of an AVL tree do not quite guarantee the "ideally-shaped BST" we might have wished. Consider the following, which is an AVL tree but yet has leaves on three different levels. If you continue this process, you can get a tree where the worst case (longest branch from root to leaf) is pretty bad.



However, experiments indicate that the "average" AVL tree is much closer to the best case. In other words, AVL trees, while not the perfect BST we might like, are a big improvement in the work to search over an unbalanced BST.

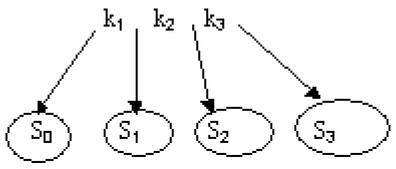
Insertions into and deletions from an AVL tree work just like they do for BSTs, unless an insertion or deletion causes some node to have left and right subtrees with heights differing by more than 1. Then we must make some local readjustments to regain the AVL property before going on to do anything else. This work generally just involves redirecting a couple of pointers, so AVT tree maintenance is not that costly.

B-Tree Structure

From BST to AVL tree, the definition picks up additional properties or constraints, and insertions into or deletions from the tree require more work to be sure the properties are preserved. A B-tree is still more complex, so in order to maintain its structure through insertions and deletions requires even more work. (The structure is sufficiently different from BSTs and AVL trees, however, that implementation is not done as a derived class.)

For the moment, we'll skip the motivation of why we would want a B-tree and just concentrate on the structure, its rules, and the actions needed to maintain the structure while inserting or deleting.

A **B-tree** is an *m-way tree*, an extension of a binary tree. Here m, called the **order** of the tree, is the maximum number of children per node, and m - 1 is the maximum number of key values per node. (For the time being, we are assuming that the data in a node is just a simple key value.) The general picture of a 4-way node, for example, is:



where

all the elements in subtree $S_0 < k_1$
$k_1 <$ all elements in $S_1 < k_2$
$k_2 <$ all elements in $S_2 < k_3$
$k_3 <$ all elements in S_3

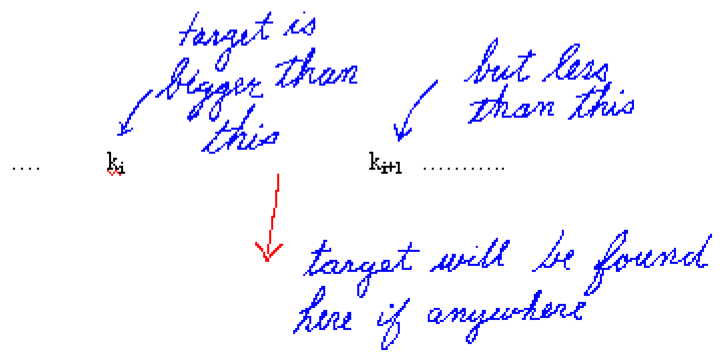
This generalizes the BST tree property, where we had a single value at a node and it was between the values in the left subtree and the values in the right subtree. As a consequence of this part of the definition, the key values at any node must be ordered, that is,

$k_1 < k_2 < \ldots < k_{m-1}$

There are some additional properties that are part of the **B-tree** definition:

1. The root is a leaf or has between 2 and m children, so the root has between 1 and m – 1 key values.
2. All other nodes have between ém/2ù - 1 (minimum) and m – 1 (maximum) key values, meaning that all interior nodes except the root have between ém/2ù and m children.
3. All leaves are at the same level.

Searching a B-tree is also a generalization of the binary tree search in that you follow the correct branch of the tree based on the size of the target compared to the size of key values you have examined. The key values at each node form a little ordered list. When you get to a node, you perform essentially a short sequential search. You travel along the list as long as the target is greater than the list elements. Once that's not true, then you check to see if the current list element equals the target (success); if not, you travel the pointer you just passed to the subtree where the target will be found if indeed it is in the tree at all.


target is bigger than this
but less than this
.... k_i k_{i+1} ....
target will be found here if anywhere

The m-way structure of the tree gives a real speed-up to the search process because with each pointer followed, you eliminate much of the tree as something you won't search, just like you eliminate half the tree in a balanced binary search tree. Putting it another way, an m-way tree is very short and fat, so there are not many nodes as you travel down one branch of the tree. Note, however, that each node along the way requires a short sequential search on perhaps as many as m - 1 elements.

Now, how about B-tree maintenance (insertions and deletions)? Unlike the other trees where we inserted a new node by starting at the root and moving down, growing the tree at the lowest levels, here we insert a new node at a leaf and grow the tree upwards. (We first have to search from the root down to find the leaf where the new key belongs, and do a short sequential search to see where to put it.) Leaves that get too big (an **overflow** condition; remember that m - 1 is the maximum number of key values at any node) split, and their median value moves up a level. This process - splitting an overfull node and promoting the median element up one level - is repeated recursively going all the way back up to the root if necessary. If the root node has to be split, the height of the tree will increase by 1. (Because the root is the "point of growth" for the tree, the root may contain fewer key values than the rest of the nodes, which is why property 1 above is true.)

Because of always promoting the median element in an overfull node, a B-tree will stay relatively balanced, hence close to the ideal shape. This also means that B-trees where m is an odd number are easiest to work with. An overfull node has m elements; if m is odd, the median element is easy to identify.

Example: Consider a B-tree of order 3. Each node has at most 3 children, and at most 3 - 1 = 2 key values. Each node except possibly the root has at least
é3/2ù - 1 = é 1.5ù - 1 = 1 key value.

Click here for an animation of insertions into a B-tree of order 3.

In the insertion process, nodes get too big, they split, and keys get pushed up the tree. For deletions, the opposite happens - nodes get too small, they coalesce and drop down, and the tree shrinks.

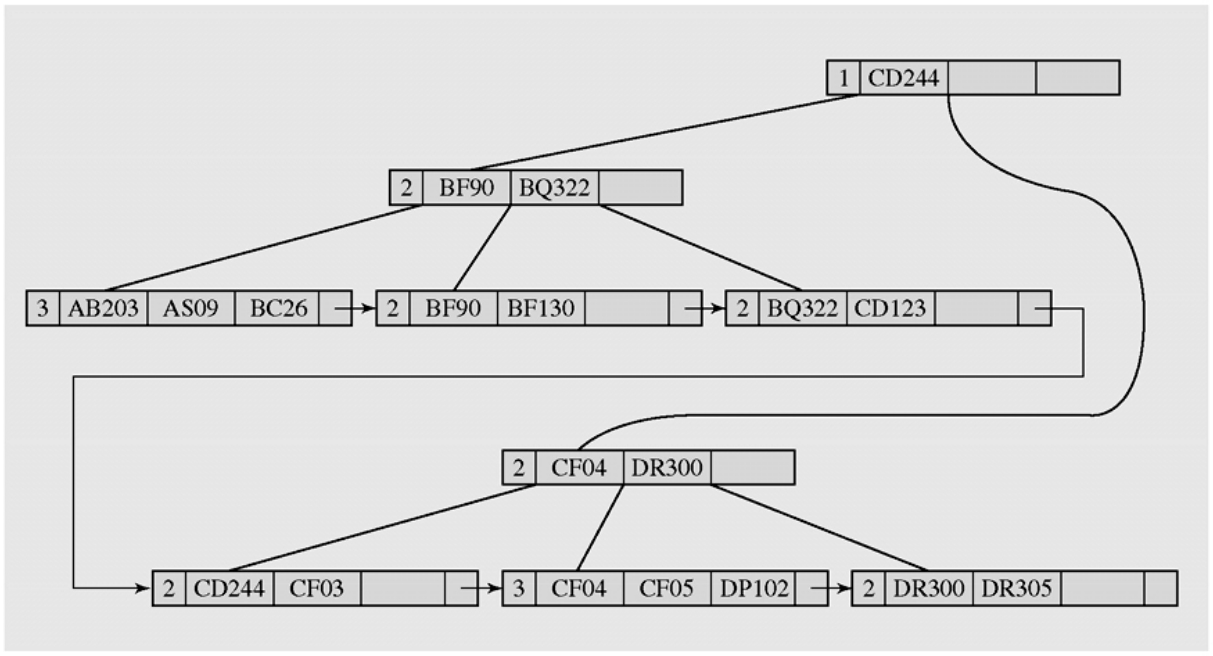
In more detail, consider removing a key value from a tree. We can assume that the key is found (again you have to do a search from the root down) in a leaf node; if not, replace this key value with its immediate predecessor (or successor), which will be found in a leaf, and then remove the leaf element. Suppose a leaf, after a removal, becomes too small (an **underflow** condition). Then, assuming an adjacent sibling node has more than the minimum number of elements, sibling promotes an element to the parent node, and the parent node sends an element down to the underfilled node. Finally, if neither adjacent sibling has elements to spare, then the parent node demotes its median element down and it, together with two siblings, combine to form one new node. If the parent node is now too small, this process is recursively repeated up the tree. If the process reaches the root, the height of the tree will decrease by 1.

Click here for an animation of removals from a B-tree of order 3.

The figure on the next page is used by permission of Cengage Learning from Data Structures and Algorithms in C++ by Adam Drozdek.

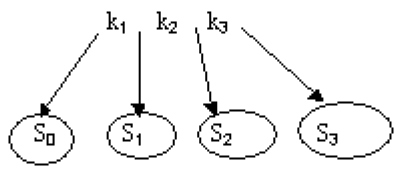
The small integer at the beginning of each block is a counter of the number of key values in that block. Key values are organized alphabetically. The sequence set shows key values only, no data references, that is, the addresses into the actual data file are missing. Note that key values in the index set also occur in the sequence set.

FIGURE 7.12 An example of a B+-tree of order 4.



B+ Trees

The B-tree provides a good solution to minimize disk access for searching, insertion, and deletion operations. However, it does not provide an efficient way to write records in sorted order. We could do a generalization of the in-order traversal used on binary search trees, which would produce output sorted by increasing key value. For example, an in-order traversal of



requires a recursive in-order traversal of subtree S0, followed by reading the k1-k2-k3 block (call this K) to visit k1, followed by a recursive in-order traversal of subtree S1, followed by reading K to visit k2, followed by, ..., etc. So as we walk through the B-tree, each node access is a disk access, and you have to visit the same node many times.

The sorted order output problem can be solved by using a B+ tree. In a B-tree, each node contains both navigation information (keys and pointers) and references to actual data. In a B+ tree, the internal nodes contain navigation information (keys and pointers) but references to the actual data are stored only in the leaves of the tree. The internal nodes form the part of the B+ tree called the **index set** and the leaves form the **sequence set**. The index set is a modified B-tree in that for the picture above, the k_i value is £ values in subtree S_i [as opposed to strictly <] and the values in subtree S_i are < the k_{i+1} value. Each leaf node contains its key-address pairs in sorted order, and the set of leaves is maintained in a linked list. To write the records in sorted order, just walk through the linked list from leaf to leaf and walk down each leaf in turn, which gives data references sorted by increasing key order. (See B+ tree figure in the Code Archives.)

Searching a B+-tree is the exact same operation as for a B-tree: follow the appropriate pointers to trace a path from the root all the way down to a leaf, then walk down the leaf to search for the target key value. If the key value is found at a node of the index set, ignore it because it contains no data reference; keep searching down the next pointer value (which goes to a subtree containing values equal to or greater than this key value). In fact, finding a key value at a node of the index set is not necessarily even an indication of an ultimately successful search - it may be that when you reach the appropriate leaf and search it, your key value isn't there(you'll see shortly how this situation could occur.)

Insertion again takes place at the leaf level. Search for the leaf in which to insert the new value and do a short sequential search to find where to put it. If there is room in the leaf, just insert the new key. If this causes an overflow, then split the leaf and COPY (not move) the median element (minus the data reference) to the parent node. You make a copy because the actual key value has to stay in the sequence set. If there are additional node splits required in the index set, the median node moves up (it's not copied) as in a regular B-tree.

Deletion again takes place at the leaf level. Search for the target key value (in a leaf, not at a node in the index set) and delete it. If this key value is also a value in a node of the index set, it can stay there - it still serves as a navigation aid for the tree. That's how when searching the tree you might find key values in the index set that are not in the sequence set. Compact the leaf node from which you deleted the element; if this causes no underflow, you are done. If this deletion causes an underflow, then grab a key from the sibling node and adjust the parent separator value, or combine the two siblings and remove a parent key value. If there is now an underflow in the parent node (which is in the index set), then continue as in doing a deletion from a regular B-tree.

More on the topics of file organization and disk access minimization later in the semester.

Binary Search Analysis

We found the average amount of work (number of comparisons) for Binary1 and Binary 2 search algorithms, both successful and unsuccessful searches, for n = 10. What we want is a general formula for any value of n. Again, we will make use of decision trees.

Decision trees for both Binary1 and Binary 2 are 2-trees; every internal node has exactly 2 children.

FACT 1: In a 2-tree, the number of vertices at level t is £ 2^t, t ³ 0.
Proof: By induction on t.
Base case: t = 0. The only vertex at level 0 is the root, and 1 £ 2^0
Assume that the number of vertices at level k is £ 2^k
At level k + 1, the maximum number of vertices is 2* (the number at level k) [some vertices at level k may have 2 children, none has more than 2 children. By the inductive hypothesis,

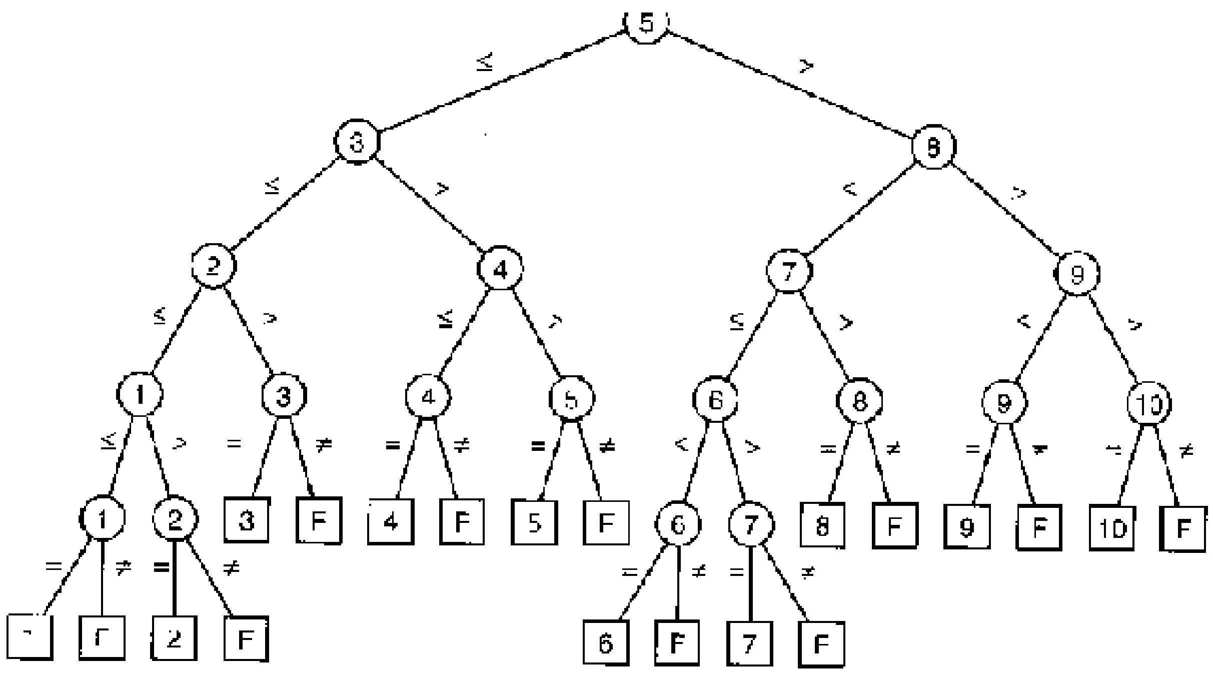
2* (the number at level k) £ $2*2^k = 2^{k+1}$

End of Proof

Furthermore, in these decision trees all the leaves are at most one level apart because of the essential symmetry of the process. The lists being searched keep getting split, and the two halves differ in size by at most one element at each split.

Binary 1

Here's the Binary1 tree for n = 10.



Thinking about how Binary 1 works, the only comparison for equality occurs after the list has been reduced to length 1, and the result is either a success or a failure. So - as we see in the n = 10 tree above, failures and successes are completely symmetric in a Binary1 search, and if there are n elements in the list, the number of leaves in the decision tree will be 2*n. The work for a given leaf is determined by its level, and all these leaves are at roughly the same level, the height of the tree.

Call the tree height h. Then h has to be just big enough to produce 2*n leaves at level h. So we have

2*n £ number of leaves at level h

and (by FACT 1 above)

number of leaves at level h £ 2^h

so

2^h ³ 2*n

or, taking the logarithm to the base 2 of both sides, (logarithms to the base 2 are often written as lg)

lg (2^h) ³ lg(2*n)

and (remember facts about logarithms)

h ³ lg(2) + lg(n) = 1 + lg(n)

Now the smallest integer greater than or equal to 1 + lg(n) is obtained by using the *ceiling function*, so

h = é 1 + lg(n)ù

So for Binary 1, the worst case (in both successful and unsuccessful cases) is

é 1 + lg(n)ù

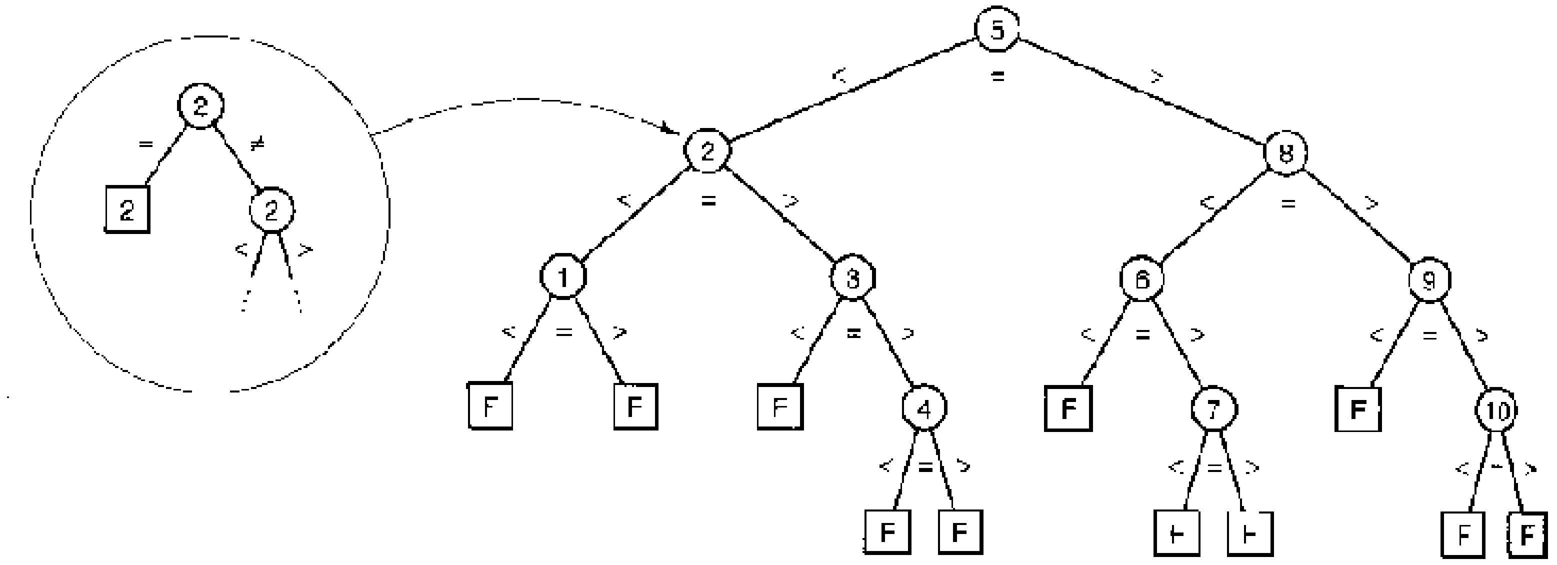
and the average case (remember that most leaves occur at roughly level h) is

~ 1 + lg(n)

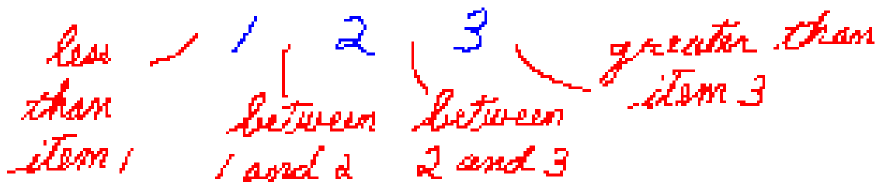
(~ denotes "is approximately equal to")

Binary 2

Here's the Binary 2 tree for n = 10.



First, let's deal with unsuccessful searches. Unsuccessful outcomes are the leaves. If there are n elements in the list, there are n + 1 "slots" where failures can occur. For example, if there are 3 elements in the list, there are 4 ways to fail:



Again we want the height h of the tree, and it must be just big enough to allow n + 1 leaves, so

n + 1 £ number of leaves at level h

and again from FACT 1,

number of leaves at level h £ 2^h

which means we want

2^h ³ n + 1

or (taking logarithms and the ceiling function as before)

h = é lg(n + 1)ù

The average leaf height is

~ lg(n + 1)

Now, how about work? In Binary 2, remember that each internal node on the way to a failure requires 2 comparisons, so in the worst case, the work will be

2* é lg(n + 1)ù

and for the average case about

$$2 * \lg(n + 1)$$

A Binary 2 successful search terminates at an internal vertex. These are scattered throughout the tree, at various levels, so we'll use a more round-about approach.

Definition: E = external path length of a tree = $\sum_{\text{leaves}} (\text{pathlength})$

I = internal path length of a tree = $\sum_{\text{interior vertices}} (\text{pathlength})$

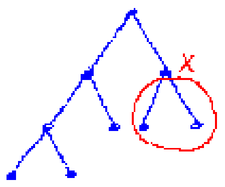
FACT 2: In a 2-tree with q internal vertices, E = I + 2q

Proof: by induction on q.
Base case: $q = 0$. There are no internal vertices, and the entire tree consists of just the single root node.
$E = 0$, $I = 0$, $q = 0$, so $E = I + 2q$.

Assume that for any tree with k internal nodes, $E_k = I_k + 2k$

Show that for any tree with k + 1 internal nodes, $E_{k+1} = I_{k+1} + 2(k + 1)$

Consider any tree with k + 1 internal nodes. Pick an internal node x whose two children are leaves, and delete the two leaves.



Now x is a leaf in a new tree with k internal nodes (since we reduced the number of internal nodes by 1). The inductive hypothesis applies to the new tree, so

$$E_k = I_k + 2k$$

Now let the path length to x be m. Then

$$I_{k+1} = I_k + (\text{path length to } x)$$

or

$$I_k = I_{k+1} - m$$

and

$$E_{k+1} = E_k + (\text{path length to } x\text{'s 2 children}) - (\text{path length to } x)$$

or

$$E_{k+1} = E_k + 2(m + 1) - m$$

$$E_{k+1} = E_k + m + 2$$

$$E_k = E_{k+1} - m - 2$$

Substituting these expressions for I_k and E_k into $E_k = I_k + 2k$ gives

$$E_{k+1} - m - 2 = I_{k+1} - m + 2k$$

$$E_{k+1} = I_{k+1} + 2k + 2$$

$$E_{k+1} = I_{k+1} + 2(k + 1)$$

End of Proof

In the Binary 2 tree, there are n + 1 leaves (failures), all occurring at approximately the height of the tree, so

$$E \sim (n + 1)\lg(n + 1)$$

There are n interior vertices (the successes), so by FACT 2, E = I + 2q,

$$(n + 1)\lg(n + 1) \sim I + 2n$$

or

$$I \sim (n + 1)\lg(n + 1) - 2n$$

This means that the average path length TO an interior vertex is

$$\sim \frac{(n+1)\lg(n+1) - 2n}{n}$$

Each node on the path TO an interior vertex has 2 comparisons, plus there is 1 more comparison at the final (success) node, so

$$\text{work} \sim 2\left[\frac{(n+1)\lg(n+1) - 2n}{n}\right] + 1$$

$$= \frac{2(n+1)}{n}\lg(n+1) - 4 + 1$$

$$= \frac{2(n+1)}{n}\lg(n+1) - 3$$

Binary Search Trees

We introduced binary trees to find another searching algorithm. Our binary search algorithms were efficient, but they used an array implementation, in which it is definitely inefficient to insert and remove data because of having to shuffle items forward and backward in the array.

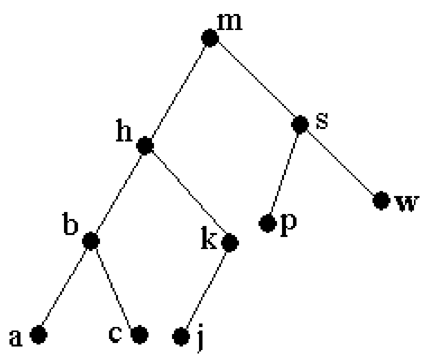
Our goal is to build a data structure that looks like the decision trees for binary search. In those trees, nodes in the left subtree of a given node represented locations in the sorted array holding data values smaller than at the given node; nodes in the right subtree represented locations holding data values larger than at the given node. A binary search tree incorporates this property.

A **binary search tree** (BST) is a binary tree that is either empty or in which every node has a key and

1. the root key > any key in the root's left subtree
2. the root key < any key in the root's right subtree
3. the left and right subtrees are again binary search trees

Once again, this definition is recursive, and says that *at any node*, the nodes in the left subtree have smaller keys and the nodes in the right subtree have larger keys.

Here is an example of a binary search tree using letters of the alphabet as key values.



Binary Tree Search

Once we have a binary search tree, it is easy to do a binary tree search on it to find a particular target item. The search begins at the root and either finds the target key value there, which terminates the search successfully, or if the target is less than the key value at the root, the process repeats on the left subtree, otherwise on the right subtree. The recursion terminates when the root is NULL and the search is then unsuccessful.

Click here for an animation of a successful binary tree search

Click here for an animation of an unsuccessful binary tree search

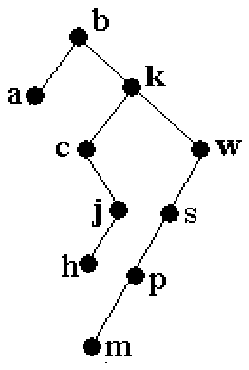
We'll postpone an analysis of the binary tree search until later. We hope it will be equivalent to binary search, that is, Q(lg n), but this depends on the shape of the tree.

Of course you can only search a BST that already exists, so our next task is to see how to create one. You create a BST by successively inserting nodes into the tree, but you have to insert them in such a way as to preserve the BST property, that is, so that all keys in the left subtree are smaller and all keys in the right subtree are larger.

Inserting a node into a BST is very similar to doing a binary tree search. In the search, you look for where the node should be and you either find it or not; for the insertion, you look for where the node should be if it were there and if it isn't, you put it there. No array shuffling.

Click here for an animation of building a BST

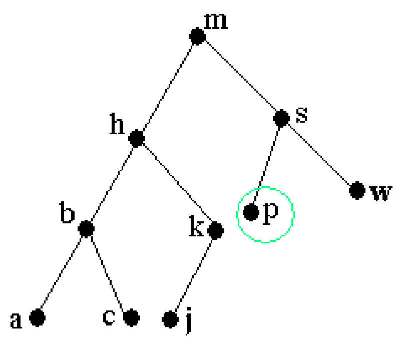
The same data can lead to different binary search trees, depending on the order in which insertions are made. Below is another BST using the same data as our previous example. This is why the shape of the tree can be so variable.



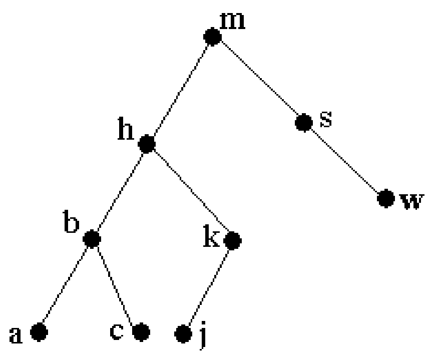
We should also be able to remove nodes from a BST. This is a bit more difficult a task, because again we must preserve the property of being a BST. There are three cases:

Case 1: Remove a leaf. This is simple - remove the leaf and the arc that connects it to its parent. This means that a pointer at the parent node must be set to NULL. The ordering of all other nodes is unchanged, so the tree is still a BST.

Example: remove p

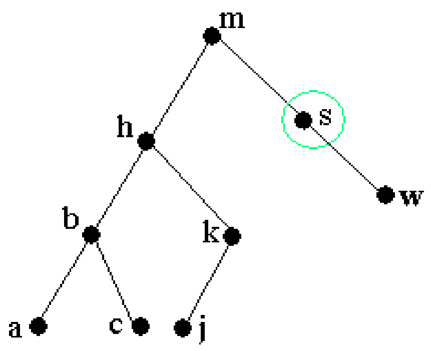


Result is

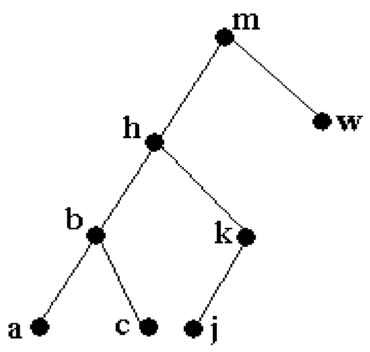


Case 2: Remove a node with one subtree. This is also relatively easy - the pointer from the parent to the node to be removed must simply be redirected to the root of the subtree.

Example: remove s



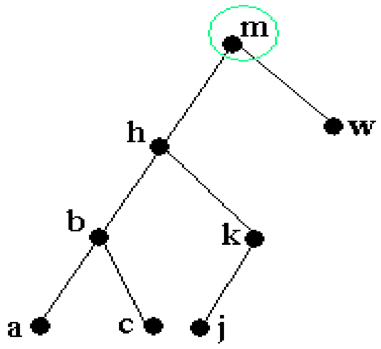
Result is:



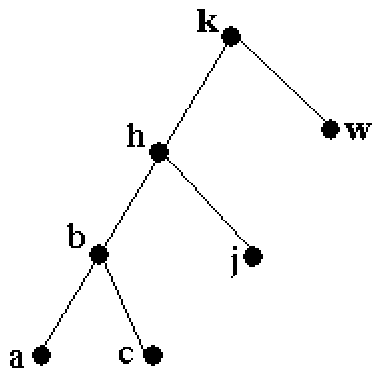
Case 3: Remove a node with 2 subtrees. The idea here is the following. We find the immediate predecessor of that node in sorted order. This can be found at the lower rightmost corner of the node's left subtree. Why? Because it's in the node's left subtree, it's key value is smaller than the node's key value. Because it is in the farthest right position in that left subtree, it's key value is larger than any of the others. This immediate predecessor is going to replace the removed node. (The immediate predecessor has at most one subtree, so it is easy to remove. If it has no subtrees, its parent node must be set to NULL, and if it has one subtree, then - just as in case 2 - its parent pointer must be redirected to pick up the subtree.)

Finding the immediate predecessor involves moving down and to the right in the left subtree, and the necessity to know about the immediate predecessor's parent in order to make the appropriate settings means that there will have to be a "lagging" pointer that, once the immediate predecessor is found, points to the predecessor's parent node.

Example: remove m



Result: m's immediate predecessor is k. k is removed, it's parent h picks up its left subtree, and k then replaces m



Binary search trees are implemented as a derived class from regular binary trees. We need a new way to insert, over-riding the inherited insert function, plus functions to search and to remove. As in the case of regular binary trees, these functions are accomplished with one-liner member functions that call recursive auxiliary functions.

Binary Tree Traversal

Traversal of a list meant starting at the front of the list and moving toward the back. This is a systematic way to visit each list element; while visiting, you can do some task such as writing out the list element, doubling the list element value [for numeric elements], etc.

A binary tree structure is not sequential, as a list is, and it's not so obvious how to traverse the tree in a systematic way so as to visit each node. There are three popular traversal mechanisms, depending on when the root of a subtree is visited as compared to its left and right subtrees.

Inorder traversal: First do an inorder traversal on the left subtree, then visit the root, then do an inorder traversal on the right subtree.

Preorder traversal: First visit the root, then do a preorder traversal on the left subtree, then do a preorder traversal on the right subtree.

Postorder traversal: First do a postorder traversal on the left subtree, then do a postorder traversal on the right subtree, then visit the root.

Note that these algorithms are recursive. Most of the algorithms about binary trees will be recursive because the binary tree structure itself was defined recursively.

Click here for animations of the three traversal algorithms.

Expression trees are binary trees in which inorder, preorder, and postorder traversal have rather practical consequences.

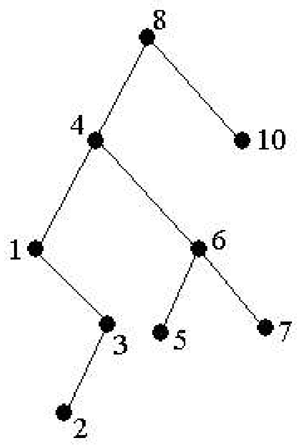
As a recursive function calls itself over and over, each invocation causes an activation record consisting of local data plus a return address to be pushed onto a run-time stack. (See the *Run-time stack for tree traversal figure.doc in the Code Archives* for a nice illustration.)

Actually, there is another tree traversal algorithm that uses a queue instead of a (implicit) stack. This is a **level-order traversal** where the root is visited first, then all nodes at level 2 in left to right order, followed by all nodes at level 3 in left to right order, etc. Here's the algorithm:

```
Create a queue and enqueue a pointer to the tree root.
While the queue is not empty
    Dequeue the head of the queue
    Visit the node
    Enqueue pointers to its left and right child (if present)
```

Example:

Given the binary tree



where "visiting" a node prints it out, the results are as follows:

in-order traversal: 1 2 3 4 5 6 7 8 10
pre-order traversal: 8 4 1 3 2 6 5 7 10
post-order traversal: 2 3 1 5 7 6 4 10 8
level-order traversal: 8 4 10 1 6 3 5 7 2

Binary Trees

We implemented lists for searching as arrays rather than linked lists because binary search involves moving toward the front of the list, difficult to do in a (simple)linked list structure. We *can* use links (pointers) in searching if we can set up a data structure that looks like the decision tree for a binary search. Remember that a decision tree is NOT a data structure, but merely a picture of how the search algorithm acts. Now we want to create a tree-like data structure.

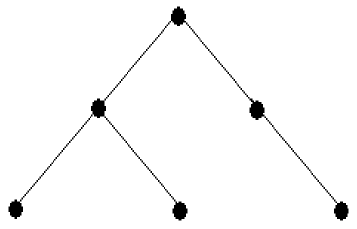
Again recall the terminology connected with trees. Except it is now convenient to change our definition of tree height so that an empty tree has height 0, the root of a tree is at level 1, etc. This does not change our previous analysis results about searching.

Definition: A **binary tree** is a tree that is empty, or where the root is the parent of two binary trees called the left subtree and the right subtree of the root.

Comments on this definition:

a. it is a recursive definition. Note that "binary tree" appears in the definition of binary tree. The empty tree is the base case.
b. left and right is important
c. the left or right subtree could be empty
d. each node has *at most* 2 children (as opposed to a 2-tree where each node has 0 or 2 children)

EXAMPLE: Binary tree with 6 nodes



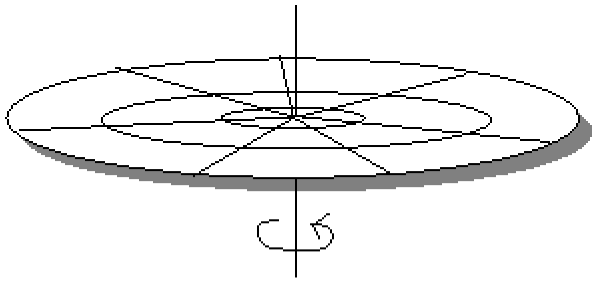
To develop the binary tree as an abstract data type (ADT), we need to consider the natural operations we might want to do on a tree. These would include

- create a tree (this will be a constructor in the implementation)
- test if a tree is empty
- insert nodes into the tree
- remove nodes from the tree
- find the number of nodes in the tree
- find the height of the tree
- traverse the tree - visit all the nodes in some systematic way

Later we will specialize this general binary tree structure (in implementation, this will mean a derived class) to tree structures that support efficient search algorithms.

B-Tree Rationale

Why use B-trees? Suppose we have an external data file, i.e., a collection of records too large to store in internal memory, and we want to search for a particular record by finding its key value (the value that uniquely identifies that record). Example: find the record for Walmart Employee # 39782. Reading a record from external storage requires a disk access. Data on a disk is stored in concentric tracks, divided into sectors. Each sector contains a **block** (a **page**) of data.



The time to read a particular block of data into memory is composed of three components:

1. *Seek time* – the time to position the read head over the proper track. This time varies depending on the relative position of the read head and the proper track when the read request is generated. In the best case, the read head is already over the proper track and the seek time is 0. At the worst case, assuming there are n tracks, the read head might be over track 1 and have to move to track n, which would be n – 1 units, where a *unit* is the distance between adjacent tracks. We can assume that the seek time would be some multiple of the number of units. This is the slowest part of the read operation because it involves mechanical movement of the read arm (think of moving a phonograph arm, if you at least have seen pictures of how a phonograph works!).

2. *Latency* – the time for the proper sector to rotate underneath the read head. This time also varies, depending on whether the correct sector is just coming under the read head (minimum latency time) or whether it has just gone by and a full rotation must occur (maximum latency time).

3. *Transfer time* – the time to read a block once it is positioned under the read head, usually a constant amount of time for all blocks.

Such accesses are orders of magnitude slower than internal operations such as fetching data from main memory in order to do a computation, or comparing two data elements. Because of the expense of a disk access, the computer never reads or writes a single record from or to a file. Instead it reads a block, which contains a number of sequentially stored records. Even though a language like C++ may make it appear that only a single record is being read, a block is read and then the particular record is located from within the block. So multiple records can be accessed by a single block read. This fact will allow us to minimize the number of disk reads needed to search for an item, even more than just by a factor of the block size.

What search options do we have available? Something like a sequential search (reading one block after another from the disk and searching the records each contains) requires many disk accesses, although perhaps reduced seek time for each access. None of our tree searches would work because trees only exist as internal data structures. However we solve this problem, we will want to minimize the number of disk accesses.

Instead of working directly with the large external file of data records, we instead work with a second, smaller external file called an **index file** for the original data file. Each block of the index file is organized in the form

p_0	k_1	a_1	p_1	k_2	a_2	p_2	....	k_{m-1}	a_{m-1}	p_{m-1}

where a block need not be full.

k_i = key value in data file (value that identifies the record)
a_i = block number of the block in the <u>data file</u> that contains the record with key value k_i
p_i = block number of a block in the <u>index file</u> (like a pointer to another block)

Think of k_i-a_i as a unit.

Now think of one of these blocks as a "node" and you see that it can be a node in a B tree, i.e., we have a tree structure *in the file itself*, not in internal memory. Note - this tree structure does not actually exist, it is only how we are interpreting this collection of blocks that makes up the index file. (Recall a tree doesn't really exist in internal memory either, it's just an array of memory locations with some pointer fields that we can *interpret* as a tree structure.)

To do a search, we search the index file following the correct pointers (like a B-tree search). Following a pointer in the index file gets you to a new node, that is, a new page, so it represents a disk access. The block (page) is read into main memory where its various key values and pointer values can be examined using short sequential search (or maybe even a binary search) - this is how we get multiple key values per disk read organized in a way that is useful to us in big leaps, instead of reading a page of the data file, which is just a speedup of sequential search. (When we follow the next pointer in the index file, we must swap *this* page back to the disk.) At the correct key value, we use the corresponding address value to find the real data record.

We must maintain the index file as a B tree. As new items are inserted into the nodes (blocks), blocks may be split. Under deletions, blocks may recombine. Note there is one entry in the index file k_i-a_i - for each record in the data file. But because data records would normally contain much more than just the key value, the index file - even with its additional "pointers" - is much smaller, and shuffling to split and recombine blocks is minimal because we have less to move. In addition, whatever organization we impose on the index file is indirectly imposed on the data file, although the data file records are never moved. For example, a new element can be appended to the end of the data file, with no reorganization of the data file, and only its key value-address pair gets shuffled into its proper place in the index file. An element can be logically deleted from the data file by deleting its key-value address pair from the index file, with no reorganization of the data file. (After a time, however, we have lots of "dead space" in the data file and we would have to use the index file to build a new data file, then build a new index file for it.)

Searching a B-tree involves traversing the tree down a a path from a root to a leaf, with each node causing a disk access. Therefore in the worst case, the number of disk accesses required is equal to the depth of the index tree. For the ideal binary search tree, this is

Q (log$_2$ n)

but for the ideal B-tree it is

Q (log$_{\text{é m/2ù}}$ n)

because there are at least ém/2ù branches per node. In other words, because of many values per node, the tree is fat and, more importantly, short. In the worst case an insertion or deletion operation follows a path from a leaf clear back to the root node, so the number of disk accesses is again Q (log$_{\text{é m/2ù}}$ n) . More specifically (no proof here)

$$h \le \log_{\lceil m/2 \rceil} \frac{n+1}{2} + 1$$

so that if m = 200 and n = 2,000,000, then h <= 4.

Notice that in dealing with data on external files, our unit of work has changed. We are now worried about the number of disk accesses required, and not about comparing a target value against a data value; that operation is done internally once the appropriate block has been pulled into main memory and thus it's so much faster than a disk access that it's not the concern any more.

Bubble Sort

The Algorithm

Bubble sort is one of the easiest sorts to understand, and probably the one most students would describe if asked to give a sorting algorithm. Unfortunately, as we will see, it's quite inefficient.

The idea behind bubble sort is to take repeated passes through the list. At each pass, successive pairs of elements are compared, and swapped if they are out of order. At the end of pass 1, the highest element will be in its proper position at the end of the list. At the end of pass 2, the second-largest element will be in place, etc. The sorted sublist grows from back to front, as in selection sort.

Click here for Bubble Sort animation

Pseudocode for the main part of bubble sort follows:

```
for (int i = count -1; i > 0; i--)
{
    for (int j = 0; j < i; j++)
    {
        if (entry[j] > entry[j+1])
            swap(entry[j], entry[j+1]);
    }
}
```

Analysis

Each pass through the inner **for** loop results in one comparison, and the inner **for** loop is executed i times for each pass through the outer **for** loop. Therefore the total number of comparisons is

$$(n-1)+(n-2)+\ldots+1=\frac{1}{2}(n-1)n=\frac{1}{2}n^2+\Theta(n)$$

The total number of assignments is harder to count because a swap is not always done. Here is where we can determine an average case, worst case, and best case.

Average case: Assume that 50% of the time the inner **for** loop is executed, a swap is required, which results in 3 assignments.

The number of assignments is therefore

$$\frac{1}{2}*3*\left(\frac{1}{2}n^2+\Theta(n)\right)=\frac{3}{4}n^2+\Theta(n)$$

Worst case: Every pass through the inner loop requires a swap, which means that the original list is in reverse sorted order. The number of assignments is then

$$3*\left(\frac{1}{2}n^2+\Theta(n)\right)=\frac{3}{2}n^2+\Theta(n)$$

Best case: A swap is never required, which means that the original list is in sorted order. No assignments are done.

Sorting Summary

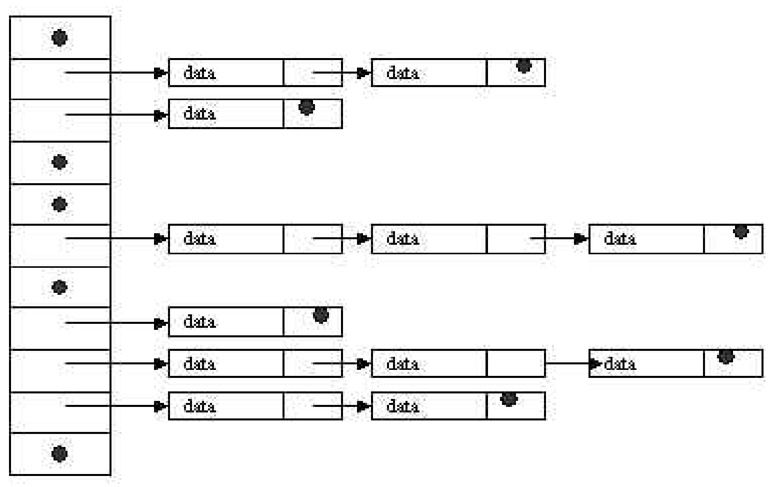
Sorting Summary			
Algorithm	Worst	Average	Best
Insertion	???	random list	already sorted
compares	???	$0.25n^2$ + Q (n)	n - 1
assigns	???	$0.25n^2$ + Q (n)	0
Selection		any list	
compares	same as ->	$0.5n^2$ + Q (n)	<- same as
assigns	same as ->	3n + Q (1)	<- same as
Bubble	reverse sorted	random list	already sorted
compares	$0.5n^2$ + Q (n)	$0.5n^2$ + Q (n)	$0.5n^2$ + Q (n)
assigns	$1.5n^2$ + Q (n)	$0.75n^2$ + Q (n)	0

Here it is clear that in the average case, bubble sort gives us the worst of both insertion sort and selection sort. Bubble sort requires the same number of comparisons as selection sort, and even more assignments than insertion sort.

Collision Resolution - Chaining

The "chaining" solution to collision resolution is to make the hash table an array of pointers to linked lists of records. The hash table is initialized to all NULL pointers, so originally each linked list is empty. When a key value hashes to a particular array index, it gets inserted into the linked list for that index.

After a while, the picture of the hash table might look something like this:



To insert an element, hash to the array index, then insert into the linked list after searching the list to be sure this is not a duplicate item. To search for an element, hash to the array index, then search the linked list. Insertions can be done at the head of the list, which is a quick operation, but searching the linked list then becomes sequential search. If the list is maintained in sorted order, then it's more work to insert the element, but a short sequential search can be used to search the list.

Removing an element from the linked list is also easy - once the item is located, rearranging a couple of pointers cuts it out of the list. There is no need for the logical delete needed for open addressing.

Chaining vs. Open Addressing

Advantages of chaining (which are pretty much the disadvantages of open addressing):

1. Chaining uses dynamic storage. When you want to store another record, you grab more space from the heap and add the record to the appropriate linked list. You can't "run out of room" as you do in an open-addressing hash table of a fixed size.
2. Along the same lines, you don't have to estimate the number of records you will store in order to set up the hash table.
3. Collision resolution is simple, no complicated probing formula to apply.
4. Items you wish to delete can actually be deleted.

Disadvantage of chaining

1. If the record size is small, then the extra space of a pointer for every record becomes significant.

These are primarily space issues. In searching under either chaining or open addressing, we still have a work unit that consists of comparisons of targets to key values. Next we'll do an analysis of the average work, for both successful and unsuccessful searches, for these two approaches.

Common Data Structures

Introduction

Consider what happens when you declare a simple int variable in your C++ program,

```
     int n;
```

This makes a "reservation" for a location to be set aside in memory when the program is loaded, and asks the compiler to bind the identifier *n* to that location. Because type int is an intrinsic (built-in) C++ data type, the system immediately understands how many bytes to set aside and how to interpret the binary data that will be stored in that location. So far, we think of this as data only. But there's much more to this data type. The extraction and insertion operators are defined for type int, so it's possible to read in a value using

```
     cin >> n;
```

or to write out a value using

```
     cout << n;
```

Furthermore there is an assignment operator defined:

```
     n = 81;
```

Arithmetic operations are defined:

```
     cout << n + 6;
     cout << n % 5;
```

and increment/decrement operators:

```
     n++;
```

But wait, there's more. Relational operators are available:

     `if (n != 7), while (n <= 5)`, etc.

So we see that this intrinsic data type includes not only the data value itself but also a whole suite of useful operations that can be done on this data value, without which the data value itself is relatively useless. You, as the programmer, don't know, and don't need to know, exactly how these operations are implemented. You don't have access to the actual low-level assembly-language type code that actually allows two binary strings that represent integers to be added together. This leads to the concept of an **Abstract Data Type (ADT)**.

An ADT is not an actual data type. As the name suggests, it's an idea.   You think about some (collection of) data you want to store, and imagine **what** sorts of operations would be useful to perform on that data collection. Only then do you begin to think about **how** to implement these operations.

The List ADT

Consider the idea of a "list" ADT. CS folks may immediately think of a linked list, but that's a particular implementation. Again, we want to think in the abstract. A good analogy is the mathematical concept of a sequence (finite, of course, since we will eventually be constrained to implement this on a real computer).

A k-element mathematical sequence is written something like

$s_1, s_2, s_3, \ldots, s_k$

Order is important - there is a first element, second element, third element, etc. That's the data part of a list. Now, what are the useful operations we might want to perform on such a list? We'd like to be able to access an element in the list. We'd like to be able to change the value of a list element. We'd also like to be able to maintain the list - insert new elements, delete existing elements. We have to be able to create the list to begin with, initially empty. And we probably want to be able to test whether the list is actually empty at any particular time. You can perhaps think of other operations that seem to be logical and natural, but this will do for now. So let's create a list (no pun intended) of the operations we want to have:

- Access a list element
- Change the value of a list element
- Insert a new list element
- Delete an existing list element
- Create an empty list
- Test whether a list is empty

Notice that the first four operations deal with individual list elements, the actual data elements. The last two operations deal with the list structure as a whole, so they are "higher-level" operations, what we might call "metadata" operations.

Now let's look at some common implementations of the list ADT, in particular what is available in C++, and see how well these implementations support our desired operations. Aside on terminology: when we get to the implementation level, we discuss **data structures**. Data structures, unlike Abstract Data Types, are concrete; they are organizational schemes for storing multiple data items in the computer. So data structures are the implementation of half of the ADT concept, the data part. The implementation of the desired ADT operations requires additional code, i.e., algorithms.

Arrays

In addition to the intrinsic numeric and char data types that hold a single item of data, most procedural programming languages, including C++, support an **array** data structure. Arrays are designed to store multiple data items that are all of the same data type (all integers, all objects of the same class, etc.). It's easy enough to create an array by specifying the maximum size of the array. For example,

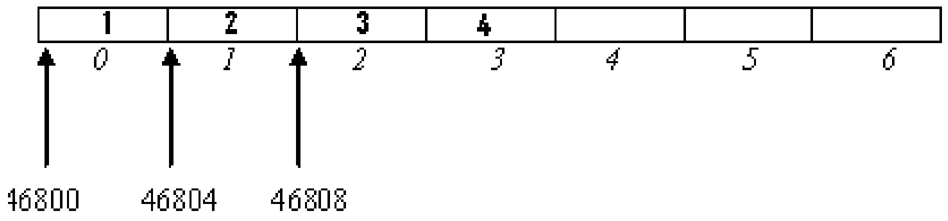
```
int Myarray[7];
```

declares an array to hold at most 7 integers. The array is "empty" at this point in the sense that it holds no meaningful data; the 7 contiguous memory locations hold whatever binary strings happen to be there. Or we could create and initialize the array in a single step.

```
int Myarray[7] = {1,2,3,4};
```

At this point *Myarray* contains four meaningful integer values (and three "garbage" values). An array is actually implemented (in part) as a pointer variable to the beginning of a contiguous block of memory. The system knows the starting address of the memory block (the pointer value) and it knows the size of each data element (for example, 4 bytes for an integer). But that's all it knows. It's up to the programmer to keep track of the number of meaningful data values, and also to be sure not to access memory locations beyond the end of the block (out-of-range errors). So for our two metadata operations, we can create an array but it would be up to us to keep track of how many meaningful data items are stored in the array, so there's no real implementation of the "is it empty?" operation.

The **subscript operator** or **index operator [ ]** supports the access operation we want for out list ADT. *Myarray[2]* takes us directly to the value stored at index position 2 in *Myarray*. Remember that array indices begin with 0, not 1. It might be more convenient if arrays counted the same way people usually do, i.e., so that the elements are identified as element1, element 2, etc. instead of element 0, element 1, etc. The indexing scheme works for the ease of the compiler, not for the programmer. In an array of integers like *Myarray*, for example, if a single integer requires 4 bytes, then to locate *Myarray[2]*, the compiler just multiplies the index value 2 * 4 bytes = 8 bytes and counts over 8 bytes from the starting memory address (the pointer value).



Updating a value also makes use of the index operator:

```
Myarray[2] = 15;
```

changes the third (not the second) array value from 3 to 15. But because the system does not keep track of the maximum index, the assignment

```
Myarray[7] = 12;
```

may or may not produce an out-of-range error, it may simply clobber whatever was stored in memory after the array block ends. Inserting a new element into the array (provided there is room) requires shuffling the remaining elements 1 cell to the right. Inserting a new value at Myarray[2] requires

```
Myarray[4] = Myarray[3];

Myarray[3] = Myarray[2];
```

and then

```
Myarray[2] = new value
```

Deleting an array element requires shuffling the remaining elements 1 cell to the left. If these operations of insertion and deletion are done (within some kind of loop statement) near the beginning of an array with *n* legitimate data values, this can require nearly *n* "shuffles" (assignments).

The scoreboard for an array as an implementation of the list ADT therefore looks like this:

Access a list element	one step if you know the index
Change the value of a list element	one step if you know the index
Insert a new list element	requires shuffling, perhaps up to n assignment statements
Delete an existing list element	requires shuffling, perhaps up to n assignment statements
Create an empty list	one step declaration
Test whether a list is empty	programmer must keep track of highest index used

So if you know exactly where you want to go in the array, you are in good shape. If you have to figure out where you want to be by searching for a particular array element, that takes some work. Searching is one of the major topics of this course - we will examine a number of searching algorithms, so we'll say no more about that here. Array maintenance - insertions and deletions - can be expensive operations.

There are some other operations at the "metadata level" - that is, dealing with the array as a whole, that we might like to have but are not available. The extraction and insertion operators are not defined for arrays in general (although they are defined for C-style strings, that is, arrays of type char), so you can't read in or write out an array all-in-a-hunk, you have to loop through the indices and read or write each element. Similarly, the assignment operator is not defined; you can't just say

```
array1 = array2;
```

and expect all the corresponding elements to be copied. Again, you have to loop through the indices and copy each element. (You can make the array a member element of a class and then write your own overloaded operators for extraction, insertion, and assignment.) One thing that you can do with an entire array is to pass it as a function argument. You will also need to pass an integer giving the highest valid array index so that within the function there's no out-of-range index value.

In addition to the danger of accessing an out-of-range index value, the other major disadvantage of the array data structure is that the memory allocation for the array is **static**. You have to say how big the array will be when you write the program, and that allocation can't be changed as the program runs. If you guess too small a value, you won't have a place to store all your data. If you guess too big a value, you'll be wasting storage space.

Vectors

The vector, unlike the array, is not an intrinsic C++ data type. It is a class defined in the C++ Standard Template Library. The STL contains a number of useful template classes. (Recall that a template class isn't actually a class, it is a "pattern" for a class that uses a dummy data type as a kind of place-holder. The template class generates a "real" class when the data type of the place-holder is specified.)

Think of a vector as a growable (dynamic) array. If you need more space, the vector will grow as needed. The vector class also supports the subscript operator [] so that direct access to the element at any given index is immediate, just as with an array. In addition, there are other useful member functions of the vector class. See *STL Vector class figures.doc* in the Code Archives folder on Oncourse.

You need

```
#include <vector>
```

to use vectors in your program. Then to construct a vector object you must specify the data type of the elements. The statement

```
vector<int> Myvector;
```

creates an empty vector that will contain integers. This creates a contiguous block of memory of system-dependent size, but the boolean *empty()* member function will return true because there's no legitimate data in the vector. You can find the maximum number of elements that can be stored in the vector's current configuration by invoking the *capacity()* member function, but in general you don't need to worry about this because if your vector is full and you add one more element, the system will create a bigger allocation, copy the current vector contents into it, and release the previous block of memory.

The STL contains a number of other **container** classes that generate data structures (like vectors) with multiple elements, usually of the same type. The STL also allows one to use **iterators** that are essentially pointers into container objects. To instantiate an iterator object, you must indicate what it will point to:

```
vector<int>::iterator i;
```

Or you can initialize the iterator at the same time, as in

```
vector<int>::iterator i = Myvector.begin();
```

where the *begin* member function returns an iterator pointing to the first vector element.

Iterators, like regular pointer variables, use the dereferencing operator * to indicate the element to which they point. They also follow the rules of pointer arithmetic, so that incrementing the iterator i causes it to point to the next vector element.

To walk sequentially through the elements of a vector, you therefore have two choices. You can increment the index, just the same way you walk through an array, or you can increment the pointer variable.

You usually build an array sequentially by advancing the array index and doing an assignment statement, thus adding new elements at the end. Similarly, you grow a vector at the end by using the *push_back*(element) member function. It's also easy to remove the last element in the vector using the *pop_back( )* member function. Other deletions require more effort. You may have to shuffle elements to the left, as in the array delete operation. You do have a short-cut if you happen to have an iterator pointing to the element to be deleted because in that case you can use the *erase* member function. Similarly, to insert an element in an arbitrary position would generally require shuffling succeeding elements to the right, as in the array insert operation. But if you happen to have an iterator pointing to the element before which you want to do an insertion, you can use the *insert* function (which actually has several forms). Do the *erase* and *insert* functions perform shuffling behind the scenes? Probably, but we'll never know. We see only the STL class header files - **what** the class operations are, not **how** they are implemented. (Information hiding at work.)

A vector's success as a list ADT implementation is similar to the array data structure, with a few minor improvements:

Access a list element	one step if you know the index
Change the value of a list element	one step if you know the index
Insert a new list element	in general requires shuffling, perhaps up to n assignment statements, unless there is an iterator pointing to the right spot
Delete an existing list element	in general requires shuffling, perhaps up to n assignment statements, unless there is an iterator pointing to the right spot
Create an empty list	one step declaration
Test whether a list is empty	use the boolean *empty()* function

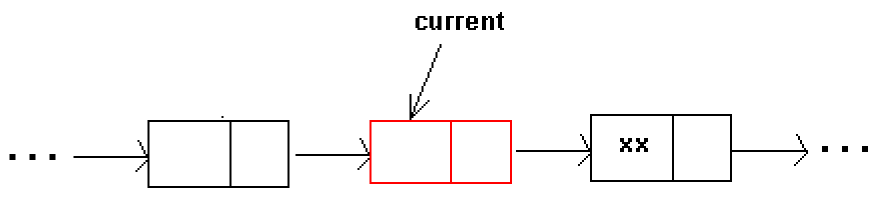
The major improvement of the vector over the array is that you always have more space if needed, so you don't have to guess at the size needed, you don't run out of room, and you don't get index out_of_range errors.

Linked Lists

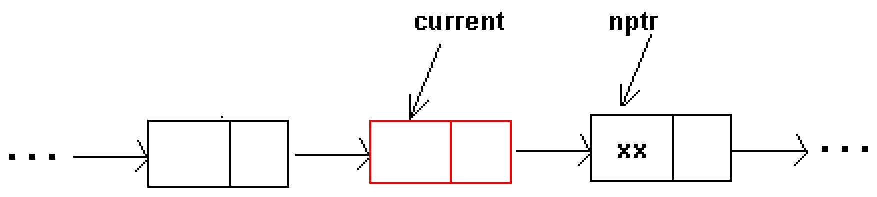
The linked list data structure makes heavy use of pointers. The list member variable is a pointer to the head of a chain of nodes, and nodes contain, at a minimum, a data item of some fixed data type plus a pointer to the next node in the chain. You create an empty list by setting the list pointer to NULL (or 0). Then you add nodes by using the *new* operator to create a new node, assign it a data value, and hook it into the existing list. The *new* operator grabs storage from an unallocated section of memory called the **heap**. Unfortunately, in C++ space allocated from the heap using the *new* operator is not always returned to the heap once its "lifetime" has passed, so the "inverse" of the *new* operator is the *delete* operator. The tidy C++ programmer will always perform "garbage collection" by using the delete operator when appropriate.

Within this general framework, there are many possible implementations of a linked list. The one we will use (for the time being) is an expansion of the linked list in Activity 2. It stores integer data. To make lists a little simpler to work with, this class has a *last* (tail) pointer as well as a *first* (head) pointer. There is also a *current* pointer (the equivalent of an iterator).

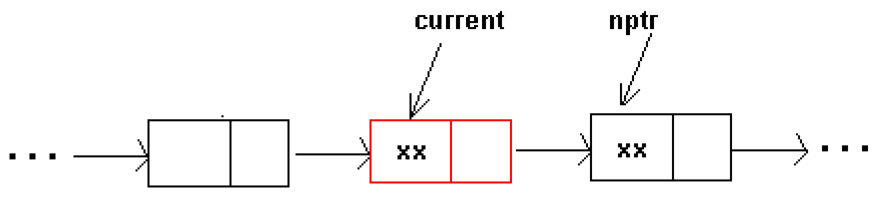
Let's illustrate what happens when you want to delete an arbitrary element from the middle of a linked list. Here's the situation:



The *current* pointer points to the node to be deleted. Next you set a new pointer to the node after *current*, which you can easily reach from *current*.

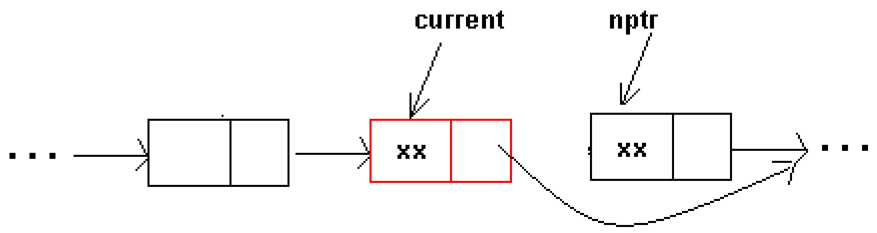


Copy the data from *nptr's* node to *current's* node - this will destroy the data originally in this node, but that's OK, you want to get of that data anyway.



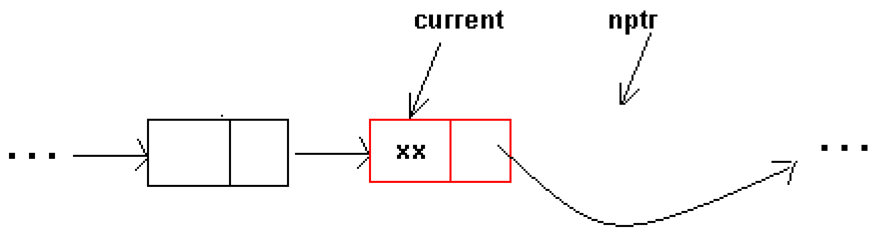
Rehook the pointer from the *current* node so that it bypasses the node pointed to by *nptr* and just cuts that node out of the chain:

```
current ® next = nptr® next;
```



This does the relinking, but to reclaim the space nptr points to, we do a delete operation:

```
delete nptr;
```



This returns the space pointed to by *nptr* to the heap and renders the value of *nptr* undefined. (The value of any other pointer variable that points to the same node as *nptr* would also be undefined at this time.) Notice the importance of the order of operations done here. If we had reversed these two statements and done a delete operation on *nptr* first, the value of the pointer `current ® next` would be undefined, and the rest of the list (after the deleted node) would be unreachable. Anytime you rearrange pointers, you need to be very careful of the order of operations.

Note also that in this implementation, we didn't actually delete the physical memory location that *current* pointed to, we copied data from the next node and then deleted the next node. Why is this (unexpected) feature so clever? Actually deleting the node *N* that *current* points to would have required walking another pointer *p* through the list to the node before *N* in order to redirect that node's *next* pointer around *N*, a strategy requiring more work than what we've done here.

But this does indicate the need to consider a special case. What if *current* points to the last node? Then the process illustrated above won't work because there's no node after this one. If this case we do indeed have to walk a pointer *p* through the list to the node before *current*, actually delete the current node, and reset the *last* pointer to the node pointed to by *p*. But this special case presumably occurs less often than the general case, so this implementation makes the general case more efficient. Other special cases include deleting the only node in a list, and trying to delete from an empty list.

The list implementation we will use has two insert functions - one to insert a new node before the node *current* points to, and one to insert a new node after the node *current* points to. Both require creation of a new node and a couple of pointer redirections. When you read the code for these two operations, I suggest you draw a series of picture like the ones above to see exactly what happens in each case..

Insertion and deletion of a node from a list is therefore in general just a matter of redirecting a couple of pointers. BUT - you have to have a pointer to the node you want to delete or to the place you want to insert. Unlike arrays or vectors, there is no subscript operator, so no direct access into the list (except at the first node and the last node). To get to any other node in the list, you have to walk through the list from the first node, following the chain of pointers. So here's how the linked list data structure meets our list ADT desires:

Access a list element	walk through the list from the first node

Change the value of a list element	walk through the list from the first node, then do an assignment
Insert a new list element	if you have a pointer to the insertion spot, the general case just requires a new statement and a couple of pointer redirections
Delete an existing list element	if you have a pointer to the deletion node, the general case just requires a pointer redirection and a delete
Create an empty list	one step declaration
Test whether a list is empty	check whether head pointer is 0

In a way, what was easy to do in arrays and vectors is harder in a linked list, and vice versa. But the "if" condition for insertions and deletions is a big one.

What we've described here is a **singly-linked list**. Fancier versions of a linked list exist. One can maintain two pointers at each node, one to the next node and one to the previous node (a **doubly-linked list**). Then you can walk through the list in either direction. Or you can have a pointer from the "last" node to the "first" node (a **circular linked list**).

Unlike the array structure, you can't run out of space with a linked list (unless you use up the entire free space in the heap!). But there is a space penalty. Although a pointer variable takes a small amount of memory, every node in the list has a pointer variable (two in a doubly-linked list) in addition to the data value, so that space can add up.

There is a template list class in the STL that provides many member functions, and we'll use it later in this course.

Look-ahead to Files

Consider again the four desired operations we want to be able to perform on the level of individual list elements, with slightly different names attached.

- Insert a new list element (Create it)
- Access a list element (Read it)
- Change the value of a list element (Update it)
- Delete an existing list element (Delete it)

When we talk about file structures at the end of the semester, we will want to be able to do these same four operations on the individual entries or "records" in a file. How these operations are done depends on the file organizational structure, just as these operations differ in implementation for the three internal data structures we've talked about above. In the context of files, these four operations are known as CRUD rules.

The Stack ADT

Back to the abstract data type - an idea about some arrangement of data and useful operations to be performed on that data. A **stack** is a special kind of list. Although we think of (linear) lists laid out from left to right, stacks are invariably described as laid out vertically. That is perhaps because the quintessential analogy of a stack is the stack of plates at the beginning of a cafeteria line. New plates are added to the top of the stack, and a plate can only be removed from the top of the stack. So that's the abstract idea - we have a special kind of list where only the "top" is active - that's where the stack grows because new elements get added there, and only the top element can be accessed, changed, or removed. Adding a new element to a stack is called a *push* operation and removing the top element is called a *pop* operation. A stack is a **LIFO structure** - last in, first out.

Here's our list of desired operations:

- Clear the stack (make it empty)
- Test whether the stack is empty
- Access the top element
- Pop the stack (remove top element)
- Push an element onto the stack top

Now, how do we implement this abstract idea? Since a stack is a restricted kind of list, we can write a class whose basic data structure is an array, a vector, or a linked list, and then write member functions that support only the desired operations. Because activity takes place only at the end (the top of the stack), the traditional disadvantages of the array or vector implementation, that is, shuffling to delete or insert elements, are non-issues. In a linked-list implementation, if you think about the fact that activity takes place only at the top of the stack (end of the list), you see the value of a pointer to the end of the list. But if you pop the stack, the previous element becomes the new stack top and now you want to have access to that element. In a singly-linked list, you have to walk a pointer through the list from the head pointer to the next-to-last element, just as we described earlier when deleting the last element of the list. Because this is the only kind of delete you are going to do in a stack, it makes sense to use a doubly-linked list where it is easy to step back to the next-to-last element from the last element. *Stack figure.doc* in Code Archive illustrates a series of stack operations in the abstract and in the vector and linked list implementations.

There must be some reason why this sort of structure is useful. The compiler does delimiter checking (in C++, this means matching left and right curly braces, parentheses, or square brackets). When a left delimiter is encountered, it is pushed onto the stack. Other symbols are ignored until a right delimiter is encountered. This is compared with the stack top; if they "match", the stack is popped and processing continues, otherwise there's an error. If the process continues until the end of file, then the stack should be empty, otherwise there's an error. No doubt you have gotten such an error, something like "mismatched or missing parentheses".

Whenever a function is invoked in a C++ program, an "activation record" is created and placed on a stack. The activation record contains a snapshot of the conditions (values of variables) at the point at which the function is invoked, and also contains the address to which the system should return when the function is exited. This allows processing to continue from the point of invocation. In a recursive function, where the function is invoked over and over, there is a danger of "stack overflow" as these activation records pile up.

The Queue ADT

The queue ADT is another variation on a list. Here elements can only be added at the back of the queue (an **enqueue** operation) and can only be deleted from the front of the queue (a **dequeue** operation). You are familiar with a queue when you stand in the checkout line at the grocery store; you enter the back of the line and you are "deleted" from the front of the line after the cashier checks you out. A queue is therefore a **FIFO structure** (first-in, first-out). What do we want to be able to do with a queue?

- Clear the queue (make it empty)

- Test whether the queue is empty
- Access the first element
- Dequeue an element from the queue (remove first element)
- Enqueue an element into the queue (add it to the back)
-

As in the case of the stack, a queue is a restricted kind of list, so could be implemented as an array, a vector, or a linked list. But now, while adding a new element at the back of an array or a vector is no problem, removing the first element requires shuffling (or in the case of the vector, using the *erase* function which we suspect also requires shuffling). For a linked list implementation, you can use a singly-linked list with a pointer to the front and a pointer to the back. You enqueue at the back, which just involves resetting the *back.next* pointer to the new node and then bumping up the *back* pointer by the assignment statement `back = back.next`. You dequeue from the front, which just involves setting a temporary pointer *tmp* to the front node, setting `front = front.next`, and then deleting *tmp*. So each operation is just a couple of pointer assignments, making the linked list the likely implementation of choice.

Applications of the queue include its use in a graph traversal algorithm (we'll see this later this semester) and in simulations. A simulation of the grocery store would use a queue in which new customers join a line at the back and are served from the front as cashiers become available. An effective simulation requires some data on the arrival rates of customers (generally not uniform) and the service rate (how fast the customer gets checked out - also not uniform). Using such simulations, the store can figure out how long a customer has to wait in line and then decide whether it needs to hire more cashiers or can use fewer cashiers. Of course the grocery store generally makes such decisions on conditions at the moment, but a simulation can apply to any sort of "throughput" problem, such as whether a factory needs more paint spray stations to service the arriving parts.

Now if you are standing in the grocery store checkout line and President Obama arrives at the end of the queue to buy a box of Twinkies, one might expect that he'll be pulled out of the line and served ahead of you. That's because he might be considered to have a higher priority as a customer than you do. A **priority queue** is a queue in which items with a higher priority are dequeued first, so it's no longer strictly a FIFO structure. There are several ways to implement a priority queue, but we'll postpone this discussion until later in the semester.

Help from the STL

The Standard Template Library provides a stack class, a queue class, and a priority queue class. In general, the member functions of these classes support our desired tasks, but of course you have to know the exact member function names. (See *STL stack and queue figures.doc* in Code Archive.) How are these structures implemented? Again, as users of these classes, we don't need to know.

Decision Trees

For an algorithm whose unit of work is comparison, a **decision tree** can be a useful tool to analyze the amount of work.

Quick review of tree terminology

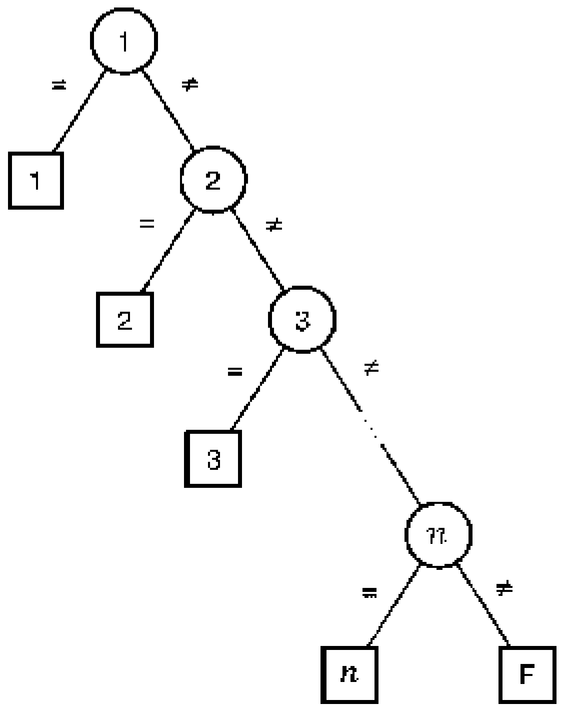
A decision tree is NOT a data structure. It is a pictorial representation of the actions of an algorithm that uses comparisons. Each interior vertex (a circle with a number in it) represents a comparison (for search algorithms, of the target against a list element); the number is the index in the array where that list element is stored (it has nothing to do with the value stored there). The branches below that vertex represent the path taken by the algorithm - what is done next - as a result of the comparison. The leaves of the tree represent the final outcome of the algorithm (for a search algorithm, these will be F -failure - for an unsuccessful search, or the position of the target for a successful search).

The level of a leaf, since it equals the number of interior vertices from the root to the leaf, also equals the number of comparisons made (the work done) to arrive at the outcome the leaf represents.

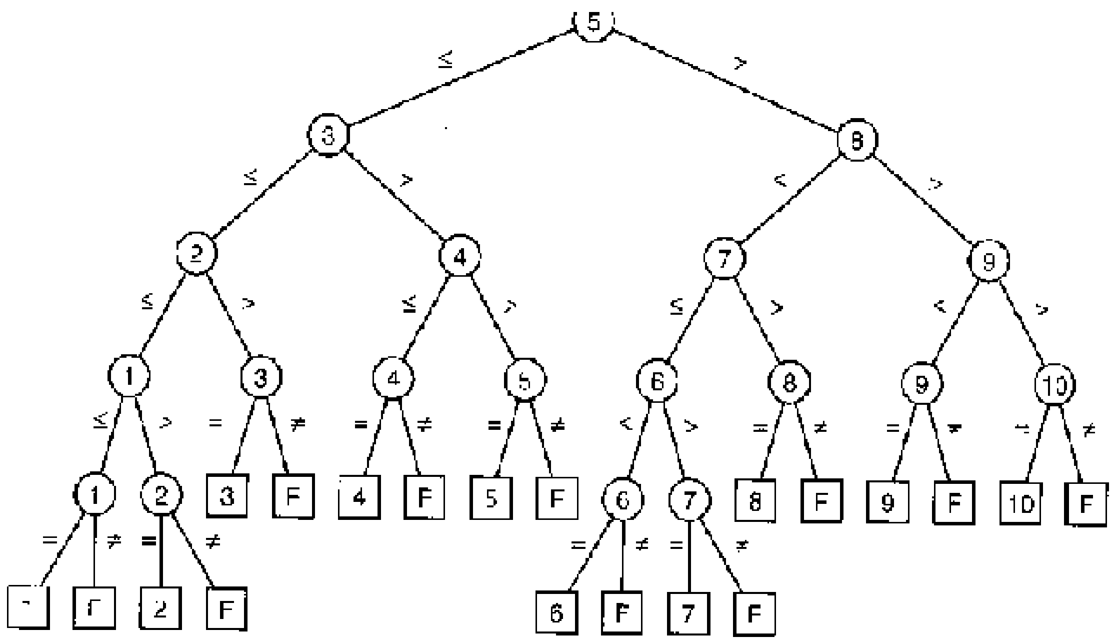
The figure below shows the decision tree for the sequential search algorithm on a list of n elements. [This shows the vertex numbers as 1 through n; in a C++ implementation, the array indices will be 0 through n-1.] Note that its shape is long and skinny, leading to lots of work for an unsuccessful case, or for successful cases where the target is found near the end of the list. You can see from the tree that the single failure node is at level n and so an unsuccessful sequential search requires n comparisons. The n various success leaves are at levels 1, 2, … n, so they require on the average

$$(1 + 2 + \ldots + n)/n = (n + 1)/2$$

comparisons. This agrees with our earlier analysis of sequential search.

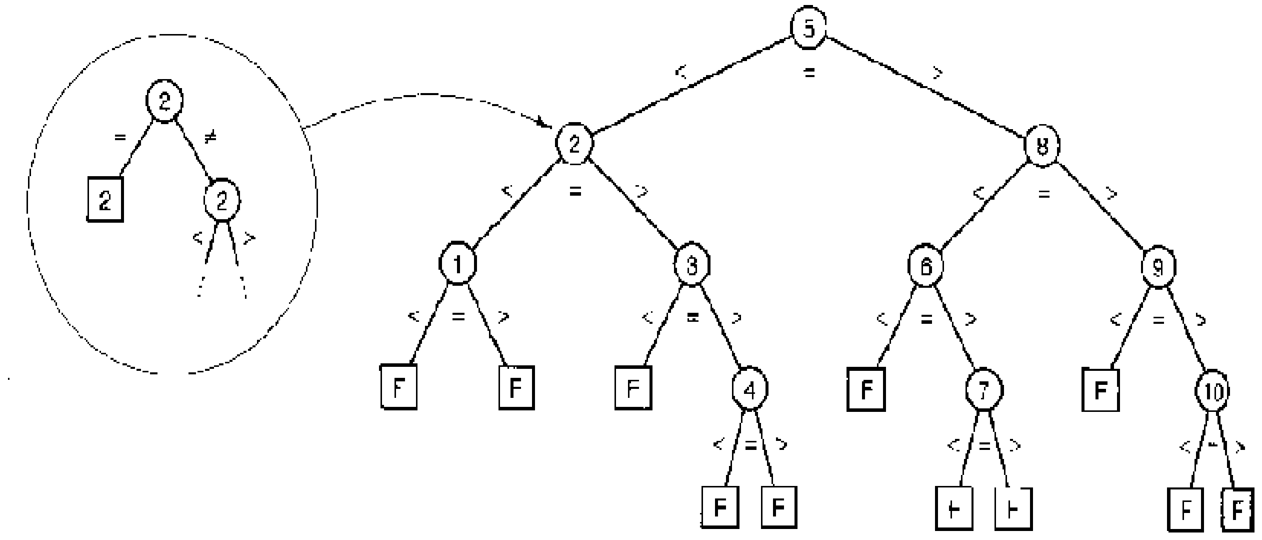


We will use decision trees to help analyze both the worst-case and the average-case behavior of the two binary searches. The two figures below represent decision trees for the particular case of n = 10 for the Binary1 and Binary2 algorithms. Because these trees are shorter and fatter than the sequential search tree, it's easy to guess that either binary search does less work in the worst case, and probably on the average, than sequential search. But how do these two algorithms compare to each other? Our intuition is that Binary2 is more efficient because it can recognize a target that occurs at a midpoint right away (as in this animation), whereas Binary 1 does not find the target until the end, when the list has been reduced to one element (as in this animation).



Binary 1, n = 10

Note that in almost all successful outcomes in the tree for Binary 1, there is a repetitive final comparison in the last node before a leaf. For example, just before leaf 3 there is a comparison that says the target equals the value at this node and we have a successful outcome, even though the target was already compared against this node higher up in the tree. At that earlier comparison, the target was merely found to be less than or equal to the node. This final compare occurs after the algorithm exits the loop that keeps cutting the list in half, i.e., it is the comparison made after reducing to a one-element list. The worst case for Binary1 on a 10-element list is the height of the tree, which is 5.



Binary 2, n = 10

In the decision tree for Binary2, each node represents a 3-way decision that identifies whether the target is =, <, or > the list value at this location. The little expanded picture shows that you can't get the result of a 3-way decision unless you do two comparisons, so each internal vertex of this tree requires two comparisons unless the target equals the value at that vertex, which is found in one comparison by doing the test for equality first. The worst case for Binary2 on a 10-element list occurs at the lowest failure outcomes in the tree. These are at level 4, but - since these are failures - there are two comparisons done at each internal node from the root to these leaves, hence the worst case is 2*4 = 8.

As we expected, these worst-case behaviors are better than the worst case of sequential search.

To compute the average amount of work done, we must consider both successful and unsuccessful cases. We will get specific values for the case n = 10 here, then later generalize to any n. We will use the formula for average work.

Binary1 **average successful search**	Each of the 10 successful cases occurs at a leaf. Compute the work for each case * the probability of that case occurring. Each case is equally likely, so the probability is 1/10. We'll factor that out of the sum. Moving left to right along the bottom of the tree: (2[leaves]*5[level of these leaves] + 3*4 + 2*5 + 3*4)*(1/10) = 44/10 = **4.4**
Binary1 **average unsuccessful search**	There are 10 failure cases that are exactly symmetric to the 10 successful cases, so the average work is the same, **4.4**
Binary2 **average successful search**	Here the successful cases occur at all the non-leaf nodes (see the little insert). Compute the path lengths to each of these nodes, beginning with the root, which is level 0: 1*0 + 2*1 + 4*2 + 3*3 = 19 There are 2 comparisons made at each node along this path. But there is 1 final compare AT that node to detect equality. So the total number of comparisons is 2*19 + 1*10 = 48, and each case has probability 1/10, giving average work of **4.8**.
Binary2 **average unsuccessful search**	Failure cases occur at all the leaves, with 2 comparisons done at each internal node on the path to the leaf. The total number of comparisons (again moving left to right in the tree) is 2*(3*3 + 2*4 + 1*3 + 2*4 + 1*3 + 2*4) = 78 But there are 11 different ways to fail, not 10, so the probability of each case is 1/11 and the average work is 78*(1/11) = **7.1**

These results show that - for n = 10 - Binary1 is on the average more efficient than Binary2 for both successful and unsuccessful searches, contrary to our intuition that Binary1 seems to do a lot of unnecessary work. Can we generalize this conclusion to an arbitrary n, where we can't draw a specific decision tree?

DEMONSTRATE: Shortest path problem

(from Gersting, *Mathematical Structures for Computer Science,* 6 ed, W.H. Freeman, 2007)

In this notation, IN is the set S, d is the distance array, s is the "parent" array, and p stands for the node about to be brought in (the minimum distance node). The source node is x and the destination node is y.

Consider the graph in Figure 6.7 and the corresponding modified adjacency matrix shown in Figure 6.8.

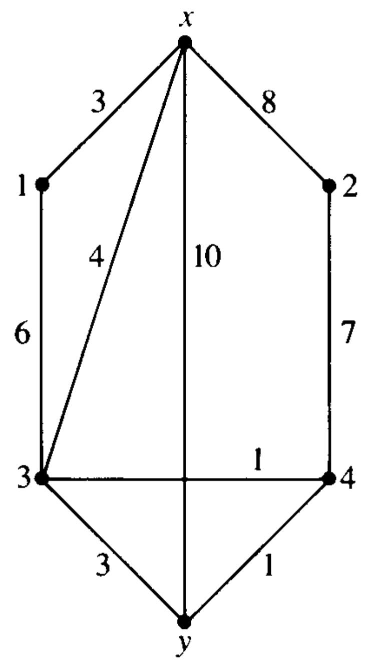


Figure 6.7

$$\begin{array}{c|cccccc} & x & 1 & 2 & 3 & 4 & y \\ \hline x & \infty & 3 & 8 & 4 & \infty & 10 \\ 1 & 3 & \infty & \infty & 6 & \infty & \infty \\ 2 & 8 & \infty & \infty & \infty & 7 & \infty \\ 3 & 4 & 6 & \infty & \infty & 1 & 3 \\ 4 & \infty & \infty & 7 & 1 & \infty & 1 \\ y & 10 & \infty & \infty & 3 & 1 & \infty \end{array}$$

Figure 6.8

We shall trace algorithm *ShortestPath* on this graph. At the end of the initialization phase, *IN* contains only x, and d contains all the direct distances from x to other nodes:

IN = {x}

	x	1	2	3	4	y
d	0	3	8	4	∞	10
s	–	x	x	x	x	x

In Figure 6.9, circled nodes are those in set *IN*, heavy lines show the current shortest paths, and the d-value for each node is written along with the node label. Figure 6.9a is the picture after initialization.

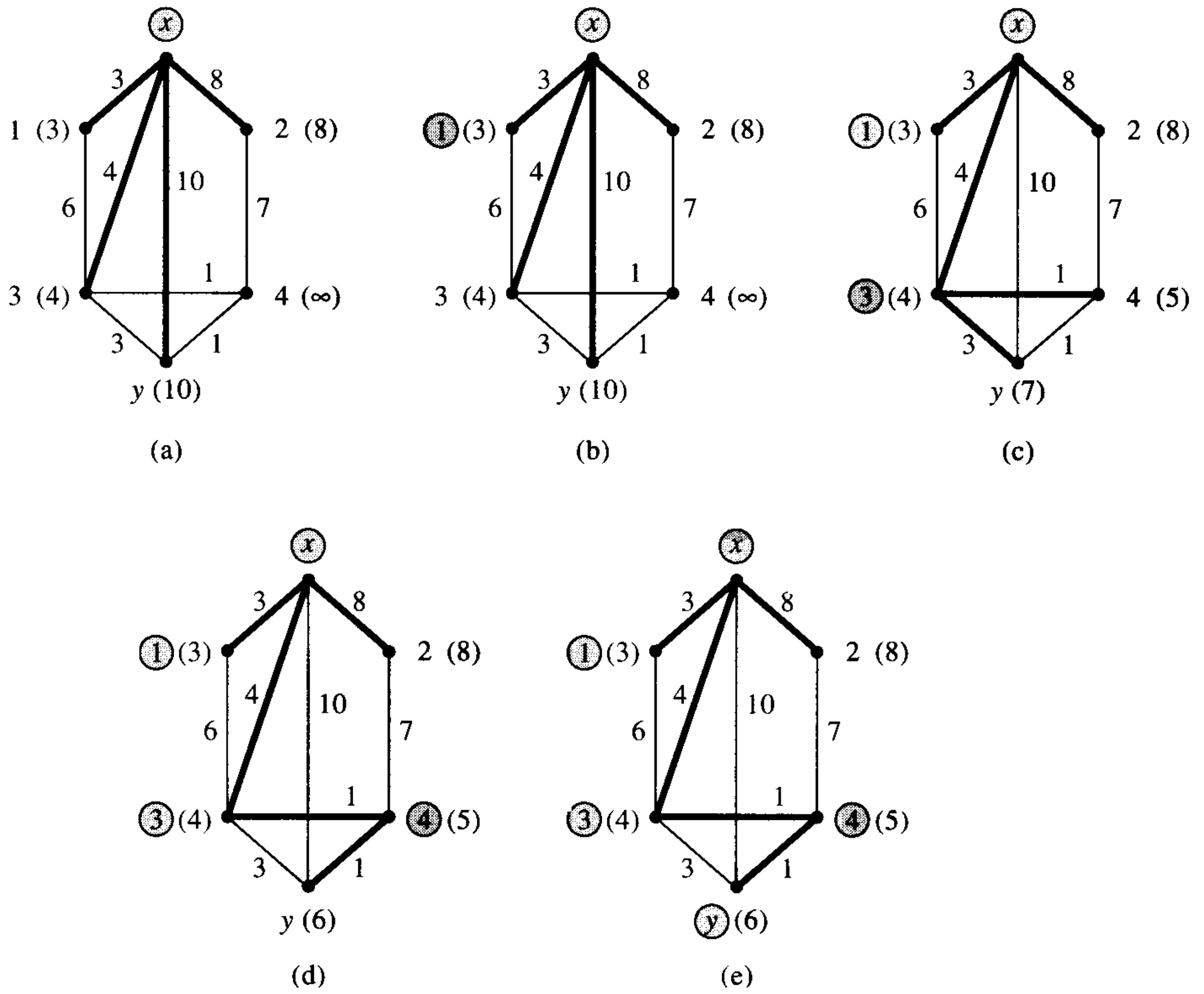


Figure 6.9

We now enter the loop and search through the d-values for the node of minimum distance that is not in *IN*; this turns out to be node 1, with *d*[1] = 3. We throw node 1 into

IN, and in the **for** loop we recompute all the *d*-values for the remaining nodes, 2, 3, 4, and y.

$p = 1$
$IN = \{x, 1\}$
$d[2] = \min(8, 3 + \mathbf{A}[1, 2]) = \min(8, \infty) = 8$
$d[3] = \min(4, 3 + \mathbf{A}[1, 3]) = \min(4, 9) = 4$
$d[4] = \min(\infty, 3 + \mathbf{A}[1, 4]) = \min(\infty, \infty) = \infty$
$d[y] = \min(10, 3 + \mathbf{A}[1, y]) = \min(10, \infty) = 10$

There were no changes in the *d*-values, so there were no changes in the *s*-values (there were no shorter paths from *x* by going through node 1 than by going directly from *x*). Figure 6.9b shows that 1 is now in *IN*.

The second pass through the loop produces the following:

$p = 3$ (3 has the smallest d-value, namely 4, of 2, 3, 4, or y)
$IN = \{x, 1, 3\}$
$d[2] = \min(8, 4 + A[3, 2]) = \min(8, 4 + \infty) = 8$
$d[4] = \min(\infty, 4 + A[3, 4]) = \min(\infty, 4 + 1) = 5$ (a change, so update *s*[4] to 3)
$d[y] = \min(10, 4 + A[3, y]) = \min(10, 4 + 3) = 7$ (a change, so update *s*[*y*] to 3)

	x	1	2	3	4	*y*
d	0	3	8	4	5	7
s	–	*x*	*x*	*x*	3	3

Shorter paths from *x* to the two nodes 4 and *y* were found by going through 3. Figure 6.9c reflects this.

On the next pass,

$p = 4$ (d-value = 5)
$IN = \{x, 1, 3, 4\}$
$d[2] = \min(8, 5 + 7) = 8$
$d[y] = \min(7, 5 + 1) = 6$ (a change, update *s*[*y*])

	x	1	2	3	4	*y*
d	0	3	8	4	5	6
s	–	*x*	*x*	*x*	3	4

See Figure 6.9d.

Processing the loop again, we get

$p = y$
$IN = \{x, 1, 3, 4, y\}$

$d[2] = \min(8, 6 + \infty) = 8$

	x	1	2	3	4	y
d	0	3	8	4	5	6
s	–	x	x	x	3	4

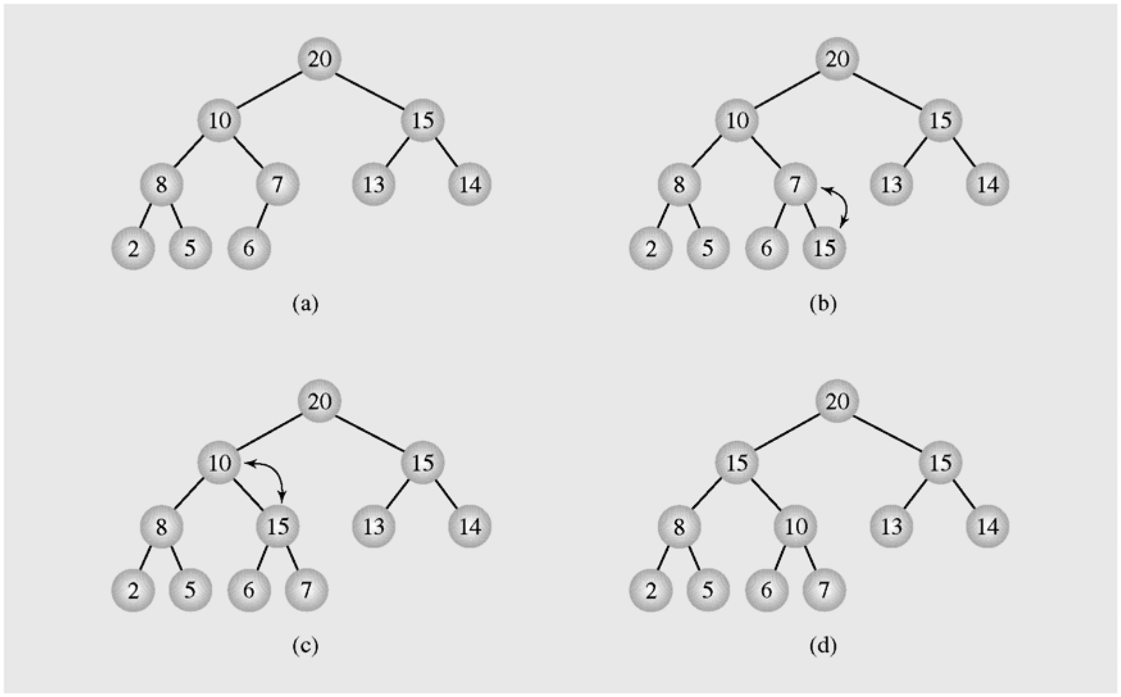
See Figure 6.9e.

Now that y is part of IN, we could terminate the loop, or just keep going and bring 2 into IN as well. The path goes through y, $s[y] = 4$, $s[4] = 3$, and $s[3] = x$. Thus the path uses nodes x, 3, 4, and y. (The algorithm gives us these nodes in reverse order.) The distance for the path is $d[y] = 6$. By looking at the graph in Figure 6.7 and checking all the possibilities, we can see that this is the shortest path from x to y.

Enqueue into heap/priority queue

The following figure is used by permission of Cengage Learning from Data Structures and Algorithms in C++ by Adam Drozdek.

FIGURE 6.54 Enqueuing an element to a heap.



File Structures Overview

The last part of CSCI 36200 is concerned with file structures - issues that arise when dealing with external storage. Up to this point we have largely ignored such issues, but in the real world, big files are the norm.

File structures used to be taught as a separate course in many universities, but it is now often covered as part of a data structures course (as we are doing here) because it deals with many of the same problems - searching and sorting - that we have been studying. The difference is that up to now we have assumed (except for our brief discussion of B+ trees) that the file is small enough that its contents can be stored entirely in main memory. Another place where file structures is taught is at the beginning of a database course because a database can be viewed as a sophisticated file structure.

.

File Structures, 1

For the last part of the course we will take an overview look at file structures. This will include:

- Different file organizations (sequential, relative, and indexed)
- The pros and cons of each organization and the difference between metadata and user-data CRUD (Create, Read, Update, Delete) rules for the different organizations
- The important features of external sort and merge algorithms

Introduction

Why do we need files? For several reasons:

1. Persistent storage. Computer memory is volatile and anything stored there is lost when you turn off the machine. Obviously no one wants to recreate documents, computer programs, data files, spreadsheets, etc., on the fly every time they are needed.
2. Limitations of computer memory. Despite the great increases in RAM storage, the amount of information that can be brought into memory is still limited, whereas the size of external storage is much greater.

So file storage is clearly a necessity. But accessing data in a file to move it into main memory where it can actually be used is a slow process. We've already talked about this for **disk storage**, where **seek time** (to move the read head over the correct track), **latency** (to move the correct sector under the read head) and **transfer time** (time for the actual data transfer) make this operation many times slower than access to an internal memory location [think of a record player – if you've never seen one, look it up!]. External storage could also use **magnetic tape**; tape is still the least expensive medium for archival storage and it is highly reliable. Time must be spent advancing the tape to the proper location for the desired data item [think of an audio tape player, popular before the days of MP3 players and iPods!].

From *File Structures* by Folk, Zoellick, and Riccardi, Addison-Wesley, 1998: "Assume that memory access is like finding something in the index of this book. Let's say that this local, book-in-hand access takes 20 seconds. Assume that accessing a disk is like sending to a library for the information you cannot find here in this book. Given that our "memory access" takes 20 seconds, how long does the "disk access" to the library take, keeping the ratio the same as that of a real memory access and disk access? The disk access is a quarter of a million times longer than the memory access. This means that getting information back from the library takes 5 million seconds, or almost 58 days." Quite a difference!

So here is the inherent conflict in files – slow access time (bad) vs. nonvolatile, large capacity (good). The central task, then, is to minimize the number of disk accesses to get the information we need from the file.

As we know, all data is stored in computer memory in binary form. Data in a file is also in binary form, but the file can be interpreted as either a text (ASCII) file or a true binary file. Suppose you want to store the integer 29 in a file. In an ASCII file, this will be stored as the character 2 followed by the character 9. This would nominally require 2 bytes (8 bits per character). When the contents are viewed through some sort of text editor (i.e., Notepad), you see 29. The file is "human-readable." The 8-bit binary representation of the integer 29 (1*16 + 1*8 + 1*4 + 0*2 + 1*1) is 00011101, so this could nominally be stored in 1 byte in a binary file, although it would actually be stored in 4 bytes (on a 32-bit word machine). If you try to view the contents through some sort of text editor, you see strange symbols. The file is only "machine readable." We'll assume we are dealing with ASCII files.

In our look at Common Data Structures early in the semester, we imagined an ADT "list" and considered the desired operations on list elements:

• Insert a new list element	(Create)
• Access a list element	(Read)
• Change the value of a list element	(Update)
• Delete an existing list element	(Delete)

We then saw how three implementations of the list ADT (array, vector, linked list) handled these operations with greater or lesser difficulty. These four operations, often called CRUD rules, apply to files as well, in two dimensions. At the data level, we want to know how to create a new file entry (item of data), how to read (access) a file entry (this is what we want to optimize), how to update a file entry and how to delete a file entry.

The term **metadata** means "data about data". If we think about files as data containers, then metadata CRUD rules can apply to files themselves. At the metadata level, how do we create a new file, read information about a file, update some information about a file, or delete a file? The operating system will help your program to create a file (through open and close statements) and to delete a file. The operating system actually keeps metadata – data about files. For example, when the file was created, when it was last modified, the file size on the disk, etc. The most important metadata maintained by the operating system is the address in the physical secondary storage medium (tape or disk) for the beginning of the file. On early Windows operating systems this information was kept in the aptly named File Allocation Table (FAT); later Windows systems use New Technology File System (NTFS), and other operating systems have their own file management systems. The operating system will not, in general, allow you to read (access) or update information about a file using a higher-level programming language; you can't, for example, change the physical address of the file on your hard drive from within a program, nor the date of its creation.

As a simple example, you can write a little text file in Notepad and save it to the disk. You are asked for the file name you want to save it as. Voila - you've created a file! Within C++ you create a text file by

```
      ofstream outfile;
      outfile.open("MyFile.txt");               //metadata-level Create rule

      //write to the file
      for (int i = 0; i < 10; i++)
      {
            outfile << i << endl;               //data-level Create rule
      }
      outfile.close();
```

and read it by

```
      ifstream infile;
      infile.open("MyFile.txt");
      int a[10];
      //read from the file
      for (int i = 0; i < 10; i++)
      {
            infile >> a[i];                     //data-level Read rule
      }
      infile.close();
```

and now you decide it's a pretty useless file, so you delete it

```
      if( remove( "MyFile.txt" ) == -1 )        //metadata-level Delete rule
        cout <<"Error deleting file" << endl;
      else
        cout <<"File successfully deleted" << endl;
```

One of the nice things about stream classes in C++ is that (once the stream is "hooked to" the file through the open statement), reading data from or writing data to a file looks no different from reading data from a keyboard or writing data to the screen. In some programming languages, communicating with a file is very different than keyboard input or screen output.

So again, we will be looking at different file organization structures and how they handle CRUD rules, mostly at the data level. At the metadata level, regardless of the file organization, we are in general limited to Create and Delete.

Sequential Organization

A sequential organization is characterized by the fact that the items stored in the file are consecutive. The first item is at the beginning of the file, then the next item, and so forth to the end of the file. Usually the only way to access an item in a sequential organization is to “process” all the items before (in front of) that item in the file. The IBM term for such a file is an **Entry Sequenced Data Set** (ESDS). This is a very descriptive term; it indicates that the only arrangement of information on this type of file organization is the way in which the data were entered.

There are two types of sequential files, stream files and record-oriented files. Stream files are the more primitive. The distinction is in whether or not there are delimiters recognizable by the I/O operations that divide the file up into physical records. Stream files have no delimiters, while record-oriented files include an end-of-record (EOR) delimiter (at least conceptually).

Stream Files

A stream file consists simply of a stream of bytes or data items. From the user's point of view there is some kind of a "pointer" maintained by the system (low level operating system I/O routines) that always points to the next location from which data is to be read or to which data is to be written. Of course, when the file was opened this pointer was set to point to the beginning of the file. Each I/O operation has the side effect of automatically moving the pointer to point to the next location. When the file is closed the pointer is no longer needed. Also, if the last I/O operation was a write, it is often the case that some special sentinel value is written to mark the end of the file (i.e., an end-of-file, EOF, mark).

There are no logical field or record delimiters within the stream, just a beginning and an end. The user's code must "parse" or break up the stream into meaningful chunks.

C++ provides operations at the stream level. In C++, a file is modeled as a linear array of bytes on the disk. The open function establishes the connection to the file, and sets the "pointer" to point to the first byte in the file. Functions like *get* (or *put*) read (or write) a single byte (character) starting from the pointer. These functions "advance" the pointer. The functions signal exceptional conditions, for example, when an end-of-file is encountered or a write operation has failed. Consider the following code that reads through a stream file (called STREAM.DAT) of characters looking for the symbol "(".

```
#include <fstream >
#include <iostream>
using namespace std;

void main()
{
  char c;
  ifstream file_handle;
  file_handle.open("STREAM.DAT");
  file_handle.get(c);
  while (!file_handle.eof())
  {
     if (c == '(')
     {
        cout << "found the (" << endl;
        break;
     }
     file_handle.get(c);
  }
}
```

The *get* function assigns the character read to c. The break statement within the loop ends the while loop if the value of c is "(". The *eof* function returns the value true if the end-of-file marker has been read.

Languages like COBOL and FORTRAN have no stream I/O capabilities; they are record oriented languages (see the discussion that follows).

Record-Oriented Files

The main difference between a stream file and a file composed of physical records is that a special delimiter, called an end-of-record indicator (EOR), is defined. The data between two successive EORs (or the beginning of the file and the first EOR) is called a record. Records may be of any length. If all the records in the file are the same length it is called a fixed-length record file, otherwise it is called a variable length record file.

Often the record delimiter is a pair of characters, a carriage return character (ASCII 13) and a line feed character (ASCII 10). This pair is often represented by the symbol CRLF (pronounced "curlif"). Most text editor programs, such as Notepad, place a CRLF in the file each time the <Enter> key is pressed. This is a carry-over from the functionality of a typewriter. On a typewriter when the <Return> key is pressed the mechanism carries out a carriage return (moving the printing point to the left edge of the paper) and a line feed (moving the paper up one line). However, pressing <Enter> on a computer keyboard has whatever effect the program assigns to that key, not a fixed action like CRLF - e.g., in Microsoft Word, <Enter> causes a paragraph mark to be added to the document; in Excel <Enter> usually causes a move to the next cell.

It is still the case that records are just a sequence (or stream) of bytes. There is no "internal structure" (no fields) on this group of bytes. The user's program must provide the instructions to parse the records into fields. The exact details of how this is accomplished depend on the programming environment.

Assuming a text file with CRLF as the only record delimiter, C++ can do a record-at-a-time read by using the *getline* function. But once the record has been read, the program itself must parse the record, picking it apart based on the known (to the programmer) structure of the file.

The following C++ program reads the file *data.txt*, which looks like this:

```
123456jack10.3
896321bill12.8
```

```
#include "utility.h"

int main()
{
       string filename = "data.txt";
       string stuff;
       int intone;         //4-digit integer
       int inttwo;         //2 digit integer
       string title;       //4-character string
       double dblone;      //4-character decimal number

       ifstream in;
       in.open(filename.c_str());
       if (in.fail())
       {
              cout << "Failure to open input file" << endl;
              exit(1);
       }
       while(!in.eof())
       {
              getline(in,stuff);
              cout << stuff << endl;
              intone = atoi( stuff.substr(0, 4).c_str());
              cout << intone << endl;
              inttwo = atoi(stuff.substr(4,2).c_str());
              cout << inttwo << endl;
              title = stuff.substr(6,4);
              cout << title << endl;
              dblone = atof(stuff.substr(10,4).c_str());
              cout << dblone << endl;
       }
       in.close();
       in.clear();

       return 0;
}
```

In the above program, the specifications (metadata) for the layout of the fields in the record is held in the source code.

In languages like COBOL and FORTRAN where the I/O operations are record oriented, the file must contain EORs and each I/O operation processes one full record. There is a syntax that allows the user to specify the parsing instructions (metadata for the field layout).

Consider the following FORTRAN code that reads up to 15 records from a file called DATA.DAT that contains records consisting of a 5 digit integer and a 20 character string and a 3 digit integer. The program displays the first number and the string on the screen (the last number is not read or processed)

```
      INTEGER N
      CHARACTER * 20 S
      OPEN(UNIT=1,FILE='DATA.DAT')
      DO 100 I=1,15
      READ(UNIT=1,FMT=5000,END=200) N, S
 5000 FORMAT(I5,A20)
      WRITE(*,FMT=5001) N, S
 5001 FORMAT(1X, I5, 2X, A20)
  100 CONTINUE
  200 CLOSE(UNIT=1)
       END
```

The points of interest here are:

- The unspecified options in the OPEN statement default to a sequential character file.
  The OPEN statement without taking the defaults would have been:
  OPEN(UNIT=1, FILE= 'DATA.DAT', ACCESS= 'SEQUENTIAL', FORM= 'FORMATTED', STATUS = 'OLD')
- The READ statement references unit 1 (the equivalent of the input stream in the C++ program) and uses FORMAT statement number 5000.
- FORMAT statement 5000 contains the parsing "instructions" to convert the character (byte) information in the file into the binary form needed by the memory variables N and S mentioned in the I/O list
- The WRITE statement references the screen, unit *, and FORMAT 5001.
- FORMAT 5001 adds a space at the left of the number (1X) and two spaces between the output items (2X).
- Each READ reads one physical record, each WRITE writes one physical record.
- The READ statement starts processing information in the file at the current location of the "pointer", parsing according to the FORMAT. Parsing stops at the end of the FORMAT statement description, in this case at the end of the character string. The "pointer" is advanced beyond the CRLF (skipping the 3 digit number) since the whole record was read even if only part of it was used (parsed).
- Rewording the previous statement: FORTRAN has no stream capability whatsoever. Only complete records are processed even if only part of the information in the records is used (i.e., the first two fields, ignoring the last field). FORTRAN was designed to read a punched card, and the whole card had to be run through the card reader.

A text file may contain other record delimiters. For example, if the data.txt file for the above C++ program has the following structure

```
2561    85          Help    15.4
```

then it could be read by

```
in >> intone >> inttwo >> title >> dblone;
```

The extraction operator skips whitespace until it finds a character that represents the beginning of an integer, and it stops reading characters for that integer at a blank space or a non-digit character because these serve as a "record delimiter" for integer data. Similarly the extraction operator for string data skips whitespace looking for any nonblank character (which represents the beginning of a string) and stops reading characters for that string at a blank space. Here the programmer has an easier task to "parse" one line of the file. But note that if you wanted a program to read first names, you could not use a single extraction operator; although this would work fine for "Fred", for a name such as "Mary Lou", you would only get Mary. Here's where you have to use getline and be sure you know how long the "name field" is, which must be the same for every record.

In summary, even for a particular file organization, e.g., a sequential organization, different programming languages implement the details of the file access operations differently.

Record-level CRUD Rules (Sequential Organization)

Since a sequentially organized file is usually stored on a sequential device, e.g., magnetic tape, the data level CRUD rules are limited to "append" and Read the next item in the sequence. Update and Delete require rewriting (copying) the file with the appropriate changes and/or deletions.

Create

In a sequential organization new items (additions to a stream or additional records) are created at the end of the file. The file is extended as items are appended to the file.

Since the arrangement of the file is entry order, the newest item is the last item in the file. If some other arrangement is desired, for example, sorted order on some part of the record, the Create operation will not be sufficient since Create can only append items to the file. The file will have to be copied over while inserting the new record in the proper position.

Read

The Read operation in a sequential file is interpreted as "read next", that is, read the next record in the sequence. The "read next" operation implies a "forward" operation. Some languages provide the opportunity to "backspace", that is, move toward the beginning of the sequence and "rewind" to get back to the start of the file.

Update

Update in a sequential organization is a problem. It is not reliable to overwrite a portion of a file on a truly sequential medium (e.g., magnetic tape). The storage device may decide that the portion of the medium onto which the information is to be written is no longer of sufficient quality to be used. The device may then mark that part of the medium as bad and move ahead and write the information on the next available "good" portion of the medium, possibly destroying some other data. The appropriate way to update information in a sequential organization is to copy the original file over into a new file, making whatever updates are needed.

If the file is stored on a disk drive it is physically possible to update an item by overwriting the item. From a design view this may not be a good idea since no backup is available, that is, it is not possible to undo the operations if the system or program fails part of the way through a file update process.

Delete

The only way to delete an item from a sequential file is to copy the file, leaving out the items that are to be deleted.

Text Files

An important application of sequential files is a text file. A text file is an automated version of the typewritten document. Variable length records delimited by CRLFs are presented to the user in entry sequence order.

Programs called text editors have been developed to manage the record level CRUD rules for this type of file. The editors allow the user to Create (append) new records to the text file just as if the document were being typed. Of course the Read operation simply reads the records in order and displays them on the screen (for the user to read).

With a typewritten document, the Update and Delete process required complete retyping of the document. The text editor has automated this process. The automation is often based on reading the text file and storing the text in the memory of the computer, that is, changing the organization from a sequential organization to a random access organization. This is the way Notepad attacks the problem.

What if the file is larger than can be stored in RAM? Instead of copying the file into RAM, programs that deal with large files often make a "working copy" of the file with an organization designed to support the Update and Delete process. When the editing session is complete the working file is written out as an entry sequenced data set, i.e., a sequential file.

Most programming language development environments require the programmer to use a text editor to enter the source code statements. An editor may also be used to construct input data files for a program to use. The program may produce printable output. The compiler and/or linker produce report files. These are all sequential text files where only Create (append) and Read operations are employed.

File Structures, 2

Sequential Organization

The topic is still sequential file organization, typically text files. Two important algorithms for processing sequential files will be discussed.

Master File Update (Balance Line Algorithm)

Data processing is a term used to denote manipulation of large amounts of data where the emphasis is primarily Input/Output oriented with a minimum of computation (as opposed to computing missile trajectories to the moon which has many computations and little input). There are multitudes of computer applications that fall into the category of data processing, e.g.,

- checking and saving account records in a bank
- billing operations of the power and light or gas companies
- premium payment records of an insurance company
- tax records of the IRS

All these applications have something in common; they depend on a "data-store", that is, a quantity of information about company activities, for example, a list of customers and what services they have used.

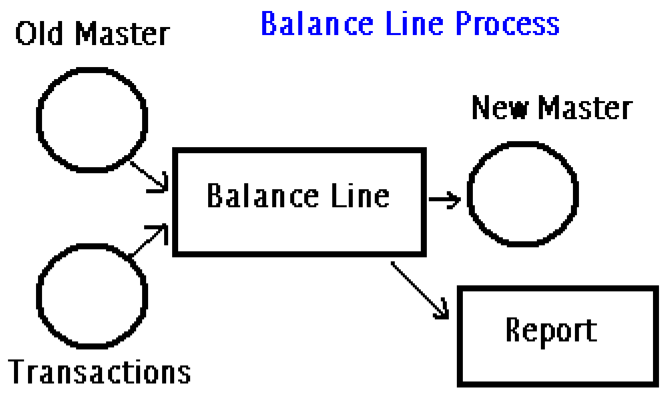
The "data-store" in a data processing application usually contains items known as a Master File and a Transaction File and sometimes other files. Each of these files is composed of entries known as records. The master file records usually contain the most up-to-date information about that entry; for example, a bank master file record might contain items such as account number, account name, address, social security number, current balance, year-to-date interest and possibly other things. The master file should have one record for each account, and the records would be arranged within the file in some specific order, say in order of increasing account number. The transaction file is just what its name implies, a set of transactions or activities over some period of time (such as one day), one record per activity. A transaction record for a bank might contain items such as account number, and the amount of a deposit or withdrawal. The transaction file will not contain a record corresponding to each master record (each bank account is not used each day). The problem is to incorporate the information contained in the transaction file into the master file. This is called "updating" the master file, and requires a merge operation.

The process called the **balance line algorithm** is a standardized approach to updating sequentially ordered master files by one or more transaction files ordered in the same sequence. This process is run in a "batch" mode, that is, all in one batch. It would not make sense to use this process in an interactive environment since, as will be seen, the entire file would have to be processed to make a single change. The performance of such a process would most likely not be acceptable.

The balance line algorithm requires at least three files: a transaction file, an original master (often called the old master) and a new master. The new master is the updated version of the old master. How are these files created? The old master is the new master from the previous execution of the process (or an empty file for the first execution). The transaction file might be created using a text editor, or by some other application program. There are several reasons why the "old" master is not updated directly:

- If the old master is on magnetic tape, it is not possible to "rewrite" a record (on a disk this may be done, but see the items that follow).
- If the old master were updated directly and it was discovered that the update was incorrect, the update would have to be "undone" before the correct run could be made.
- If a hardware or software failure occurs during the run, one is left with a partial update, leading to inconsistencies in the data.
- One always wants a "back-up" file so if a tape breaks or is lost you aren't "out of business".

Thus, during an update the program copies the old master onto the new master while incorporating the information in the transaction file. It is also reasonable to expect some kind of printed output to be produced so the user of the program can "see" that everything is correct. Printed output may be items such as bills, checks or other types of forms. When the update is complete, the old master becomes a back-up file and the new master is ready for use as the "old" master for the next set of transactions.



The old master and transaction files must be sorted (see the next section) in ascending order on the same key field (part of the record). The key in the transaction file is called the transaction key and the one in the master file is called the master key.

Transactions to be applied to the old master consist of three types: additions of new records, deletions of existing records, and updates to existing records. A single transaction can affect only one old master file record, but several transactions may be applied to the same old master record. Transactions are applied only when the transaction key value matches the master key value. In essence the balance line algorithm as described here is an implementation of the record-level CRUD rules for a sequential organization. The process is basically a copying (Read) operation with changes (Create, Update, Delete) driven by the transaction file. The major task of the balance line algorithm is to synchronize the interaction of the master and transaction records so that the appropriate actions can be performed.

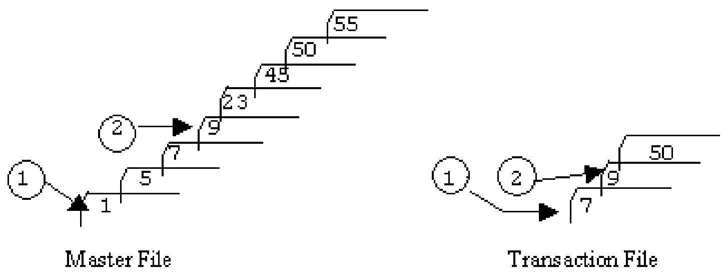
The process of merging two sequential data files requires the following information:

- The location of the data files
- The exact layout of the records that make up the files
- The field (known as the sort key) on which each file is sorted (so that the comparison may be made). Remember that a record contains no metadata information; it is too dumb to know anything about some special field.
- The transaction sort key should be used as the indicator that master file records should be read (except for the first master record which must be read at the start)

Simplified Example

Consider a simplified example composed of the two files shown below. The simplification is that the transaction file only contains Update data, so no new records will be added and no old records will be deleted. And this example does not create a new master file. In a database when you are "combining" two "files", this simplified algorithm (walking over two sorted lists and matching up key values, called a **join** operation) is very typical of the kind of process used - take CSCI 443 next fall!.

Back to our example:



Merge Process

The numbers on the records are sort key values. At the start of the merge process one record is read from each file. The arrows marked (1) indicate the records just read. The transaction with key of 7 has been read. The process compares the transaction key value with the key value of the first master record read. (Note: If the master key value is ever greater than the transaction key value, an error has occurred.) Because the master key is not equal to 7, the next master record is read. Again, no match - another master record is read, this time giving a match of key values (both are 7). The transaction record with key value 7 is processed. The (2) arrows indicate the next record in each file. Since the last transaction record has been processed, a new transaction record is read with a key value of 9 and the search for a match from the master file is continued. (Keep in mind that there might be two transaction records with a key value of 7, so don't be in too big a hurry to advance through the master file.)

When the end of the transaction file is reached, the rest of the old master file is copied onto the new master file. This may be accomplished with the same search process described above by setting (when the end of the transaction file is reached) the transaction key to a value larger than could ever occur as a master key value, so records will continue to be read from the master file in an attempt to "catch up" to the transaction key value. The end of the master file terminates the merge process.
End of Simplified Example

The "report" shown in the figure could be an exception report (indicating only problems, e.g., updates with transaction key values and no matching master key value, out of order keys, ...) or a full listing of the activities that were carried out by the process (e.g., printed values for the old master, transaction and new master), or something in between.

As noted, both the old master file and the transaction file must be sorted in key order. It's time to talk (again) about sorting.

Sorting

Sorting has to do with the **order** of items, not their organization. Sorting is a generic process and not limited to any particular file organization. We've spent quite a large part of the semester on algorithms for sorting an array of data items in main memory (**internal sorting algorithms**). Now we need approaches for sorting a file too large to be brought into main memory all at once (**external sorting algorithms**).

Sorting consists of ordering the records in a sequence based on the values of some part(s) (i.e., fields) of each record. The part(s) is called the sort key. As discussed previously, the exact description of the sort key is not part of the record but is part of the program involved in the sorting process (there is insufficient metadata in the sequential organization to "remember" the description of the sort key).

Sorting is discussed here in the section concerned with sequential file organization for two reasons: when the items to be sorted cannot be retained in memory (i.e., there are too many to be sorted internally) files are needed. The files that are needed are often ordered, that is, already sorted, so that their "entry order" is their "sorted order". Sequential files are best for this type of application.

First the ideas of merge/sort are discussed. Then a few comments are made about sort packages. Sort techniques involving other file organizations (relative and indexed) will be discussed later.

Merge/Sort

The task to be considered here begins with a set of items that are to be sorted, that is, placed into some specified order. It is assumed that these items are stored in a file (on a persistent medium). It is also assumed that there are a large number of these items. The question is how to get these items into order?

Nearly all techniques for file sorting involve the following three steps:

1. An internal sort is used to produce a number of sorted sublists, called runs or partitions, from the original input file.
2. A merge process is used to combine the sorted runs into a single sorted run.
3. An output step copies the sorted run to its final destination (file).

Several comments are in order. Step 3 is usually trivial as you will end up with one sorted file. Step 2 is, as we will see, virtually the same process used in the internal merge sort where two sorted sublists were merged into one sorted list by walking through the two sublists and picking the smaller of the two current values, one from each list. Notice that memory size is not an issue here. Even if you have two (or more) giant sorted sublists, you only have one element from each sublist in RAM at any one time.

Step 1 is the starting point for the overall sort process. A way is needed to build the initial sorted files that are to be merged.

There are many possible internal sorting processes that can be used for Step 1 (some were discussed earlier this semester). Step 1 amounts to repetitively reading some of the items from the file into RAM, sorting them using an internal sorting technique, and writing the result to other files. Surprisingly, one of the more important issues is the size of the runs and how many to have on each of the working tapes. The limit here is, of course, the size of RAM. The runs need not be of equal length for the merge processes to function properly.

There are two ways to illustrate the details of these sort/merge processes. One involves specific items (numbers) that are to be sorted, the other deals with runs and describes the algorithms in terms of these runs. The former shows detail, the latter emphasizes the large size problem. Both are important and both will be used in this discussion. The first approach (showing specific numbers) will be used in the example that follows, but think of each number shown below as hundreds or thousands of items (too many to sort all at once internally). The starting point for the examples is the 20 items:

57 17 30 69 24 53 50 95 92 15 73 89 43 70 62 37 40 56

Assume these items are stored on a sequential device, for example, a magnetic tape. (The purpose of this last comment is to say that **only** sequential file operations are going to be used in the processes that follow!)

First carry out Step 1. Suppose Step 1 results in the following set of 5 runs. (The symbol "|" is used to separate the runs.) Note that each run is sorted with low value first.

17 24 30 57 69 | 50 53 | 15 59 73 89 91 95 | 43 | 37 40 47 56 62 70

Different length runs have been used on purpose. The two methods to follow begin with these runs.

Natural Merge (M input files, 1 output file)

A merge that handles two input files at once is called a two-way merge; a merge that handles M input files at once is called an M-way merge. A natural merge merges M input files into one output file. Of course, the input files contain sorted runs (with possibly more than one run per file) and the output file is sorted in the same order. This process is very similar to the balance line algorithm discussed above, where each input file had only one run (the old master file was in order as was the transaction file) and only one output was expected (the new master file).

How is the sample data sorted? Consider a 2-way merge with the following distribution (the 5 runs from the sample data have been copied onto two tapes):

Tape 1: 17 24 30 57 69 | 50 53

Tape 2: 15 59 73 89 91 95 | 43 | 37 40 47 56 62 70

The first merge takes the first run from Tape 1 and merges it with the first run from Tape 2, writing the results to Tape 3. Then the second runs are merged. The third run is just copied from Tape 2 to Tape 3. The result after the first merge would be:

Tape 3: 15 17 24 30 57 59 69 73 89 91 95 | 43 50 53 | 37 40 47 56 62 70

To merge again, the information on Tape 3 must be distributed into two input files. Here, instead of doing a complete copy, half of the runs (1 in this case) are copied to another tape and the remaining are used from the original tape. So, Tape 1 is used for the second input (the first run from Tape 3 is copied to Tape 1) and Tape 2 is used for the output, thus:

```
Tape 1:   15  17  24  30  57  59  69  73  89  91  95
          ^
Tape 3:   15  17  24  30  57  59  69  73  89  91  95  |  43 50 53  |  37  40  47  56  62  70
                                                         ^
```

The ^ marks the location of the first read operation. The result of this merge is:

Tape 2: 15 17 24 30 43 50 53 57 59 69 73 89 91 95

To finish the process (in this case) use Tape 2 and Tape 3 as input with Tape 1 as output:

```
    Tape 2:   15  17  24  30  43  50  53  57  59  69  73  89  91  95
              ^
    Tape 3:   15  17  24  30  57  59  69  73  89  91  95  |  43  50  53  |  37  40  47  56  62  70
                                                                           ^
```

Then

Tape 1: 15 17 24 30 37 40 43 47 50 53 56 57 59 62 69 70 73 89 91 95

and the process is complete.

The natural sort/merge can require a significant amount of copying of the data between files. The greater the number of initial runs and the lower the degree of the merge (i.e., the smaller the value of M, meaning fewer tapes), the more I/O is required.

Balanced merge (M input files, M output files)

How can the copying or redistribution seen in the natural merge be reduced? The balanced merge avoids much of the copying by distributing the output into M files rather than one file (the natural merge requires a total of M+1 files, the balanced merge requires 2M files).

Again look at a 2-way balanced merge. This requires 4 tapes. For the example data this might be:

Tape 1: 17 24 30 57 69 | 50 53

Tape 2: 15 59 73 89 91 95 | 43 | 37 40 47 56 62 70

Tape 3:

Tape 4:

The first merge takes the first run from each input tape (1 and 2) and merges them onto Tape 3. The second runs on the input tapes are merged onto Tape 4. The unpaired run is copied to Tape 3. After the first merge pass:

Tape 1:

Tape 2:

Tape 3: 15 17 24 30 57 59 69 73 89 91 95 | 37 40 47 56 62 70

Tape 4: 43 50 53

Now the roles of input and output tape are reversed: 3 and 4 are input tapes, 1 and 2 are output tapes. The results after another pass are:

Tape 1: 15 17 24 30 43 50 53 57 59 69 73 89 91 95

Tape 2: 37 40 47 56 62 70

Tape 3:

Tape 4:

Finally:

Tape 1:

Tape 2:

Tape 3: 15 17 24 30 37 40 43 47 50 53 56 57 59 62 69 70 73 89 91 95

Tape 4:

Since each step combines two runs, the number of runs after each pass is about half the number for the previous pass. If R is the number of runs and p is the number of passes required to reduce the number of runs to 1, then

$$(1/2)^p \times R \leq 1$$

or, solving for p:

$$p = \lceil \log_2 R \rceil.$$

where $\lceil\ \rceil$ represents the ceiling function. In the above example, R = 5 and p = 3. For the M-way case this becomes

$$p = \lceil \log_M R \rceil .$$

so that the M-way balanced merge will require $\Theta(\log_M R)$ passes.

Polyphase Merge (M input files, 1 output file)

It is fairly easy to implement the balanced merge process. The 2M tapes are just switched back and forth from input to output. What about the case where "extra" runs are merged with "empty" runs (like the first pass in the example above)? Is it possible to take advantage of such a situation and reduce the amount of copying still further? Yes, but of course the algorithm becomes more complicated. A more important flaw in the balanced approach is that although 2M tapes are required, at any one time runs are being read from M files and being written to one file, leaving M-1 files idle.

The polyphase or multi-phase merge is an improvement over the balanced merge because it uses a total of only M + 1 tapes, just as the natural merge required, but with less copying than natural merge. The important issue here is how to distribute the initial runs onto the M input files.

Because of the complexity of this process, the second method of describing sort/merge algorithms will be employed here, that is, keeping track of runs and not the detailed items within runs. Consider again a case with M = 2, which will require 3 available tapes, and suppose there are 34 initial runs, distributed as follows:

Tape 1: 0 runs

Tape 2: 21 runs

Tape 3: 13 runs

The first merge processes runs in sets of 2, one from each tape, combining them and writing the resulting merged runs to the output tape (Tape 1). The process stops when one of the tapes is exhausted, in this case Tape 3. So, 13 sets of runs are merged creating 13 longer (merged) runs on Tape 1 and leaving 8 (of the original 21) untouched on Tape 2 The results of this merge are:

Tape 1: 13 runs

Tape 2: 8 runs

Tape 3: 0 runs

On the next pass, 8 runs are merged from Tapes 1 and 2 and written to Tape 3, resulting in

Tape 1: 5 runs

Tape 2: 0 runs

Tape 3: 8 runs

The next merge uses 5 runs from Tapes 1 and 3, writing the output on Tape 2.

Tape 1: 0 runs

Tape 2: 5 runs

Tape 3: 3 runs

Successive passes result in

Tape 1: 3 runs

Tape 2: 2 runs

Tape 3: 0 runs

then

Tape 1: 1 runs

Tape 2: 0 runs

Tape 3: 2 runs

and

Tape 1: 0 runs

Tape 2: 1 runs

Tape 3: 1 runs

and finally

Tape 1: 1 runs

Tape 2: 0 runs

Tape 3: 0 runs

at which point the final sorted output is on Tape 1. See how neatly this worked out, with 2 tapes used as input and 1 for output on each pass, and how efficiently one tape was emptied on each pass. Now look again at the run distribution numbers, excluding the empty tape:

{21, 13}, {13, 8}, {8, 5}, {5, 3}, {3, 2}, {2, 1}, {1, 1}, {1, 0}

These numbers pair represent successive values from the Fibonacci sequence! There is a total of 7 passes required, and it turns out [no proof] that this is optimal. No other distribution would result in fewer passes required to produce the final sorted output. These number sets constitute a *perfect Fibonacci distribution* of runs on the tape after each pass. [Note: you don't want a distribution where you at some point get x > 1 runs on each of the two input tapes; this would result in x runs on the output tape and two empty input tapes, which would require a process of splitting the output tape and copying some of the runs to another tape.]

It's easy to see how these pairs of values are related, but let's look at them in another way that will generalize to an M value > 2. In the following table, we start from the desired end result, that all data is in one long sorted run on a single tape. The tapes below are called Tape A and Tape B, which will represent the two non-empty tapes for each pass. As we saw, these labels will move around among the actual Tapes 1, 2, and 3.

Tape A	Tape B
1	0
1	1
2	1
3	2
5	3
8	5
13	8
21	13

Call the values in a given row of the Tape A column a_i and in the same row of the Tape B column b_i. Then the next row is constructed from the formulas

$$a_{i+1} = a_i + b_i \qquad b_{i+1} = a_i$$

Here you can see how the A column numbers are constructed as the sum of the two previous values, which is how the Fibonacci numbers are being created.

To do the general M-way merge, we need a generalization of the Fibonacci sequence. The **pth order Fibonacci numbers $\mathbf{F^{(p)}_n}$** are defined recursively by the following rules:

$F^{(p)}_n = F^{(p)}_{n-1} + ... + F^{(p)}_{n-2} + ... + F^{(p)}_{n-p}$, for n>=p
$F^{(p)}_n = 0$, for 0 <= n <= p-2;
$F^{(p)}_{p-1} = 1$

In other words, we start with p-1 zeros, then one 1, and then each number is the sum of the preceding p values. ("Regular" Fibonacci numbers are $F^{(2)}_n$.)

Now consider a 3-way merge. In order to get a perfect Fibonacci distribution, we need to use a 3rd-order Fibonacci sequence. The merge process will require 4 tapes, one of which is empty at the start of each pass, so we will only look at the three tapes A, B, and C that are non-empty. As in the 2-way merge example above, Tapes A, B, and C move around on the physical tapes 1-4. What are the run distributions? The first row of the table below starts with the desired end, which is that one tape contains 1 long (sorted) run and the other two tapes are empty. The formulas used to build the successive rows, which produces 3rd-order Fibonacci numbers, are

$$a_{i+1} = a_i + b_i \qquad b_{i+1} = a_i + c_i \qquad c_{i+1} = a_i$$

Tape A	Tape B	Tape C
1	0	0
1	1	1
2	2	1
4	3	2
7	6	4

We'll stop here. The A column is the 3rd order Fibonacci sequence 0, 0, 1, 1, 2, 4, 7, … where each value from the 4th value on is the sum of the previous 3 values. Algebraically,

$$\begin{aligned} a_{i+1} &= a_i + b_i \\ &= a_i + a_{i-1} + c_{i-1} \\ &= a_i + a_{i-1} + a_{i-2} \end{aligned}$$

So our initial setup is 17 runs distributed as follows:

Tape 1:	7 runs
Tape 2:	6 runs
Tape 3:	4 runs
Tape 4:	0 runs

The first merge processes 4 runs in sets of three, one from each tape, combining them and writing the resulting merged (longer) runs to the output tape (Tape 4). The results of this merge are:

Tape 1:	3 runs
Tape 2:	2 runs
Tape 3:	0 runs
Tape 4:	4 runs

Proceeding, after pass 2,

Tape 1:	1 run
Tape 2:	0 runs
Tape 3:	2 runs
Tape 4:	2 runs

After pass 3:

Tape 1:	0 runs
Tape 2:	1 run
Tape 3:	1 run
Tape 4:	1 run

Finally:

Tape 1:	1 run
Tape 2:	0 runs
Tape 3:	0 runs
Tape 4:	0 runs

As in the previous example, the run distributions agree with the successive rows of the table.

The general formulas for computing the values in a table are:

$$a_{i+1} = a_i + b_i \qquad b_{i+1} = a_i + c_i \qquad c_{i+1} = a_i + d_i \qquad d_{i+1} = a_i + e_i \qquad \ldots \qquad t_{i+1} = a_i$$

It may be necessary to "pad" the data collection with dummy records or to carefully split runs to obtain the exact distributions.

Does every user of an external sort process have to implement one of these (or some other) algorithms? No. There are commercially available sorting packages.

Sort Packages

Sorting is a very frequently used process. Implementation of general, efficient sorting algorithms is a difficult task. Two approaches to "packaged" sort algorithms are considered here, the UNIX sort command, and the Windows XP/Vista sort command.

UNIX Sort Command

The UNIX sort command is a general purpose sort, which can sort files numerically or alphabetically. As with any sort process it is necessary to tell the sort process what file is to be sorted and what field(s) compose the sort key.

In UNIX a field is delimited by a <blank> or a <tab> character by default, or by some specified character. Records end with <LF> (other operating systems use <CRLF>). The syntax for the sort command is:

```
sort  [-d, -f, -n, -r] [-tc, +n.m, -n.m]  <file_name>
```

where the options are:

-d Dictionary order, using only letters, digits, and blanks to determine order.

-f Ignore the distinction between capital and lowercase letters.

-n Sort numbers by arithmetic value instead of by first digit.

-r Sort in reverse order.

-tc Sets the field separator to the character c.

+n.m Skips n fields and then m characters before beginning the comparison.
If m=0, the .m portion may be omitted.

-n.m Stops comparison after skipping n fields from the beginning plus m characters.
If m=0, the .m portion may be omitted.

As an example consider a file named *maillist* containing records of the form:

```
Smith:Jane:315 N. 1st St:Yourtown:IN:46260
Dough:John:5585 Main St:Mytown:IN:47032
Black:Anne:8805 Koa Dr:AnyTown:HI:96720
Dough:Mike:5585 Main St:Yourtown:IN:46205
```

The command

```
sort -t: +0 -1 maillist
```

sorts the file on the first field, +0 -1, that is, by last name. The command

```
sort -t: +5 maillist
```

sorts the file starting with field 5 and ending with the end of the record, that is on zip code.

The command

```
sort -t: +0 -1 +5 maillist
```

sorts so that address lines containing the same last name are further sorted by zip code.

Windows XP/Vista Sort Command

Windows XP/Vista has a sort program that can be run from the command line. Consider the text file *NamesUnsorted.txt* that contains

```
Tom     34
Fred    28
Jane    23
Susan   54
Abe     29
Betty   31
```

The command

```
      sort NamesUnsorted.txt /o NamesSorted.txt
```

produces a file *NamesSorted.txt* with content

```
Abe     29
Betty   31
Fred    28
Jane    23
Susan   54
Tom     34
```

The general syntax is:

sort [**/r**] [**/+***n*] [**/m** *kilobytes*] [**/l** *locale*] [**/rec** *characters*] [[*drive1***:**][*path1*]*filename1*] [**/t** [*drive2***:**][*path2*]] [**/o** [*drive3***:**][*path3*]*filename3*]

where

/r : Reverses the sort order (that is, sorts from Z to A, and then from 9 to 0).

/+*n* : Specifies the character position number, *n*, at which **sort** begins each comparison.

/m *kilobytes* : Specifies the amount of main memory to use for the sort, in kilobytes (KB).

/l *locale* : Overrides the sort order of characters defined by the system default locale (that is, the language and Country/Region selected during installation).

/rec *characters* : Specifies the maximum number of characters in a record, or a line of the input file (the default is 4,096, and the maximum is 65,535).

[*drive1:*][*path1*]*filename1* : Specifies the file to be sorted. If no file name is specified, the standard input is sorted. Specifying the input file is faster than redirecting the same file as standard input.

/t [*drive2:*][*path2*] : Specifies the path of the directory to hold the **sort** command's working storage, in case the data does not fit in main memory. The default is to use the system temporary directory.

/o [*drive3:*][*path3*]*filename3* : Specifies the file where the sorted input is to be stored. If not specified, the data is written to the standard output. Specifying the output file is faster than redirecting standard output to the same file.

On the same data file, the command

```
      sort /+9 NamesUnsorted.txt /o NamesSorted.txt
```

sorts on column 9 and produces

```
Jane    23
Fred    28
Abe     29
Betty   31
Tom     34
Susan   54
```

File Structures, 3

Relative Organization

Direct access devices overcome the disadvantages of sequential access by allowing items/records to be accessed independently of one another. The usual device for storing such files is a disk drive. The problem for the user then becomes how to carry out a mapping from information known to the user about some data item(s) into the platter, track and sector disk address of the item(s).

The mapping may be characterized by

$f(\text{value}) \rightarrow \text{address}$

where "value" is some value known by the programmer (e.g., the value of some field in the record), f is the function that takes in the value and produces an address (location for the record). In general the process f can be quite complex, however, the low-level file I/O functions (e.g., the open operation) supply the starting point for the file (platter, track and sector). So the mapping problem is reduced to addressing/accessing items within the body of a file, that is, specifying a location **relative** to the start of the file.

$R(\text{value}) \rightarrow \text{relative address}$

For this reason files of this type are called **relative organization files** or, using the IBM terminology, Relative Record Data Sets (RRDS).

Aside: Using this functional notation the "function" for sequential organization is

$f(\,) \rightarrow \text{next item}$

where item is either a byte (for a stream file) or a record (for a record file). The function needs no argument since in sequential organization the only operation available is to move to the next item.

Unfortunately relative organization files provide no more metadata support than sequential organization files and more information is needed, for example, record length. The only information that can be obtained is the starting location of the file (from the open function). The user must provide any other information, such as record length and number of records in the file.

C++ random file access

C++ provides a very flexible, low-level way of directly accessing items (a carryover from C, which is designed for low-level operations). C++ treats all disk files as a linear array of bytes on the disk. The open function establishes the origin or starting address for the file. Each *fstream* object maintains a file position marker, a long integer that represents an offset from the beginning of the file and that "points to" the next byte to be read from or written to the file. C++ provides functions to access and change the file position marker. All the user has to do is specify an offset value to one of the two *seek* functions (discussed below) and that will move the file position marker to the specified location so reading/writing of items can begin. No concept of a record is employed here.

Files in C++ are usually opened for input or for output by specifying an input-stream object (an instance of class *ifstream*) or an output-stream object (an instance of class *ofstream*). If you want to modify a byte in a file, you don't want to have to open the input file, read the data, close the input file, change the data, open the output file, write the changed data, and close the output file. Using an object of type *fstream* you have both read and write access to the data file at the same time, which is a speedup factor. You have to indicate as part of the open statement how you want to use the file:

```
fstream iostream;
iostream.open(filename.c_str(), ios::in | ios:: out); //both for input and output
```

In the following table, the function *seekg* positions the "get" pointer to read a value from the file. The function *seekp* positions the "put" pointer to write a value to the file. This is how the file gets switched back and forth between its input and output forms. The get pointer and the put pointer are synchronized, so that if the get pointer is advanced, the put pointer is automatically advanced to the same position in the file. The *tellg* and *tellp* functions return the current value of the get pointer or the put pointer. (If you were using a stream just for input, you could only use the seekg and tellg functions; if you are using a stream just for output, you could only use *seekp* and *tellp*). The *tell* functions therefore access the position marker and the *seek* functions change it.

seekg(offset, mode)	**mode** can be ios::beg //beginning of file ios:: cur //current get pointer ios:: end //end of file If the mode parameter is absent, the default is the beginning of the file. **offset** is the number of bytes from the mode position, and can be either 0, positive (move forward in the file) or negative (move backward in the file)
seekp(offset, mode)	Parameters as for seekg
tellg()	Returns the current value of the get pointer in the file
tellp()	Returns the current value of the put pointer in the file

Again, the structure of the file is "in the eyes of the beholder", that is, the programmer. So you must be able to figure out the offset (number of bytes) to reach the byte you want to read.

If you have a record-oriented file with fixed-length records, you don't have to "manually" count bytes. Consider the following data file called *Datefile.txt*, which can serve to demonstrate the ideas even though it is an ordinary text file.

```
07 16 1987
02 22 1996
11 01 2003
12 04 1962
04 05 1995
06 24 2005
```

and a C++ program to modify the 5th record in the file:

```
#include "utility.h"
#include "fstream"
#include <iomanip>

class Date
{
public:
	int getDay();
	int getMonth();
	int getYear();
	void setDay(int newDay);
	void setMonth(int newMonth);
	void setYear(int newYear);
	friend fstream& operator >> (fstream& outs, Date &x);
	friend fstream& operator << (fstream& ins, const Date &x);
private:
	int month;
	int day;
	int year;
};

int main()
{

	Date Birthdate;
	string filename = "Datefile.txt";
	fstream iostream;
	iostream.open(filename.c_str(), ios::in | ios:: out);  //for both input and
	                                                       //output
	if(iostream.fail())
	{
		cout << "File not found" << endl; ;
		exit(1);
	}

	iostream.seekg(4*sizeof(Date));          //to read record 5
	//iostream.seekg(0);                     //to read first record

	iostream >> Birthdate;                   //read the record

	cout << setfill('0');
	cout << setw(2) << Birthdate.getMonth() << " "
	        << setw(2) << Birthdate.getDay() << " "
	        << setw(4) << Birthdate.getYear() << endl;

	Birthdate.setDay(15);                    //modify the record

	iostream.seekp(4*sizeof(Date));          //reset the write pointer
	iostream << Birthdate;                   //write the new record

	iostream.seekg(-long(sizeof(Date)), ios::cur);  //reset the read pointer
	iostream >> Birthdate;                   //read the new record

	cout << setfill('0');
	cout << setw(2) << Birthdate.getMonth() << " "
	        << setw(2) << Birthdate.getDay() << " "
	        << setw(4) << Birthdate.getYear() << endl;

	iostream.close();
	return 0;
}

int Date::getDay()
{
	return day;
}

int Date::getMonth()
{
	return month;
}

int Date::getYear()
{
	return year;
}

void Date::setDay(int newDay)
{
	day = newDay;
}
```

```
void Date::setMonth(int newMonth)
{
	month = newMonth;
}
void Date::setYear(int newYear)
{
	year = newYear;
}

fstream& operator >> (fstream& ins, Date &x)
{
	ins >> x.month >> x.day >> x.year;
	return ins;
}

fstream& operator << (fstream& outs, const Date &x)
{
	outs << setfill('0');
	outs << setw(2) << x.month << " "
	       << setw(2) << x.day << " " << setw(4) << x.year << endl;
	return outs;
}
```

The resulting file is

```
07 16 1987
02 22 1996
11 01 2003
12 04 1962
04 15 1995
06 24 2005
```

The file is a record-oriented file where each record is an instance of the Date class. In this program, the **sizeof** function returns the number of bytes of the Date object Birthdate, so the statement

```
	iostream.seekg(4*sizeof(Date));
```

walks the get pointer over 4 record-lengths from the beginning of the file to position itself at the beginning of the 5th record. The record is read from the file using the overloaded extraction operator that was defined for the class. The record is modified, then

```
	iostream.seekp(4*sizeof(Date));
```

in a similar fashion walks the put pointer 4 record-lengths from the beginning of the file. The new record is written to the file using the overloaded insertion operator that was defined for the class. Then, because we really want to see what was actually written to the file, we "backup" one record length using

```
	iostream.seekg(-long(sizeof(Date)), ios::cur);
```

in order to get back to start of the record we just wrote so we can read it and write it to the screen. (The *sizeof* function returns an unsigned integer, so I've cast it to type long before making it a negative number.) [Aside: it's a pain to debug a program that changes the contents of a file because you have to have an extra copy of the original file and replace the output file with the original copy after each run.]

Fixed-length record organization

The more usual model used for a relative organization file is a singly dimensioned "array" on the disk. The array consists of a set of fixed length records (the elements of the array). Access to the elements/records is through a subscript. The subscript is called the Relative Record Number (RRN). The RRN represents the ordinal position of the record in the file. There are no special delimiters (such as a CRLF) between records. It is tempting to think of the model as being a 2-dimensional array on the disk, one dimension being the records and the other being the fields. Recall the organization does not provide any metadata support for fields. Fields, if they exist, are known only to the user through the source code.

Note the similarities with ordinary (internal) 1-D arrays. The elements of an array must all be the same data type. When an internal array is created, the system retains information about the starting address in memory of the array and the number of bytes used by a single array element. It does NOT keep track of the size of the array, which is why it is up to the programmer to avoid going past the end of the array. When you ask for MyArray[2], you are asking for the 3rd array element because array indexing starts with 0. The system finds the memory address of the 3rd array element by computing an offset from the starting array address:

starting address + 2(bytes/array element) = address for element 3

In a relative organization file, the records must all have the same structure. The system knows the disk address of the start of the file and finds the disk address of the 3rd record by computing an offset from the start address:

starting address + 2(bytes/record) = address for record 3

which is exactly what we did in the C++ program above.

Programming languages that support the one-dimensional array type of model (file organization) through special language syntax hide this offset computation. COBOL and FORTRAN are about the only examples. In fact, while relative organization files are an important conceptual steppingstone, a better file organization technique was developed later.

Metadata (file-level CRUD rules)

Again, the important ideas of metadata are what and where things are stored. As mentioned earlier, only Create (through the open statement) and Delete are supported at the file level.

But, in a relative organization file it is necessary to know the length of the records. It is also necessary to know how many records there are in the file. The concept of EOF makes no sense here since it is possible to jump over an EOF marker if one were present, that is, it is possible to address out of the range of the file by specifying a RRN that is too large (just like it is possible to exceed the bounds of an internal array).

Some of these metadata values often are stored in the source program text, for example from FORTRAN,

```
      OPEN(UNIT=1, FILE='EMPLOY', ACCESS='DIRECT', FORM ='UNFORMATTED',
     *              RECL = 22, STATUS = 'NEW')
```

where the RECL=22 is the length of the record in bytes. (Compare this with the OPEN statement in the FORTRAN example in an earlier section where RECL does not appear and ACCESS is sequential.) Neither COBOL nor FORTRAN provides a syntactic mechanism to remember the number of records in the file. What can be done?

One very useful technique is for the **user / programmer** to define the first record in the relative file to be a "control record". This record is defined to contain two pieces of information, the length of each record and the number of records (usually not counting the control record) that are currently in the file. Doesn't this violate the "array" model since now all the elements in the array do not look alike? Yes, but ... First the control record must be the same length as all the other records in the file. The problem is that the length is unknown. However, since the control record is the first record in the file, that is, it begins at the origin of the file, it is possible to open the file temporarily with a record length of say 8 bytes (two 4-byte integers) for FORTRAN (or 12 characters – two 6-digit numbers - for COBOL). This would be just enough length to read two integer values, one for the length of the records and the other for the number of records. Then the file is closed and reopened with the "right" record length (of course this technique assumes that the data records are longer than the control record so that the control record contains the two integers plus junk to get the "right" length record).

A FORTRAN sample for this might be

```
      OPEN(UNIT=1, FILE='EMPLOY', ACCESS='DIRECT', FORM ='UNFORMATTED',
     *             RECL = 8, STATUS = 'OLD')
      READ(UNIT=1,REC=1) LENGTH, NUMREC
      CLOSE(UNIT=1)
      OPEN(UNIT=1, FILE='EMPLOY', ACCESS='DIRECT', FORM ='UNFORMATTED',
     *             RECL = LENGTH, STATUS = 'OLD')
        ...

C       Test the  REC=  value in each READ to be sure it does not exceed NUMREC.
C       The  +1  is to account for the control record.
      N =  ...
      IF(N .GT. NUMREC) GO TO 1000
      READ(UNIT=1,REC=N+1) ...
      ...

      CLOSE(UNIT=1)
C          Error branch
1000  WRITE(*, 'RRN out range.')

      END
```

Note that the first record in the file is REC = 1, unlike the C++ array index that begins at 0. In the above code, the lowest value of N that can be used is 1, which will cause a read of REC = 2, the first "legitimate" record after the control record.

The idea of a "control record" is a user supplied artifice to introduce metadata into the file. It is NOT part of the file organization.

Using this technique the metadata about the file is in the file, not in the program. This is an initial step toward data independence of the program. The alternative is that the information must be in the user's head so it can be incorporated into any source code that accesses the file.

Next consider the record-level CRUD rules.

Record-level CRUD Rules (Relative Organization)

The CRUD rules for relative organization files take advantage of the random access nature of the storage device. Create is still limited to an "append" operation (as in the sequential case). Read and Update are straightforward once the RRN is specified. Delete still poses a problem.

The examples used to demonstrate the CRUD rules employ binary format files. The same ideas work for character format files. Binary format is used most of the time for relative files for two reasons: human readability is usually not required and a timesaving occurs since the character to binary conversions needed to go between the file and memory is avoided.

Create

In a relative organization the Create rule is implemented as append. This means that the process of adding a new record extends the file and the new record is added at the end of the file. A program must be written to append the record to the file (there are no "text" editors for relative files).

To add a new record to the end of the file it is necessary to know how many records were in the file at the time of the addition. If the control record idea is used, the number of records in the file may be obtained from the control record. Then, of course, the control record must be updated to reflect the new count. Such a process might involve (using FORTRAN)

```
      READ(UNIT=1,REC=1) LENGTH, NUMREC
      NUMREC = NUMREC + 1
      WRITE(UNIT=1,REC=NUMREC+1) ...
      WRITE(UNIT=1,REC=1) LENGTH, NUMREC
```

The READ and WRITE to REC=1 is the update process for the control record. Note that this is a change to the stored metadata for the file. This indicates that record-level CRUD rules can have an effect on the metadata.

Read

The Read rule is very straightforward. Once the RRN is known, the Read operation accesses the record directly. In FORTRAN, the code

```
      N = 11
      READ(UNIT=1, REC=N + 1)
```

will read the 11th user record (record 1 is the control record) in the file.

The question is, how is the value of the RRN determined? This is discussed in the section on algorithms that follow. However, it is very important to recognize that a relative organization file can be read sequentially. This can be done with FORTRAN instructions like:

```
      DO 100 I=1, NUMREC
      READ(UNIT=1,REC=I+1) ...
100   CONTINUE
```

This type of construction helps to answer the question of how to find the number of records in the file if the control record idea is not used. Without a control record, some kind of sentinel value must be stored in the last physical record in the file, perhaps an EMP_NUM of -999. Then a code fragment like:

```
      N = 1
100   READ(UNIT=1, REC=N) EMP_NUM
      IF(EMP_NUM .EQ. -999) GO TO 200
      N = N + 1
      GO TO 100
 200  NUMREC = N - 1
```

could be used to set the value of NUMREC. Notice that the "special case" record now becomes the last record in the file (the sentinel record) rather than the first record in the file (the control record). The other CRUD rules have to be "taught" either to handle the sentinel record properly or to handle the control record properly. As mentioned, the concept of EOF does not apply for relative files.

Update

The Update rule is straightforward too. It is a combination of READ and WRITE operations at the programming language level. For example to Update user record 17 using FORTRAN

```
      N = 17
      READ(UNIT=1,REC=N+1) LIST1, LIST2, ...
C           Change the value of the LIST items to be updated
      LIST1 = ...
      WRITE(UNIT=1,REC=N+1) LIST1, LIST2, ...
```

The WRITE operation performs the Update (write over the existing record). Of course once the old values are overwritten there is no way to undo the operation unless the user has made provisions to save a copy of the old record someplace else, that is to say, the WRITE operation is destructive and makes no automatic backup. No change is needed in the control record since the number of records in the file has not changed.

Delete

The Delete rule poses a problem. The question is what to do with the space that was being used by the record that is to be deleted. There are two choices, physically remove the record from the file, or logically remove it from the file.

Physical removal amounts to "rolling up" the records left in the file one "slot". Suppose record number 10, out of 15, in a file is to be deleted (removed). Record 11 is copied into slot 10, then 12 into 11, and 13 into 12, and so on. Then, of course, the control record must be read, NUMREC must be decremented by one, and the control record must be written back.

The idea behind logically deleting a record is to assign a field in the record (usually the first byte in the record) to be an "activity flag". If the flag in a particular record is set one way, say off, it indicates that the record is part of the file. To logically delete the record from the file the flag is set the other way, say on. Now, all the other CRUD rules, especially Read, must be taught to look at this flag. If the flag is on, the record is logically deleted, and it must be ignored.

Logical deletion does not free up the space, it only marks the record as deleted. It is usually the case that there is some kind of garbage collection or packing process that can be executed periodically to physically remove the logically deleted records from the file.

Any programming language that supports relative organization files can employ the physical deletion method. Some COBOL compilers support logical deletion using an activity byte set to HIGH-VALUES to indicate a deleted status.

Typical Applications

If the analogy between relative organization files and arrays is carried to the limit, anything that can be done with arrays can be done with relative files. The concern is time! Arrays in RAM operate at electronic speed (nanoseconds), while relative files operate at mechanical speed (milliseconds).

A frequent application of relative files is an update process similar to the situation described in the section on the balance line algorithm but running in an on-line environment rather than a batch environment. The basis of this process is the function R(value). Suppose the value is an employee number; then R(emp_num) must return the RRN for the record for that employee. Given the RRN, the data-level CRUD rules (described above) can be used to make the necessary adjustments to the data file.

Two concerns are: What is the exact form of the function R, and what if something goes wrong? Several different R functions are discussed in the next section on algorithms. The concepts of backup and recovery of file systems / databases belongs in a database course and will not be covered here.

Algorithms

Sorting

The importance of being able to get records in a file into some prescribed order and maintain them in that order is independent of the file organization. Different organizations provide different advantages (and disadvantages) in the ways sorting can be implemented. The direct access nature of relative organization (i.e., the array model) allows approaches to external sorting not available with sequential organizations. Basically, the simple internal array-based sorting algorithms can all be used.

Insertion Sort

The basic idea of an insertion sort is to take the next record from the unsorted file and place it in the proper position (the correct position based on its key value and the key values in the already sorted file). One way to implement this type of process using relative files is to start at the end of the sorted relative file. Create a new empty record at the end of the file (append it to the file) making a new, empty last record (Nth record). Compare the incoming key value with the key value of the second to the last record in the file (N-1st record). If the incoming key is greater, place the record in the last record (N). If the incoming key is less than the key of the N-1st record, move the N-1st record down to the Nth slot. Repeat this process, comparing the incoming, unsorted key value with the third to the last record in the file. Continue this process until the "correct" position is found. Update the control record if one is being used.

The individual worst case occurs when the unsorted key is smaller than any key in the sorted file. Then all of the records in the sorted file have to be moved, which is an $\Theta(N)$ operation. The overall worst case is when the set of "unsorted" records is actually sorted in the opposite order desired in the sorted file. Then every insertion requires that every record be moved. This leads to the conclusion that the insertion algorithm is $\Theta(N^2)$ in the worst case.

Selection sort and bubble sort algorithms could be implemented in a similar way. They would also be $\Theta(N^2)$. Please note that even though the order is the same as a similar internal sort, the time will differ significantly since the internal sort counts CPU operations (speed range 10^{-9} sec) and the external sort counts disk operations (speed 10^{-3} sec). This is a factor of a million difference in time!

More complex internal sorting algorithms such as quicksort are just too complicated to carry out at the file level. Instead, use the polyphase merge sort discussed earlier.

Searching

Searching means looking for a value in a file, more particularly, looking for a record with a target key value. We have discussed several internal array-based search algorithms, and these also can be carried out on a relative organization file structure:

- Sequential search (accessing the records sequentially)
- Short sequential search (if the file is sorted by key value)
- Binary search (if the file is sorted by key value). Recall that binary search requires knowing the high index value in the section of the array being searched, initially the entire array. This is why we need to know the number of records in the file, which will provide the maximum RRN.

You will recall that each of these internal search algorithms returned the position in the array of the target value. When applied to a sorted relative organization file, this value is the RRN of the desired record. The order of the binary search is $\Theta(\log_2 N)$. This is much better than a linear search, especially when the number of records, N, is large. But again recall that the operations being counted are disk operations.

The binary search could be used as the Read rule function in the EMPLOY example as long as the file is a relative file and maintained in sorted order by `EMP_NUM`. It could be used as the "location" part of the update process too. But what about Create (adding a new record) and Delete? Again, getting-something-for-nothing does not succeed. The file must be maintained in sorted order. So, added records must be inserted in their proper place and the deleted records should be physically removed from the file (the algorithms given above do not expect to access a "logically deleted" record).

Hashing

We also discussed hashing as an internal search technique, and it too can be applied to relative organization files. Hashing is another way of implementing the functional relationship between a key value and an address, i.e.,

$$H(\text{Key}) \rightarrow \text{RRN}$$

where H is the hash function. In the ideal case – no collisions – only one disk access would be required to retrieve the employee record given the EMP_NUM, rather than having to search for the correct record. But, in this imperfect world, a collision resolution scheme is needed. For simplicity, we'll assume a linear probing scheme.

The hash table is actually itself a relative organization file B built initially from relative organization file A by sequentially reading records from A and storing them in B. File B is a higher-order organization than a relative file organization in that the scheme is implemented "on top of" a relative organization. Do CRUD rules apply? Of course they do. What new or different properties do these rules have?

The Create rule was discussed earlier. This rule must find the correct slot for the record based on the hashed value of its key, and adjust the structure accordingly.

The Read rule was the driving force for the design of this process. The hashing scheme is designed to reduce the number of disk accesses needed to find an item (the trade-off here is the introduction of the hashing function and the more complex file structure built on top of the relative structure). Note that the Read rule has two parts. First the target key is hashed using the hash function to obtain the hashed address (RRN ultimately). But the comparison of keys (target key value to file key value) is not made on hashed key values but on unhashed key values. So finally the Read rule must look at the actual key values in any records involved in the linear probing process.

To get the efficiency of the Read rule something was sacrificed. There is no way to access the hashed structure in a "sequential" manner. The Read rule is driven by hashed values and there is no way to know what value produces what address!

Update must, in general, be implemented by a Delete followed by a Create. If the key value changes its hashed value, hence its hash address changes, the record will "move" in the structure. Non-key updates could be done in place.

The Delete rule begins with hashing a key and then uses the same type of probing followed by unhashed key comparisons used in Read. Once the desired record is found (and possibly copied for use by Update) whatever mechanism is used to denote an "empty" slot is invoked to logically remove the record from the structure.

Closing Thoughts

In the discussion of data structures, the array was found to be a very useful primitive structure, useful in modeling other, more complex structures (e.g., stack, queue, ...). As mentioned earlier, relative file organization models the concepts of an array on the disk (a direct access device). The analogy between arrays within data structures and relative organization files within file structures is not coincidental. Relative file organization is a primitive organization upon which other file organizations are based.

File Structures, 1

For the last part of the course we will take an overview look at file structures. This will include:

- Different file organizations (sequential, relative, and indexed)
- The pros and cons of each organization and the difference between metadata and user-data CRUD (Create, Read, Update, Delete) rules for the different organizations
- The important features of external sort and merge algorithms

Introduction

Why do we need files? For several reasons:

1. Persistent storage. Computer memory is volatile and anything stored there is lost when you turn off the machine. Obviously no one wants to recreate documents, computer programs, data files, spreadsheets, etc., on the fly every time they are needed.
2. Limitations of computer memory. Despite the great increases in RAM storage, the amount of information that can be brought into memory is still limited, whereas the size of external storage is much greater.

So file storage is clearly a necessity. But accessing data in a file to move it into main memory where it can actually be used is a slow process. We've already talked about this for **disk storage**, where **seek time** (to move the read head over the correct track), **latency** (to move the correct sector under the read head) and **transfer time** (time for the actual data transfer) make this operation many times slower than access to an internal memory location [think of a record player – if you've never seen one, look it up!]. External storage could also use **magnetic tape**; tape is still the least expensive medium for archival storage and it is highly reliable. Time must be spent advancing the tape to the proper location for the desired data item [think of an audio tape player, popular before the days of MP3 players and iPods!].

From *File Structures* by Folk, Zoellick, and Riccardi, Addison-Wesley, 1998: "Assume that memory access is like finding something in the index of this book. Let's say that this local, book-in-hand access takes 20 seconds. Assume that accessing a disk is like sending to a library for the information you cannot find here in this book. Given that our "memory access" takes 20 seconds, how long does the "disk access" to the library take, keeping the ratio the same as that of a real memory access and disk access? The disk access is a quarter of a million times longer than the memory access. This means that getting information back from the library takes 5 million seconds, or almost 58 days." Quite a difference!

So here is the inherent conflict in files – slow access time (bad) vs. nonvolatile, large capacity (good). The central task, then, is to minimize the number of disk accesses to get the information we need from the file.

As we know, all data is stored in computer memory in binary form. Data in a file is also in binary form, but the file can be interpreted as either a text (ASCII) file or a true binary file. Suppose you want to store the integer 29 in a file. In an ASCII file, this will be stored as the character 2 followed by the character 9. This would nominally require 2 bytes (8 bits per character). When the contents are viewed through some sort of text editor (i.e., Notepad), you see 29. The file is "human-readable." The 8-bit binary representation of the integer 29 (1*16 + 1*8 + 1*4 + 0*2 + 1*1) is 00011101, so this could nominally be stored in 1 byte in a binary file, although it would actually be stored in 4 bytes (on a 32-bit word machine). If you try to view the contents through some sort of text editor, you see strange symbols. The file is only "machine readable." We'll assume we are dealing with ASCII files.

In our look at Common Data Structures early in the semester, we imagined an ADT "list" and considered the desired operations on list elements:

• Insert a new list element	(Create)
• Access a list element	(Read)
• Change the value of a list element	(Update)
• Delete an existing list element	(Delete)

We then saw how three implementations of the list ADT (array, vector, linked list) handled these operations with greater or lesser difficulty. These four operations, often called CRUD rules, apply to files as well, in two dimensions. At the data level, we want to know how to create a new file entry (item of data), how to read (access) a file entry (this is what we want to optimize), how to update a file entry and how to delete a file entry.

The term **metadata** means "data about data". If we think about files as data containers, then metadata CRUD rules can apply to files themselves. At the metadata level, how do we create a new file, read information about a file, update some information about a file, or delete a file? The operating system will help your program to create a file (through open and close statements) and to delete a file. The operating system actually keeps metadata – data about files. For example, when the file was created, when it was last modified, the file size on the disk, etc. The most important metadata maintained by the operating system is the address in the physical secondary storage medium (tape or disk) for the beginning of the file. On early Windows operating systems this information was kept in the aptly named File Allocation Table (FAT); later Windows systems use New Technology File System (NTFS), and other operating systems have their own file management systems. The operating system will not, in general, allow you to read (access) or update information about a file using a higher-level programming language; you can't, for example, change the physical address of the file on your hard drive from within a program, nor the date of its creation.

As a simple example, you can write a little text file in Notepad and save it to the disk. You are asked for the file name you want to save it as. Voila - you've created a file! Within C++ you create a text file by

```
ofstream outfile;
outfile.open("MyFile.txt");               //metadata-level Create rule

//write to the file
for (int i = 0; i < 10; i++)
{
      outfile << i << endl;               //data-level Create rule
}
outfile.close();
```

and read it by

```
ifstream infile;
infile.open("MyFile.txt");
int a[10];
//read from the file
for (int i = 0; i < 10; i++)
{
      infile >> a[i];                     //data-level Read rule
}
infile.close();
```

and now you decide it's a pretty useless file, so you delete it

```
if( remove( "MyFile.txt" ) == -1 )        //metadata-level Delete rule
  cout <<"Error deleting file" << endl;
else
  cout <<"File successfully deleted" << endl;
```

One of the nice things about stream classes in C++ is that (once the stream is "hooked to" the file through the open statement), reading data from or writing data to a file looks no different from reading data from a keyboard or writing data to the screen. In some programming languages, communicating with a file is very different than keyboard input or screen output.

So again, we will be looking at different file organization structures and how they handle CRUD rules, mostly at the data level. At the metadata level, regardless of the file organization, we are in general limited to Create and Delete.

Sequential Organization

A sequential organization is characterized by the fact that the items stored in the file are consecutive. The first item is at the beginning of the file, then the next item, and so forth to the end of the file. Usually the only way to access an item in a sequential organization is to “process” all the items before (in front of) that item in the file. The IBM term for such a file is an **Entry Sequenced Data Set** (ESDS). This is a very descriptive term; it indicates that the only arrangement of information on this type of file organization is the way in which the data were entered.

There are two types of sequential files, stream files and record-oriented files. Stream files are the more primitive. The distinction is in whether or not there are delimiters recognizable by the I/O operations that divide the file up into physical records. Stream files have no delimiters, while record-oriented files include an end-of-record (EOR) delimiter (at least conceptually).

Stream Files

A stream file consists simply of a stream of bytes or data items. From the user's point of view there is some kind of a "pointer" maintained by the system (low level operating system I/O routines) that always points to the next location from which data is to be read or to which data is to be written. Of course, when the file was opened this pointer was set to point to the beginning of the file. Each I/O operation has the side effect of automatically moving the pointer to point to the next location. When the file is closed the pointer is no longer needed. Also, if the last I/O operation was a write, it is often the case that some special sentinel value is written to mark the end of the file (i.e., an end-of-file, EOF, mark).

There are no logical field or record delimiters within the stream, just a beginning and an end. The user's code must "parse" or break up the stream into meaningful chunks.

C++ provides operations at the stream level. In C++, a file is modeled as a linear array of bytes on the disk. The open function establishes the connection to the file, and sets the "pointer" to point to the first byte in the file. Functions like *get* (or *put*) read (or write) a single byte (character) starting from the pointer. These functions "advance" the pointer. The functions signal exceptional conditions, for example, when an end-of-file is encountered or a write operation has failed. Consider the following code that reads through a stream file (called STREAM.DAT) of characters looking for the symbol "(".

```
#include <fstream >
#include <iostream>
using namespace std;

void main()
{
  char c;
  ifstream file_handle;
  file_handle.open("STREAM.DAT");
  file_handle.get(c);
  while (!file_handle.eof())
  {
     if (c == '(')
     {
        cout << "found the (" << endl;
        break;
     }
     file_handle.get(c);
  }
}
```

The *get* function assigns the character read to c. The break statement within the loop ends the while loop if the value of c is "(". The *eof* function returns the value true if the end-of-file marker has been read.

Languages like COBOL and FORTRAN have no stream I/O capabilities; they are record oriented languages (see the discussion that follows).

Record-Oriented Files

The main difference between a stream file and a file composed of physical records is that a special delimiter, called an end-of-record indicator (EOR), is defined. The data between two successive EORs (or the beginning of the file and the first EOR) is called a record. Records may be of any length. If all the records in the file are the same length it is called a fixed-length record file, otherwise it is called a variable length record file.

Often the record delimiter is a pair of characters, a carriage return character (ASCII 13) and a line feed character (ASCII 10). This pair is often represented by the symbol CRLF (pronounced "curlif"). Most text editor programs, such as Notepad, place a CRLF in the file each time the <Enter> key is pressed. This is a carry-over from the functionality of a typewriter. On a typewriter when the <Return> key is pressed the mechanism carries out a carriage return (moving the printing point to the left edge of the paper) and a line feed (moving the paper up one line). However, pressing <Enter> on a computer keyboard has whatever effect the program assigns to that key, not a fixed action like CRLF - e.g., in Microsoft Word, <Enter> causes a paragraph mark to be added to the document; in Excel <Enter> usually causes a move to the next cell.

It is still the case that records are just a sequence (or stream) of bytes. There is no "internal structure" (no fields) on this group of bytes. The user's program must provide the instructions to parse the records into fields. The exact details of how this is accomplished depend on the programming environment.

Assuming a text file with CRLF as the only record delimiter, C++ can do a record-at-a-time read by using the *getline* function. But once the record has been read, the program itself must parse the record, picking it apart based on the known (to the programmer) structure of the file.

The following C++ program reads the file *data.txt*, which looks like this:

```
123456jack10.3
896321bill12.8
```

```
#include "utility.h"

int main()
{
       string filename = "data.txt";
       string stuff;
       int intone;         //4-digit integer
       int inttwo;         //2 digit integer
       string title;       //4-character string
       double dblone;      //4-character decimal number

       ifstream in;
       in.open(filename.c_str());
       if (in.fail())
       {
             cout << "Failure to open input file" << endl;
             exit(1);
       }
       while(!in.eof())
       {
             getline(in,stuff);
             cout << stuff << endl;
             intone = atoi( stuff.substr(0, 4).c_str());
             cout << intone << endl;
             inttwo = atoi(stuff.substr(4,2).c_str());
             cout << inttwo << endl;
             title = stuff.substr(6,4);
             cout << title << endl;
             dblone = atof(stuff.substr(10,4).c_str());
             cout << dblone << endl;
       }
       in.close();
       in.clear();

       return 0;
}
```

In the above program, the specifications (metadata) for the layout of the fields in the record is held in the source code.

In languages like COBOL and FORTRAN where the I/O operations are record oriented, the file must contain EORs and each I/O operation processes one full record. There is a syntax that allows the user to specify the parsing instructions (metadata for the field layout).

Consider the following FORTRAN code that reads up to 15 records from a file called DATA.DAT that contains records consisting of a 5 digit integer and a 20 character string and a 3 digit integer. The program displays the first number and the string on the screen (the last number is not read or processed)

```
      INTEGER N
      CHARACTER * 20 S
      OPEN(UNIT=1,FILE='DATA.DAT')
      DO 100 I=1,15
      READ(UNIT=1,FMT=5000,END=200) N, S
 5000 FORMAT(I5,A20)
      WRITE(*,FMT=5001) N, S
 5001 FORMAT(1X, I5, 2X, A20)
  100 CONTINUE
  200 CLOSE(UNIT=1)
       END
```

The points of interest here are:

- The unspecified options in the OPEN statement default to a sequential character file.
  The OPEN statement without taking the defaults would have been:
  OPEN(UNIT=1, FILE= 'DATA.DAT', ACCESS= 'SEQUENTIAL', FORM= 'FORMATTED', STATUS = 'OLD')
- The READ statement references unit 1 (the equivalent of the input stream in the C++ program) and uses FORMAT statement number 5000.
- FORMAT statement 5000 contains the parsing "instructions" to convert the character (byte) information in the file into the binary form needed by the memory variables N and S mentioned in the I/O list
- The WRITE statement references the screen, unit *, and FORMAT 5001.
- FORMAT 5001 adds a space at the left of the number (1X) and two spaces between the output items (2X).
- Each READ reads one physical record, each WRITE writes one physical record.
- The READ statement starts processing information in the file at the current location of the "pointer", parsing according to the FORMAT. Parsing stops at the end of the FORMAT statement description, in this case at the end of the character string. The "pointer" is advanced beyond the CRLF (skipping the 3 digit number) since the whole record was read even if only part of it was used (parsed).
- Rewording the previous statement: FORTRAN has no stream capability whatsoever. Only complete records are processed even if only part of the information in the records is used (i.e., the first two fields, ignoring the last field). FORTRAN was designed to read a punched card, and the whole card had to be run through the card reader.

A text file may contain other record delimiters. For example, if the data.txt file for the above C++ program has the following structure

```
2561    85          Help    15.4
```

then it could be read by

```
in >> intone >> inttwo >> title >> dblone;
```

The extraction operator skips whitespace until it finds a character that represents the beginning of an integer, and it stops reading characters for that integer at a blank space or a non-digit character because these serve as a "record delimiter" for integer data. Similarly the extraction operator for string data skips whitespace looking for any nonblank character (which represents the beginning of a string) and stops reading characters for that string at a blank space. Here the programmer has an easier task to "parse" one line of the file. But note that if you wanted a program to read first names, you could not use a single extraction operator; although this would work fine for "Fred", for a name such as "Mary Lou", you would only get Mary. Here's where you have to use getline and be sure you know how long the "name field" is, which must be the same for every record.

In summary, even for a particular file organization, e.g., a sequential organization, different programming languages implement the details of the file access operations differently.

Record-level CRUD Rules (Sequential Organization)

Since a sequentially organized file is usually stored on a sequential device, e.g., magnetic tape, the data level CRUD rules are limited to "append" and Read the next item in the sequence. Update and Delete require rewriting (copying) the file with the appropriate changes and/or deletions.

Create

In a sequential organization new items (additions to a stream or additional records) are created at the end of the file. The file is extended as items are appended to the file.

Since the arrangement of the file is entry order, the newest item is the last item in the file. If some other arrangement is desired, for example, sorted order on some part of the record, the Create operation will not be sufficient since Create can only append items to the file. The file will have to be copied over while inserting the new record in the proper position.

Read

The Read operation in a sequential file is interpreted as "read next", that is, read the next record in the sequence. The "read next" operation implies a "forward" operation. Some languages provide the opportunity to "backspace", that is, move toward the beginning of the sequence and "rewind" to get back to the start of the file.

Update

Update in a sequential organization is a problem. It is not reliable to overwrite a portion of a file on a truly sequential medium (e.g., magnetic tape). The storage device may decide that the portion of the medium onto which the information is to be written is no longer of sufficient quality to be used. The device may then mark that part of the medium as bad and move ahead and write the information on the next available "good" portion of the medium, possibly destroying some other data. The appropriate way to update information in a sequential organization is to copy the original file over into a new file, making whatever updates are needed.

If the file is stored on a disk drive it is physically possible to update an item by overwriting the item. From a design view this may not be a good idea since no backup is available, that is, it is not possible to undo the operations if the system or program fails part of the way through a file update process.

Delete

The only way to delete an item from a sequential file is to copy the file, leaving out the items that are to be deleted.

Text Files

An important application of sequential files is a text file. A text file is an automated version of the typewritten document. Variable length records delimited by CRLFs are presented to the user in entry sequence order.

Programs called text editors have been developed to manage the record level CRUD rules for this type of file. The editors allow the user to Create (append) new records to the text file just as if the document were being typed. Of course the Read operation simply reads the records in order and displays them on the screen (for the user to read).

With a typewritten document, the Update and Delete process required complete retyping of the document. The text editor has automated this process. The automation is often based on reading the text file and storing the text in the memory of the computer, that is, changing the organization from a sequential organization to a random access organization. This is the way Notepad attacks the problem.

What if the file is larger than can be stored in RAM? Instead of copying the file into RAM, programs that deal with large files often make a "working copy" of the file with an organization designed to support the Update and Delete process. When the editing session is complete the working file is written out as an entry sequenced data set, i.e., a sequential file.

Most programming language development environments require the programmer to use a text editor to enter the source code statements. An editor may also be used to construct input data files for a program to use. The program may produce printable output. The compiler and/or linker produce report files. These are all sequential text files where only Create (append) and Read operations are employed.

File Structures, 2

Sequential Organization

The topic is still sequential file organization, typically text files. Two important algorithms for processing sequential files will be discussed.

Master File Update (Balance Line Algorithm)

Data processing is a term used to denote manipulation of large amounts of data where the emphasis is primarily Input/Output oriented with a minimum of computation (as opposed to computing missile trajectories to the moon which has many computations and little input). There are multitudes of computer applications that fall into the category of data processing, e.g.,

- checking and saving account records in a bank
- billing operations of the power and light or gas companies
- premium payment records of an insurance company
- tax records of the IRS

All these applications have something in common; they depend on a "data-store", that is, a quantity of information about company activities, for example, a list of customers and what services they have used.

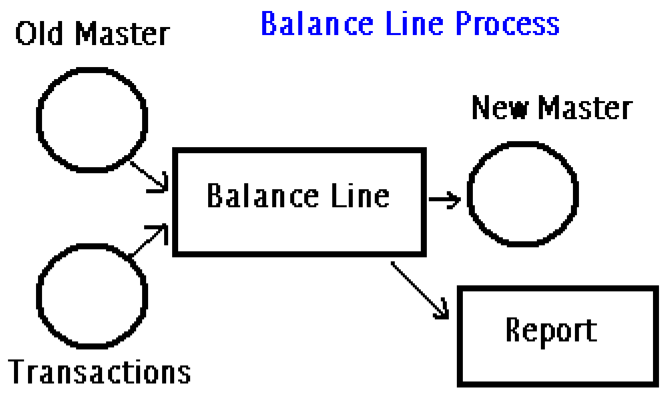
The "data-store" in a data processing application usually contains items known as a Master File and a Transaction File and sometimes other files. Each of these files is composed of entries known as records. The master file records usually contain the most up-to-date information about that entry; for example, a bank master file record might contain items such as account number, account name, address, social security number, current balance, year-to-date interest and possibly other things. The master file should have one record for each account, and the records would be arranged within the file in some specific order, say in order of increasing account number. The transaction file is just what its name implies, a set of transactions or activities over some period of time (such as one day), one record per activity. A transaction record for a bank might contain items such as account number, and the amount of a deposit or withdrawal. The transaction file will not contain a record corresponding to each master record (each bank account is not used each day). The problem is to incorporate the information contained in the transaction file into the master file. This is called "updating" the master file, and requires a merge operation.

The process called the **balance line algorithm** is a standardized approach to updating sequentially ordered master files by one or more transaction files ordered in the same sequence. This process is run in a "batch" mode, that is, all in one batch. It would not make sense to use this process in an interactive environment since, as will be seen, the entire file would have to be processed to make a single change. The performance of such a process would most likely not be acceptable.

The balance line algorithm requires at least three files: a transaction file, an original master (often called the old master) and a new master. The new master is the updated version of the old master. How are these files created? The old master is the new master from the previous execution of the process (or an empty file for the first execution). The transaction file might be created using a text editor, or by some other application program. There are several reasons why the "old" master is not updated directly:

- If the old master is on magnetic tape, it is not possible to "rewrite" a record (on a disk this may be done, but see the items that follow).
- If the old master were updated directly and it was discovered that the update was incorrect, the update would have to be "undone" before the correct run could be made.
- If a hardware or software failure occurs during the run, one is left with a partial update, leading to inconsistencies in the data.
- One always wants a "back-up" file so if a tape breaks or is lost you aren't "out of business".

Thus, during an update the program copies the old master onto the new master while incorporating the information in the transaction file. It is also reasonable to expect some kind of printed output to be produced so the user of the program can "see" that everything is correct. Printed output may be items such as bills, checks or other types of forms. When the update is complete, the old master becomes a back-up file and the new master is ready for use as the "old" master for the next set of transactions.



The old master and transaction files must be sorted (see the next section) in ascending order on the same key field (part of the record). The key in the transaction file is called the transaction key and the one in the master file is called the master key.

Transactions to be applied to the old master consist of three types: additions of new records, deletions of existing records, and updates to existing records. A single transaction can affect only one old master file record, but several transactions may be applied to the same old master record. Transactions are applied only when the transaction key value matches the master key value. In essence the balance line algorithm as described here is an implementation of the record-level CRUD rules for a sequential organization. The process is basically a copying (Read) operation with changes (Create, Update, Delete) driven by the transaction file. The major task of the balance line algorithm is to synchronize the interaction of the master and transaction records so that the appropriate actions can be performed.

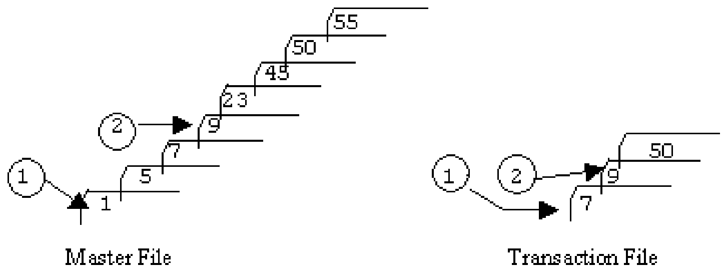
The process of merging two sequential data files requires the following information:

- The location of the data files
- The exact layout of the records that make up the files
- The field (known as the sort key) on which each file is sorted (so that the comparison may be made). Remember that a record contains no metadata information; it is too dumb to know anything about some special field.
- The transaction sort key should be used as the indicator that master file records should be read (except for the first master record which must be read at the start)

Simplified Example

Consider a simplified example composed of the two files shown below. The simplification is that the transaction file only contains Update data, so no new records will be added and no old records will be deleted. And this example does not create a new master file. In a database when you are "combining" two "files", this simplified algorithm (walking over two sorted lists and matching up key values, called a **join** operation) is very typical of the kind of process used - take CSCI 443 next fall!.

Back to our example:



Merge Process

The numbers on the records are sort key values. At the start of the merge process one record is read from each file. The arrows marked (1) indicate the records just read. The transaction with key of 7 has been read. The process compares the transaction key value with the key value of the first master record read. (Note: If the master key value is ever greater than the transaction key value, an error has occurred.) Because the master key is not equal to 7, the next master record is read. Again, no match - another master record is read, this time giving a match of key values (both are 7). The transaction record with key value 7 is processed. The (2) arrows indicate the next record in each file. Since the last transaction record has been processed, a new transaction record is read with a key value of 9 and the search for a match from the master file is continued. (Keep in mind that there might be two transaction records with a key value of 7, so don't be in too big a hurry to advance through the master file.)

When the end of the transaction file is reached, the rest of the old master file is copied onto the new master file. This may be accomplished with the same search process described above by setting (when the end of the transaction file is reached) the transaction key to a value larger than could ever occur as a master key value, so records will continue to be read from the master file in an attempt to "catch up" to the transaction key value. The end of the master file terminates the merge process.
End of Simplified Example

The "report" shown in the figure could be an exception report (indicating only problems, e.g., updates with transaction key values and no matching master key value, out of order keys, ...) or a full listing of the activities that were carried out by the process (e.g., printed values for the old master, transaction and new master), or something in between.

As noted, both the old master file and the transaction file must be sorted in key order. It's time to talk (again) about sorting.

Sorting

Sorting has to do with the **order** of items, not their organization. Sorting is a generic process and not limited to any particular file organization. We've spent quite a large part of the semester on algorithms for sorting an array of data items in main memory (**internal sorting algorithms**). Now we need approaches for sorting a file too large to be brought into main memory all at once (**external sorting algorithms**).

Sorting consists of ordering the records in a sequence based on the values of some part(s) (i.e., fields) of each record. The part(s) is called the sort key. As discussed previously, the exact description of the sort key is not part of the record but is part of the program involved in the sorting process (there is insufficient metadata in the sequential organization to "remember" the description of the sort key).

Sorting is discussed here in the section concerned with sequential file organization for two reasons: when the items to be sorted cannot be retained in memory (i.e., there are too many to be sorted internally) files are needed. The files that are needed are often ordered, that is, already sorted, so that their "entry order" is their "sorted order". Sequential files are best for this type of application.

First the ideas of merge/sort are discussed. Then a few comments are made about sort packages. Sort techniques involving other file organizations (relative and indexed) will be discussed later.

Merge/Sort

The task to be considered here begins with a set of items that are to be sorted, that is, placed into some specified order. It is assumed that these items are stored in a file (on a persistent medium). It is also assumed that there are a large number of these items. The question is how to get these items into order?

Nearly all techniques for file sorting involve the following three steps:

1. An internal sort is used to produce a number of sorted sublists, called runs or partitions, from the original input file.
2. A merge process is used to combine the sorted runs into a single sorted run.
3. An output step copies the sorted run to its final destination (file).

Several comments are in order. Step 3 is usually trivial as you will end up with one sorted file. Step 2 is, as we will see, virtually the same process used in the internal merge sort where two sorted sublists were merged into one sorted list by walking through the two sublists and picking the smaller of the two current values, one from each list. Notice that memory size is not an issue here. Even if you have two (or more) giant sorted sublists, you only have one element from each sublist in RAM at any one time.

Step 1 is the starting point for the overall sort process. A way is needed to build the initial sorted files that are to be merged.

There are many possible internal sorting processes that can be used for Step 1 (some were discussed earlier this semester). Step 1 amounts to repetitively reading some of the items from the file into RAM, sorting them using an internal sorting technique, and writing the result to other files. Surprisingly, one of the more important issues is the size of the runs and how many to have on each of the working tapes. The limit here is, of course, the size of RAM. The runs need not be of equal length for the merge processes to function properly.

There are two ways to illustrate the details of these sort/merge processes. One involves specific items (numbers) that are to be sorted, the other deals with runs and describes the algorithms in terms of these runs. The former shows detail, the latter emphasizes the large size problem. Both are important and both will be used in this discussion. The first approach (showing specific numbers) will be used in the example that follows, but think of each number shown below as hundreds or thousands of items (too many to sort all at once internally). The starting point for the examples is the 20 items:

57 17 30 69 24 53 50 95 92 15 73 89 43 70 62 37 40 56

Assume these items are stored on a sequential device, for example, a magnetic tape. (The purpose of this last comment is to say that **only** sequential file operations are going to be used in the processes that follow!)

First carry out Step 1. Suppose Step 1 results in the following set of 5 runs. (The symbol "|" is used to separate the runs.) Note that each run is sorted with low value first.

17 24 30 57 69 | 50 53 | 15 59 73 89 91 95 | 43 | 37 40 47 56 62 70

Different length runs have been used on purpose. The two methods to follow begin with these runs.

Natural Merge (M input files, 1 output file)

A merge that handles two input files at once is called a two-way merge; a merge that handles M input files at once is called an M-way merge. A natural merge merges M input files into one output file. Of course, the input files contain sorted runs (with possibly more than one run per file) and the output file is sorted in the same order. This process is very similar to the balance line algorithm discussed above, where each input file had only one run (the old master file was in order as was the transaction file) and only one output was expected (the new master file).

How is the sample data sorted? Consider a 2-way merge with the following distribution (the 5 runs from the sample data have been copied onto two tapes):

Tape 1: 17 24 30 57 69 | 50 53

Tape 2: 15 59 73 89 91 95 | 43 | 37 40 47 56 62 70

The first merge takes the first run from Tape 1 and merges it with the first run from Tape 2, writing the results to Tape 3. Then the second runs are merged. The third run is just copied from Tape 2 to Tape 3. The result after the first merge would be:

Tape 3: 15 17 24 30 57 59 69 73 89 91 95 | 43 50 53 | 37 40 47 56 62 70

To merge again, the information on Tape 3 must be distributed into two input files. Here, instead of doing a complete copy, half of the runs (1 in this case) are copied to another tape and the remaining are used from the original tape. So, Tape 1 is used for the second input (the first run from Tape 3 is copied to Tape 1) and Tape 2 is used for the output, thus:

```
Tape 1:  15 17 24 30 57 59 69 73 89 91 95
         ^
Tape 3:  15 17 24 30 57 59 69 73 89 91 95 | 43 50 53 | 37 40 47 56 62 70
                                            ^
```

The ^ marks the location of the first read operation. The result of this merge is:

Tape 2: 15 17 24 30 43 50 53 57 59 69 73 89 91 95

To finish the process (in this case) use Tape 2 and Tape 3 as input with Tape 1 as output:

```
   Tape 2:   15  17  24  30  43  50  53  57  59  69  73  89  91  95
             ^
   Tape 3:   15  17  24  30  57  59  69  73  89  91  95  |  43  50  53  |  37  40  47  56  62  70
                                                                           ^
```

Then

Tape 1: 15 17 24 30 37 40 43 47 50 53 56 57 59 62 69 70 73 89 91 95

and the process is complete.

The natural sort/merge can require a significant amount of copying of the data between files. The greater the number of initial runs and the lower the degree of the merge (i.e., the smaller the value of M, meaning fewer tapes), the more I/O is required.

Balanced merge (M input files, M output files)

How can the copying or redistribution seen in the natural merge be reduced? The balanced merge avoids much of the copying by distributing the output into M files rather than one file (the natural merge requires a total of M+1 files, the balanced merge requires 2M files).

Again look at a 2-way balanced merge. This requires 4 tapes. For the example data this might be:

Tape 1: 17 24 30 57 69 | 50 53

Tape 2: 15 59 73 89 91 95 | 43 | 37 40 47 56 62 70

Tape 3:

Tape 4:

The first merge takes the first run from each input tape (1 and 2) and merges them onto Tape 3. The second runs on the input tapes are merged onto Tape 4. The unpaired run is copied to Tape 3. After the first merge pass:

Tape 1:

Tape 2:

Tape 3: 15 17 24 30 57 59 69 73 89 91 95 | 37 40 47 56 62 70

Tape 4: 43 50 53

Now the roles of input and output tape are reversed: 3 and 4 are input tapes, 1 and 2 are output tapes. The results after another pass are:

Tape 1: 15 17 24 30 43 50 53 57 59 69 73 89 91 95

Tape 2: 37 40 47 56 62 70

Tape 3:

Tape 4:

Finally:

Tape 1:

Tape 2:

Tape 3: 15 17 24 30 37 40 43 47 50 53 56 57 59 62 69 70 73 89 91 95

Tape 4:

Since each step combines two runs, the number of runs after each pass is about half the number for the previous pass. If R is the number of runs and p is the number of passes required to reduce the number of runs to 1, then

$$(1/2)^p \times R \le 1$$

or, solving for p:

$$p = \lceil \log_2 R \rceil .$$

where $\lceil\ \rceil$ represents the ceiling function. In the above example, R = 5 and p = 3. For the M-way case this becomes

$$p = \lceil \log_M R \rceil .$$

so that the M-way balanced merge will require $\Theta(\log_M R)$ passes.

Polyphase Merge (M input files, 1 output file)

It is fairly easy to implement the balanced merge process. The 2M tapes are just switched back and forth from input to output. What about the case where "extra" runs are merged with "empty" runs (like the first pass in the example above)? Is it possible to take advantage of such a situation and reduce the amount of copying still further? Yes, but of course the algorithm becomes more complicated. A more important flaw in the balanced approach is that although 2M tapes are required, at any one time runs are being read from M files and being written to one file, leaving M-1 files idle.

The polyphase or multi-phase merge is an improvement over the balanced merge because it uses a total of only M + 1 tapes, just as the natural merge required, but with less copying than natural merge. The important issue here is how to distribute the initial runs onto the M input files.

Because of the complexity of this process, the second method of describing sort/merge algorithms will be employed here, that is, keeping track of runs and not the detailed items within runs. Consider again a case with M = 2, which will require 3 available tapes, and suppose there are 34 initial runs, distributed as follows:

Tape 1: 0 runs

Tape 2: 21 runs

Tape 3: 13 runs

The first merge processes runs in sets of 2, one from each tape, combining them and writing the resulting merged runs to the output tape (Tape 1). The process stops when one of the tapes is exhausted, in this case Tape 3. So, 13 sets of runs are merged creating 13 longer (merged) runs on Tape 1 and leaving 8 (of the original 21) untouched on Tape 2 The results of this merge are:

Tape 1: 13 runs

Tape 2: 8 runs

Tape 3: 0 runs

On the next pass, 8 runs are merged from Tapes 1 and 2 and written to Tape 3, resulting in

Tape 1: 5 runs

Tape 2: 0 runs

Tape 3: 8 runs

The next merge uses 5 runs from Tapes 1 and 3, writing the output on Tape 2.

Tape 1: 0 runs

Tape 2: 5 runs

Tape 3: 3 runs

Successive passes result in

Tape 1: 3 runs

Tape 2: 2 runs

Tape 3: 0 runs

then

Tape 1: 1 runs

Tape 2: 0 runs

Tape 3: 2 runs

and

Tape 1: 0 runs

Tape 2: 1 runs

Tape 3: 1 runs

and finally

Tape 1: 1 runs

Tape 2: 0 runs

Tape 3: 0 runs

at which point the final sorted output is on Tape 1. See how neatly this worked out, with 2 tapes used as input and 1 for output on each pass, and how efficiently one tape was emptied on each pass. Now look again at the run distribution numbers, excluding the empty tape:

{21, 13}, {13, 8}, {8, 5}, {5, 3}, {3, 2}, {2, 1}, {1, 1}, {1, 0}

These numbers pair represent successive values from the Fibonacci sequence! There is a total of 7 passes required, and it turns out [no proof] that this is optimal. No other distribution would result in fewer passes required to produce the final sorted output. These number sets constitute a *perfect Fibonacci distribution* of runs on the tape after each pass. [Note: you don't want a distribution where you at some point get x > 1 runs on each of the two input tapes; this would result in x runs on the output tape and two empty input tapes, which would require a process of splitting the output tape and copying some of the runs to another tape.]

It's easy to see how these pairs of values are related, but let's look at them in another way that will generalize to an M value > 2. In the following table, we start from the desired end result, that all data is in one long sorted run on a single tape. The tapes below are called Tape A and Tape B, which will represent the two non-empty tapes for each pass. As we saw, these labels will move around among the actual Tapes 1, 2, and 3.

Tape A	Tape B
1	0
1	1
2	1
3	2
5	3
8	5
13	8
21	13

Call the values in a given row of the Tape A column a_i and in the same row of the Tape B column b_i. Then the next row is constructed from the formulas

$$a_{i+1} = a_i + b_i \qquad b_{i+1} = a_i$$

Here you can see how the A column numbers are constructed as the sum of the two previous values, which is how the Fibonacci numbers are being created.

To do the general M-way merge, we need a generalization of the Fibonacci sequence. The **pth order Fibonacci numbers $F^{(p)}_n$** are defined recursively by the following rules:

$F^{(p)}_n = F^{(p)}_{n-1} + ... + F^{(p)}_{n-2} + ... + F^{(p)}_{n-p}$, for n>=p
$F^{(p)}_n = 0$, for 0 <= n <= p-2;
$F^{(p)}_{p-1} = 1$

In other words, we start with p-1 zeros, then one 1, and then each number is the sum of the preceding p values. ("Regular" Fibonacci numbers are $F^{(2)}_n$.)

Now consider a 3-way merge. In order to get a perfect Fibonacci distribution, we need to use a 3rd-order Fibonacci sequence. The merge process will require 4 tapes, one of which is empty at the start of each pass, so we will only look at the three tapes A, B, and C that are non-empty. As in the 2-way merge example above, Tapes A, B, and C move around on the physical tapes 1-4. What are the run distributions? The first row of the table below starts with the desired end, which is that one tape contains 1 long (sorted) run and the other two tapes are empty. The formulas used to build the successive rows, which produces 3rd-order Fibonacci numbers, are

$$a_{i+1} = a_i + b_i \qquad b_{i+1} = a_i + c_i \qquad c_{i+1} = a_i$$

Tape A	Tape B	Tape C
1	0	0
1	1	1
2	2	1
4	3	2
7	6	4

We'll stop here. The A column is the 3rd order Fibonacci sequence 0, 0, 1, 1, 2, 4, 7, … where each value from the 4th value on is the sum of the previous 3 values. Algebraically,

$$\begin{aligned} a_{i+1} &= a_i + b_i \\ &= a_i + a_{i-1} + c_{i-1} \\ &= a_i + a_{i-1} + a_{i-2} \end{aligned}$$

So our initial setup is 17 runs distributed as follows:

Tape 1:	7 runs
Tape 2:	6 runs
Tape 3:	4 runs
Tape 4:	0 runs

The first merge processes 4 runs in sets of three, one from each tape, combining them and writing the resulting merged (longer) runs to the output tape (Tape 4). The results of this merge are:

Tape 1:	3 runs
Tape 2:	2 runs
Tape 3:	0 runs
Tape 4:	4 runs

Proceeding, after pass 2,

Tape 1:	1 run
Tape 2:	0 runs
Tape 3:	2 runs
Tape 4:	2 runs

After pass 3:

Tape 1:	0 runs
Tape 2:	1 run
Tape 3:	1 run
Tape 4:	1 run

Finally:

Tape 1:	1 run
Tape 2:	0 runs
Tape 3:	0 runs
Tape 4:	0 runs

As in the previous example, the run distributions agree with the successive rows of the table.

The general formulas for computing the values in a table are:

$$a_{i+1} = a_i + b_i \qquad b_{i+1} = a_i + c_i \qquad c_{i+1} = a_i + d_i \qquad d_{i+1} = a_i + e_i \qquad \ldots \qquad t_{i+1} = a_i$$

It may be necessary to "pad" the data collection with dummy records or to carefully split runs to obtain the exact distributions.

Does every user of an external sort process have to implement one of these (or some other) algorithms? No. There are commercially available sorting packages.

Sort Packages

Sorting is a very frequently used process. Implementation of general, efficient sorting algorithms is a difficult task. Two approaches to "packaged" sort algorithms are considered here, the UNIX sort command, and the Windows XP/Vista sort command.

UNIX Sort Command

The UNIX sort command is a general purpose sort, which can sort files numerically or alphabetically. As with any sort process it is necessary to tell the sort process what file is to be sorted and what field(s) compose the sort key.

In UNIX a field is delimited by a <blank> or a <tab> character by default, or by some specified character. Records end with <LF> (other operating systems use <CRLF>). The syntax for the sort command is:

```
sort  [-d, -f, -n, -r] [-tc, +n.m, -n.m]  <file_name>
```

where the options are:

-d Dictionary order, using only letters, digits, and blanks to determine order.

-f Ignore the distinction between capital and lowercase letters.

-n Sort numbers by arithmetic value instead of by first digit.

-r Sort in reverse order.

-tc Sets the field separator to the character c.

+n.m Skips n fields and then m characters before beginning the comparison.
If m=0, the .m portion may be omitted.

-n.m Stops comparison after skipping n fields from the beginning plus m characters.
If m=0, the .m portion may be omitted.

As an example consider a file named *maillist* containing records of the form:

```
Smith:Jane:315 N. 1st St:Yourtown:IN:46260
Dough:John:5585 Main St:Mytown:IN:47032
Black:Anne:8805 Koa Dr:AnyTown:HI:96720
Dough:Mike:5585 Main St:Yourtown:IN:46205
```

The command

```
sort -t: +0 -1 maillist
```

sorts the file on the first field, +0 -1, that is, by last name. The command

```
sort -t: +5 maillist
```

sorts the file starting with field 5 and ending with the end of the record, that is on zip code.

The command

```
sort -t: +0 -1 +5 maillist
```

sorts so that address lines containing the same last name are further sorted by zip code.

Windows XP/Vista Sort Command

Windows XP/Vista has a sort program that can be run from the command line. Consider the text file *NamesUnsorted.txt* that contains

```
Tom     34
Fred    28
Jane    23
Susan   54
Abe     29
Betty   31
```

The command

```
      sort NamesUnsorted.txt /o NamesSorted.txt
```

produces a file *NamesSorted.txt* with content

```
Abe     29
Betty   31
Fred    28
Jane    23
Susan   54
Tom     34
```

The general syntax is:

sort [**/r**] [**/+***n*] [**/m** *kilobytes*] [**/l** *locale*] [**/rec** *characters*] [[*drive1***:**][*path1*]*filename1*] [**/t** [*drive2***:**][*path2*]] [**/o** [*drive3***:**][*path3*]*filename3*]

where

/r : Reverses the sort order (that is, sorts from Z to A, and then from 9 to 0).

/+*n* : Specifies the character position number, *n*, at which **sort** begins each comparison.

/m *kilobytes* : Specifies the amount of main memory to use for the sort, in kilobytes (KB).

/l *locale* : Overrides the sort order of characters defined by the system default locale (that is, the language and Country/Region selected during installation).

/rec *characters* : Specifies the maximum number of characters in a record, or a line of the input file (the default is 4,096, and the maximum is 65,535).

[*drive1:*][*path1*]*filename1* : Specifies the file to be sorted. If no file name is specified, the standard input is sorted. Specifying the input file is faster than redirecting the same file as standard input.

/t [*drive2:*][*path2*] : Specifies the path of the directory to hold the **sort** command's working storage, in case the data does not fit in main memory. The default is to use the system temporary directory.

/o [*drive3:*][*path3*]*filename3* : Specifies the file where the sorted input is to be stored. If not specified, the data is written to the standard output. Specifying the output file is faster than redirecting standard output to the same file.

On the same data file, the command

```
    sort /+9 NamesUnsorted.txt /o NamesSorted.txt
```

sorts on column 9 and produces

```
Jane    23
Fred    28
Abe     29
Betty   31
Tom     34
Susan   54
```

File Structures, 3

Relative Organization

Direct access devices overcome the disadvantages of sequential access by allowing items/records to be accessed independently of one another. The usual device for storing such files is a disk drive. The problem for the user then becomes how to carry out a mapping from information known to the user about some data item(s) into the platter, track and sector disk address of the item(s).

The mapping may be characterized by

f(value) $\rightarrow$ address

where "value" is some value known by the programmer (e.g., the value of some field in the record), f is the function that takes in the value and produces an address (location for the record). In general the process f can be quite complex, however, the low-level file I/O functions (e.g., the open operation) supply the starting point for the file (platter, track and sector). So the mapping problem is reduced to addressing/accessing items within the body of a file, that is, specifying a location **relative** to the start of the file.

R(value) $\rightarrow$ relative address

For this reason files of this type are called **relative organization files** or, using the IBM terminology, Relative Record Data Sets (RRDS).

> **Aside:** Using this functional notation the "function" for sequential organization is
>
> f ( ) $\rightarrow$ next item
>
> where item is either a byte (for a stream file) or a record (for a record file). The function needs no argument since in sequential organization the only operation available is to move to the next item.

Unfortunately relative organization files provide no more metadata support than sequential organization files and more information is needed, for example, record length. The only information that can be obtained is the starting location of the file (from the open function). The user must provide any other information, such as record length and number of records in the file.

C++ random file access

C++ provides a very flexible, low-level way of directly accessing items (a carryover from C, which is designed for low-level operations). C++ treats all disk files as a linear array of bytes on the disk. The open function establishes the origin or starting address for the file. Each *fstream* object maintains a file position marker, a long integer that represents an offset from the beginning of the file and that "points to" the next byte to be read from or written to the file. C++ provides functions to access and change the file position marker. All the user has to do is specify an offset value to one of the two *seek* functions (discussed below) and that will move the file position marker to the specified location so reading/writing of items can begin. No concept of a record is employed here.

Files in C++ are usually opened for input or for output by specifying an input-stream object (an instance of class *ifstream*) or an output-stream object (an instance of class *ofstream*). If you want to modify a byte in a file, you don't want to have to open the input file, read the data, close the input file, change the data, open the output file, write the changed data, and close the output file. Using an object of type *fstream* you have both read and write access to the data file at the same time, which is a speedup factor. You have to indicate as part of the open statement how you want to use the file:

```
fstream iostream;
iostream.open(filename.c_str(), ios::in | ios:: out); //both for input and output
```

In the following table, the function *seekg* positions the "get" pointer to read a value from the file. The function *seekp* positions the "put" pointer to write a value to the file. This is how the file gets switched back and forth between its input and output forms. The get pointer and the put pointer are synchronized, so that if the get pointer is advanced, the put pointer is automatically advanced to the same position in the file. The *tellg* and *tellp* functions return the current value of the get pointer or the put pointer. (If you were using a stream just for input, you could only use the seekg and tellg functions; if you are using a stream just for output, you could only use *seekp* and *tellp*). The *tell* functions therefore access the position marker and the *seek* functions change it.

seekg(offset, mode)	**mode** can be ios::beg //beginning of file ios:: cur //current get pointer ios:: end //end of file If the mode parameter is absent, the default is the beginning of the file. **offset** is the number of bytes from the mode position, and can be either 0, positive (move forward in the file) or negative (move backward in the file)
seekp(offset, mode)	Parameters as for seekg
tellg()	Returns the current value of the get pointer in the file
tellp()	Returns the current value of the put pointer in the file

Again, the structure of the file is "in the eyes of the beholder", that is, the programmer. So you must be able to figure out the offset (number of bytes) to reach the byte you want to read.

If you have a record-oriented file with fixed-length records, you don't have to "manually" count bytes. Consider the following data file called *Datefile.txt*, which can serve to demonstrate the ideas even though it is an ordinary text file.

```
07 16 1987
02 22 1996
11 01 2003
12 04 1962
04 05 1995
06 24 2005
```

and a C++ program to modify the 5th record in the file:

```
#include "utility.h"
#include "fstream"
#include <iomanip>

class Date
{
public:
        int getDay();
        int getMonth();
        int getYear();
        void setDay(int newDay);
        void setMonth(int newMonth);
        void setYear(int newYear);
        friend fstream& operator >> (fstream& outs, Date &x);
        friend fstream& operator << (fstream& ins, const Date &x);
private:
	int month;
	int day;
	int year;
};

int main()
{

	Date Birthdate;
	string filename = "Datefile.txt";
	fstream iostream;
	iostream.open(filename.c_str(), ios::in | ios:: out);  //for both input and
	                                                       //output
	if(iostream.fail())
	{
		cout << "File not found" << endl; ;
		exit(1);
	}

	iostream.seekg(4*sizeof(Date));         //to read record 5
	//iostream.seekg(0);                    //to read first record

	iostream >> Birthdate;                  //read the record

	cout << setfill('0');
	cout << setw(2) << Birthdate.getMonth() << " "
	       << setw(2) << Birthdate.getDay() << " "
	       << setw(4) << Birthdate.getYear() << endl;

	Birthdate.setDay(15);                   //modify the record

	iostream.seekp(4*sizeof(Date));         //reset the write pointer
	iostream << Birthdate;                  //write the new record

	iostream.seekg(-long(sizeof(Date)), ios::cur);  //reset the read pointer
	iostream >> Birthdate;                  //read the new record

	cout << setfill('0');
	cout << setw(2) << Birthdate.getMonth() << " "
	       << setw(2) << Birthdate.getDay() << " "
	       << setw(4) << Birthdate.getYear() << endl;

	iostream.close();
	return 0;
}

int Date::getDay()
{
	return day;
}

int Date::getMonth()
{
	return month;
}

int Date::getYear()
{
	return year;
}

void Date::setDay(int newDay)
{
	day = newDay;
}
```

```
void Date::setMonth(int newMonth)
{
      month = newMonth;
}
void Date::setYear(int newYear)
{
      year = newYear;
}

fstream& operator >> (fstream& ins, Date &x)
{
      ins >> x.month >> x.day >> x.year;
      return ins;
}

fstream& operator << (fstream& outs, const Date &x)
{
      outs << setfill('0');
      outs << setw(2) << x.month << " "
              << setw(2) << x.day << " " << setw(4) << x.year << endl;
      return outs;
}
```

The resulting file is

```
07 16 1987
02 22 1996
11 01 2003
12 04 1962
04 15 1995
06 24 2005
```

The file is a record-oriented file where each record is an instance of the Date class. In this program, the **sizeof** function returns the number of bytes of the Date object Birthdate, so the statement

```
      iostream.seekg(4*sizeof(Date));
```

walks the get pointer over 4 record-lengths from the beginning of the file to position itself at the beginning of the 5th record. The record is read from the file using the overloaded extraction operator that was defined for the class. The record is modified, then

```
      iostream.seekp(4*sizeof(Date));
```

in a similar fashion walks the put pointer 4 record-lengths from the beginning of the file. The new record is written to the file using the overloaded insertion operator that was defined for the class. Then, because we really want to see what was actually written to the file, we "backup" one record length using

```
      iostream.seekg(-long(sizeof(Date)), ios::cur);
```

in order to get back to start of the record we just wrote so we can read it and write it to the screen. (The *sizeof* function returns an unsigned integer, so I've cast it to type long before making it a negative number.) [Aside: it's a pain to debug a program that changes the contents of a file because you have to have an extra copy of the original file and replace the output file with the original copy after each run.]

Fixed-length record organization

The more usual model used for a relative organization file is a singly dimensioned "array" on the disk. The array consists of a set of fixed length records (the elements of the array). Access to the elements/records is through a subscript. The subscript is called the Relative Record Number (RRN). The RRN represents the ordinal position of the record in the file. There are no special delimiters (such as a CRLF) between records. It is tempting to think of the model as being a 2-dimensional array on the disk, one dimension being the records and the other being the fields. Recall the organization does not provide any metadata support for fields. Fields, if they exist, are known only to the user through the source code.

Note the similarities with ordinary (internal) 1-D arrays. The elements of an array must all be the same data type. When an internal array is created, the system retains information about the starting address in memory of the array and the number of bytes used by a single array element. It does NOT keep track of the size of the array, which is why it is up to the programmer to avoid going past the end of the array. When you ask for MyArray[2], you are asking for the 3rd array element because array indexing starts with 0. The system finds the memory address of the 3rd array element by computing an offset from the starting array address:

starting address + 2(bytes/array element) = address for element 3

In a relative organization file, the records must all have the same structure. The system knows the disk address of the start of the file and finds the disk address of the 3rd record by computing an offset from the start address:

starting address + 2(bytes/record) = address for record 3

which is exactly what we did in the C++ program above.

Programming languages that support the one-dimensional array type of model (file organization) through special language syntax hide this offset computation. COBOL and FORTRAN are about the only examples. In fact, while relative organization files are an important conceptual steppingstone, a better file organization technique was developed later.

Metadata (file-level CRUD rules)

Again, the important ideas of metadata are what and where things are stored. As mentioned earlier, only Create (through the open statement) and Delete are supported at the file level.

But, in a relative organization file it is necessary to know the length of the records. It is also necessary to know how many records there are in the file. The concept of EOF makes no sense here since it is possible to jump over an EOF marker if one were present, that is, it is possible to address out of the range of the file by specifying a RRN that is too large (just like it is possible to exceed the bounds of an internal array).

Some of these metadata values often are stored in the source program text, for example from FORTRAN,

```
      OPEN(UNIT=1, FILE='EMPLOY', ACCESS='DIRECT', FORM ='UNFORMATTED',
     *              RECL = 22, STATUS = ‘NEW’)
```

where the RECL=22 is the length of the record in bytes. (Compare this with the OPEN statement in the FORTRAN example in an earlier section where RECL does not appear and ACCESS is sequential.) Neither COBOL nor FORTRAN provides a syntactic mechanism to remember the number of records in the file. What can be done?

One very useful technique is for the **user / programmer** to define the first record in the relative file to be a "control record". This record is defined to contain two pieces of information, the length of each record and the number of records (usually not counting the control record) that are currently in the file. Doesn't this violate the "array" model since now all the elements in the array do not look alike? Yes, but ... First the control record must be the same length as all the other records in the file. The problem is that the length is unknown. However, since the control record is the first record in the file, that is, it begins at the origin of the file, it is possible to open the file temporarily with a record length of say 8 bytes (two 4-byte integers) for FORTRAN (or 12 characters – two 6-digit numbers - for COBOL). This would be just enough length to read two integer values, one for the length of the records and the other for the number of records. Then the file is closed and reopened with the "right" record length (of course this technique assumes that the data records are longer than the control record so that the control record contains the two integers plus junk to get the "right" length record).

A FORTRAN sample for this might be

```
      OPEN(UNIT=1, FILE='EMPLOY', ACCESS='DIRECT', FORM ='UNFORMATTED',
     *             RECL = 8, STATUS = 'OLD')
      READ(UNIT=1,REC=1) LENGTH, NUMREC
      CLOSE(UNIT=1)
      OPEN(UNIT=1, FILE='EMPLOY', ACCESS='DIRECT', FORM ='UNFORMATTED',
     *             RECL = LENGTH, STATUS = 'OLD')
        ...

C       Test the  REC=  value in each READ to be sure it does not exceed NUMREC.
C       The  +1  is to account for the control record.
      N =  ...
      IF(N .GT. NUMREC) GO TO 1000
      READ(UNIT=1,REC=N+1) ...
      ...

      CLOSE(UNIT=1)
C          Error branch
1000  WRITE(*, 'RRN out range.')

      END
```

Note that the first record in the file is REC = 1, unlike the C++ array index that begins at 0. In the above code, the lowest value of N that can be used is 1, which will cause a read of REC = 2, the first "legitimate" record after the control record.

The idea of a “control record” is a user supplied artifice to introduce metadata into the file. It is NOT part of the file organization.

Using this technique the metadata about the file is in the file, not in the program. This is an initial step toward data independence of the program. The alternative is that the information must be in the user’s head so it can be incorporated into any source code that accesses the file.

Next consider the record-level CRUD rules.

Record-level CRUD Rules (Relative Organization)

The CRUD rules for relative organization files take advantage of the random access nature of the storage device. Create is still limited to an "append" operation (as in the sequential case). Read and Update are straightforward once the RRN is specified. Delete still poses a problem.

The examples used to demonstrate the CRUD rules employ binary format files. The same ideas work for character format files. Binary format is used most of the time for relative files for two reasons: human readability is usually not required and a timesaving occurs since the character to binary conversions needed to go between the file and memory is avoided.

Create

In a relative organization the Create rule is implemented as append. This means that the process of adding a new record extends the file and the new record is added at the end of the file. A program must be written to append the record to the file (there are no "text" editors for relative files).

To add a new record to the end of the file it is necessary to know how many records were in the file at the time of the addition. If the control record idea is used, the number of records in the file may be obtained from the control record. Then, of course, the control record must be updated to reflect the new count. Such a process might involve (using FORTRAN)

```
      READ(UNIT=1,REC=1) LENGTH, NUMREC
      NUMREC = NUMREC + 1
      WRITE(UNIT=1,REC=NUMREC+1) ...
      WRITE(UNIT=1,REC=1) LENGTH, NUMREC
```

The READ and WRITE to REC=1 is the update process for the control record. Note that this is a change to the stored metadata for the file. This indicates that record-level CRUD rules can have an effect on the metadata.

Read

The Read rule is very straightforward. Once the RRN is known, the Read operation accesses the record directly. In FORTRAN, the code

```
      N = 11
      READ(UNIT=1, REC=N + 1)
```

will read the 11th user record (record 1 is the control record) in the file.

The question is, how is the value of the RRN determined? This is discussed in the section on algorithms that follow. However, it is very important to recognize that a relative organization file can be read sequentially. This can be done with FORTRAN instructions like:

```
          DO 100 I=1, NUMREC
          READ(UNIT=1,REC=I+1) ...
     100  CONTINUE
```

This type of construction helps to answer the question of how to find the number of records in the file if the control record idea is not used. Without a control record, some kind of sentinel value must be stored in the last physical record in the file, perhaps an EMP_NUM of -999. Then a code fragment like:

```
          N = 1
     100  READ(UNIT=1, REC=N) EMP_NUM
          IF(EMP_NUM .EQ. -999) GO TO 200
          N = N + 1
          GO TO 100
      200 NUMREC = N - 1
```

could be used to set the value of NUMREC. Notice that the "special case" record now becomes the last record in the file (the sentinel record) rather than the first record in the file (the control record). The other CRUD rules have to be "taught" either to handle the sentinel record properly or to handle the control record properly. As mentioned, the concept of EOF does not apply for relative files.

Update

The Update rule is straightforward too. It is a combination of READ and WRITE operations at the programming language level. For example to Update user record 17 using FORTRAN

```
          N = 17
          READ(UNIT=1,REC=N+1) LIST1, LIST2, ...
     C          Change the value of the LIST items to be updated
          LIST1 = ...
          WRITE(UNIT=1,REC=N+1) LIST1, LIST2, ...
```

The WRITE operation performs the Update (write over the existing record). Of course once the old values are overwritten there is no way to undo the operation unless the user has made provisions to save a copy of the old record someplace else, that is to say, the WRITE operation is destructive and makes no automatic backup. No change is needed in the control record since the number of records in the file has not changed.

Delete

The Delete rule poses a problem. The question is what to do with the space that was being used by the record that is to be deleted. There are two choices, physically remove the record from the file, or logically remove it from the file.

Physical removal amounts to "rolling up" the records left in the file one "slot". Suppose record number 10, out of 15, in a file is to be deleted (removed). Record 11 is copied into slot 10, then 12 into 11, and 13 into 12, and so on. Then, of course, the control record must be read, NUMREC must be decremented by one, and the control record must be written back.

The idea behind logically deleting a record is to assign a field in the record (usually the first byte in the record) to be an "activity flag". If the flag in a particular record is set one way, say off, it indicates that the record is part of the file. To logically delete the record from the file the flag is set the other way, say on. Now, all the other CRUD rules, especially Read, must be taught to look at this flag. If the flag is on, the record is logically deleted, and it must be ignored.

Logical deletion does not free up the space, it only marks the record as deleted. It is usually the case that there is some kind of garbage collection or packing process that can be executed periodically to physically remove the logically deleted records from the file.

Any programming language that supports relative organization files can employ the physical deletion method. Some COBOL compilers support logical deletion using an activity byte set to HIGH-VALUES to indicate a deleted status.

Typical Applications

If the analogy between relative organization files and arrays is carried to the limit, anything that can be done with arrays can be done with relative files. The concern is time! Arrays in RAM operate at electronic speed (nanoseconds), while relative files operate at mechanical speed (milliseconds).

A frequent application of relative files is an update process similar to the situation described in the section on the balance line algorithm but running in an on-line environment rather than a batch environment. The basis of this process is the function R(value). Suppose the value is an employee number; then R(emp_num) must return the RRN for the record for that employee. Given the RRN, the data-level CRUD rules (described above) can be used to make the necessary adjustments to the data file.

Two concerns are: What is the exact form of the function R, and what if something goes wrong? Several different R functions are discussed in the next section on algorithms. The concepts of backup and recovery of file systems / databases belongs in a database course and will not be covered here.

Algorithms

Sorting

The importance of being able to get records in a file into some prescribed order and maintain them in that order is independent of the file organization. Different organizations provide different advantages (and disadvantages) in the ways sorting can be implemented. The direct access nature of relative organization (i.e., the array model) allows approaches to external sorting not available with sequential organizations. Basically, the simple internal array-based sorting algorithms can all be used.

Insertion Sort

The basic idea of an insertion sort is to take the next record from the unsorted file and place it in the proper position (the correct position based on its key value and the key values in the already sorted file). One way to implement this type of process using relative files is to start at the end of the sorted relative file. Create a new empty record at the end of the file (append it to the file) making a new, empty last record (Nth record). Compare the incoming key value with the key value of the second to the last record in the file (N-1st record). If the incoming key is greater, place the record in the last record (N). If the incoming key is less than the key of the N-1st record, move the N-1st record down to the Nth slot. Repeat this process, comparing the incoming, unsorted key value with the third to the last record in the file. Continue this process until the "correct" position is found. Update the control record if one is being used.

The individual worst case occurs when the unsorted key is smaller than any key in the sorted file. Then all of the records in the sorted file have to be moved, which is an $\Theta(N)$ operation. The overall worst case is when the set of "unsorted" records is actually sorted in the opposite order desired in the sorted file. Then every insertion requires that every record be moved. This leads to the conclusion that the insertion algorithm is $\Theta(N^2)$ in the worst case.

Selection sort and bubble sort algorithms could be implemented in a similar way. They would also be $\Theta(N^2)$. Please note that even though the order is the same as a similar internal sort, the time will differ significantly since the internal sort counts CPU operations (speed range 10^{-9} sec) and the external sort counts disk operations (speed 10^{-3} sec). This is a factor of a million difference in time!

More complex internal sorting algorithms such as quicksort are just too complicated to carry out at the file level. Instead, use the polyphase merge sort discussed earlier.

Searching

Searching means looking for a value in a file, more particularly, looking for a record with a target key value. We have discussed several internal array-based search algorithms, and these also can be carried out on a relative organization file structure:

- Sequential search (accessing the records sequentially)
- Short sequential search (if the file is sorted by key value)
- Binary search (if the file is sorted by key value). Recall that binary search requires knowing the high index value in the section of the array being searched, initially the entire array. This is why we need to know the number of records in the file, which will provide the maximum RRN.

You will recall that each of these internal search algorithms returned the position in the array of the target value. When applied to a sorted relative organization file, this value is the RRN of the desired record. The order of the binary search is $\Theta(\log_2 N)$. This is much better than a linear search, especially when the number of records, N, is large. But again recall that the operations being counted are disk operations.

The binary search could be used as the Read rule function in the EMPLOY example as long as the file is a relative file and maintained in sorted order by `EMP_NUM`. It could be used as the "location" part of the update process too. But what about Create (adding a new record) and Delete? Again, getting-something-for-nothing does not succeed. The file must be maintained in sorted order. So, added records must be inserted in their proper place and the deleted records should be physically removed from the file (the algorithms given above do not expect to access a "logically deleted" record).

Hashing

We also discussed hashing as an internal search technique, and it too can be applied to relative organization files. Hashing is another way of implementing the functional relationship between a key value and an address, i.e.,

$$H(\text{Key}) \rightarrow \text{RRN}$$

where H is the hash function. In the ideal case – no collisions – only one disk access would be required to retrieve the employee record given the EMP_NUM, rather than having to search for the correct record. But, in this imperfect world, a collision resolution scheme is needed. For simplicity, we'll assume a linear probing scheme.

The hash table is actually itself a relative organization file B built initially from relative organization file A by sequentially reading records from A and storing them in B. File B is a higher-order organization than a relative file organization in that the scheme is implemented "on top of" a relative organization. Do CRUD rules apply? Of course they do. What new or different properties do these rules have?

The Create rule was discussed earlier. This rule must find the correct slot for the record based on the hashed value of its key, and adjust the structure accordingly.

The Read rule was the driving force for the design of this process. The hashing scheme is designed to reduce the number of disk accesses needed to find an item (the trade-off here is the introduction of the hashing function and the more complex file structure built on top of the relative structure). Note that the Read rule has two parts. First the target key is hashed using the hash function to obtain the hashed address (RRN ultimately). But the comparison of keys (target key value to file key value) is not made on hashed key values but on unhashed key values. So finally the Read rule must look at the actual key values in any records involved in the linear probing process.

To get the efficiency of the Read rule something was sacrificed. There is no way to access the hashed structure in a "sequential" manner. The Read rule is driven by hashed values and there is no way to know what value produces what address!

Update must, in general, be implemented by a Delete followed by a Create. If the key value changes its hashed value, hence its hash address changes, the record will "move" in the structure. Non-key updates could be done in place.

The Delete rule begins with hashing a key and then uses the same type of probing followed by unhashed key comparisons used in Read. Once the desired record is found (and possibly copied for use by Update) whatever mechanism is used to denote an "empty" slot is invoked to logically remove the record from the structure.

Closing Thoughts

In the discussion of data structures, the array was found to be a very useful primitive structure, useful in modeling other, more complex structures (e.g., stack, queue, ...). As mentioned earlier, relative file organization models the concepts of an array on the disk (a direct access device). The analogy between arrays within data structures and relative organization files within file structures is not coincidental. Relative file organization is a primitive organization upon which other file organizations are based.

Depth-first and Breadth-first Graph Traversal

The following figures are used by permission of Cengage Learning from Data Structures and Algorithms in C++ by Adam Drozdek.

FIGURE 8.3 An example of application of the `depthFirstSearch()` algorithm to a graph.

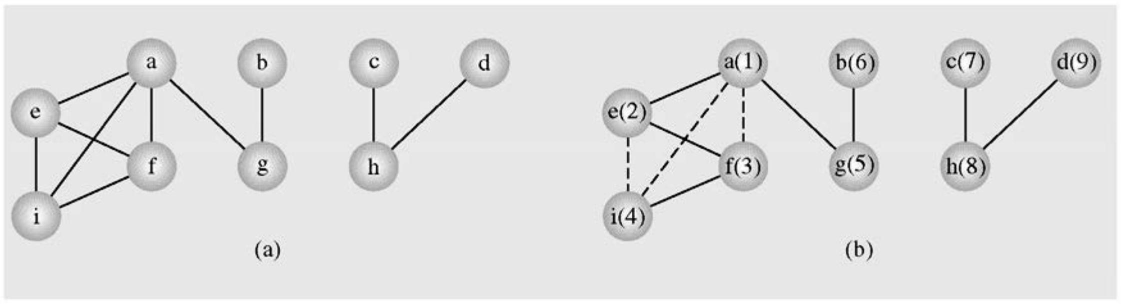
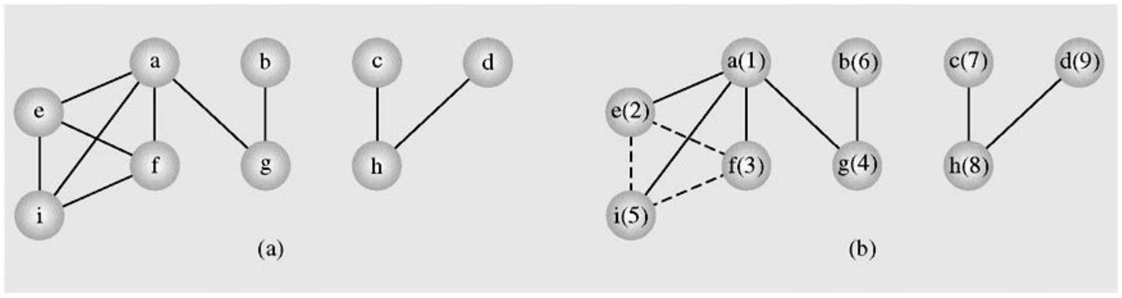


FIGURE 8.5 An example of application of the `breadthFirstSearch()` algorithm to a graph.



Graph Overview

Graphs are incredibly useful modeling tools, reflecting a variety of real-world situations. They can be represented in a computer program by the appropriate choice of data structure.

Graph theory is an extensive topic. Not only is there a great deal of mathematical theory about graphs, but there are many useful graph algorithms. We will only cover traversal algorithms, shortest path, minimal spanning tree, and topological sorting.

Graph Traversal

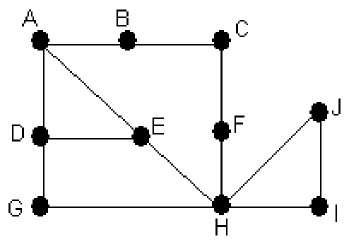
The idea of graph traversal is similar to that of tree traversal - we want a systematic procedure to visit all nodes in the graph.

There are two approaches. Using **depth-first search**, you probe as deep as possible, then back up, exploring side paths that were missed on the way down. When there is a choice of next nodes to visit, we will break ties alphabetically, although in reality this would depend on the order in which the nodes were stored in the adjacency matrix or adjacency list.

A **breadth-first search** is like a level-order traversal. You spread out from a node visiting all its adjacent nodes, then all of their adjacent nodes, etc.

We need an appropriate data structure to facilitate each traversal. Depth-first search says to drill down some path to a dead end and then back your way out. When you find a side path you missed on the way in, you follow it to the end, back your way out, etc. This suggests a stack implementation - push the nodes along the path onto a stack and pop them as you retreat back up the path. A stack, in turn, suggests recursion, and depth-first search traversal is generally implemented recursively. Breadth-first search says you process nodes in the order visited and never see them again; this is a FIFO approach (first in, first out) and suggests a queue implementation.

Example:



A DFS traversal beginning at node A gives: A B C F H E D G I J

A BFS traversal beginning at node A gives: A B D E C G H F I J

For either algorithm, we must have some mechanism to mark a node that has already been visited. Also, since the graph may not be connected, either search may not traverse the entire graph in one pass. When you come to a stopping point, you must check to see if there are any unvisited nodes - these will be in some other component of the graph, unreachable from any place you have already visited. Pick one of these and begin the traversal again.

The outline of the depth-first search algorithm is shown here:

```
depthFirstSearch()
  for all vertices v
    num(v) = 0;
  edges = empty;
  i = 1;
  while there exists a vertex v with num(v) = 0
    DFS(v);
  output edges;

DFS(v)
  num(v) = i;
  i = i + 1;
  for all vertices u adjacent to v
    if num(u) = 0
      attach edge uv to edges;
      DFS(u);
```

Because of the recursion, there is a top-level function (*depthFirstSearch*) that calls a recursive function (*DFS*). In the top-level function, there is an array of vertices called *num* that is initialized to all 0's and a "counting variable" i is initialized. A vertex with a *num* value of 0 is unvisited. Some starting vertex is passed to the recursive *DFS* function, and its num value is set to *i*, then *i* is incremented. Within *DFS*, the *for* loop walks through all vertices adjacent to the vertex that invoked *DFS*. Whenever an unvisited node is encountered, that node is used to invoke *DFS* again. When *DFS* finally climbs out of all its recursive invocations and control returns to the top-level function, it's in a while loop that is looking for any unvisited nodes. If the graph is connected, there won't be any unvisited nodes at this point, but if not, this starts the process over on another component of the graph.

This particular version of depth-first search actually does a bit more than just a depth-first traversal. For one thing, it numbers the nodes in the order in which they are visited (that's what the counting variable *i* is for.) . For another, it maintains a list of the edges traveled in the depth-first traversal. Such edges are called **forward edges (tree edges)**; edges of the graph not traveled in the depth-first traversal are called **back edges**. Because the algorithm never travels to an already-visited node, there are no cycles in the set of forward edges. The set of forward edges therefore is a subset of all the graph edges that forms a tree that touches each vertex of the graph. This is called a **spanning tree** of the graph. The following figure shows the result of a depth-first search on a graph, including the forward and back edges. This figure and the next are used by permission of Cengage Learning from Data Structures and Algorithms in C++ by Adam Drozdek. (Better pictures are in Code Archives.)

The outline of the breadth-first search algorithm is given here:

```
breadthFirstSearch()
   for all vertices v
      num v = 0;
   edges = empty;
   i = 1;
   while there exists a vertex v with num(v) = 0
      num(v) = i;
      i = i + 1;
      enqueue(v);
      while queue is not empty
         v = dequeue();
         for all vertices u adjacent to v
            if num(u) = 0
               num(u) = i + 1;
               enqueue(u);
               attach edge uv to edges;
   output edges;
```

This algorithm also numbers nodes and keeps track of tree edges. It uses a queue data structure. A starting node is visited and put into the queue. When the node at the front of the queue is processed, any unvisited adjacent nodes get numbered (thus marking them as visited) and appended to the back of the queue. When the queue is empty, the algorithm is again in the middle of a while loop looking for any unvisited vertices. As before, this condition will be true only for an unconnected graph. The following figure shows the result of a breadth-first search on the same graph as before. Note that the node numbering and the spanning tree are both different than before.

Both algorithms contain pseudocode of the form "for all vertices u adjacent to v." How you find the nodes adjacent to v depends on the implementation of the graph. If the graph is represented as an adjacency matrix, then there is some index k representing the current vertex v, and a temporary index j that must traverse row k in the matrix. If the graph is represented as an adjacency list, then there is some index k representing the current vertex v, and a temporary pointer j that traverses k's adjacency list. An advantage goes to the adjacency list representation because it is better at the task of finding all vertices adjacent to a given vertex v; the entire adjacency list for v must be traversed, but that list could be quite short (or even empty), whereas the entire row v of the adjacency matrix must be examined even if it contains many 0s.

Assuming an adjacency list representation for the graph, either algorithm eventually covers all adjacency lists. Note that for DFS, the traversal of a given list is suspended when an unvisited node u is found so that a traversal of u's adjacency list can occur. When you return from that recursive call, you resume traversing the original list from u on. Recursion maintains the value of the local pointer variable that is traversing the list so you can pick up where you left off.

Both of these algorithms would probably be implemented as member functions of a graph class. In an actual implementation, you presumably want to do something when you visit a node, so you would pass a pointer to some generic "visit" function that could do whatever you want to do at each node - print it out, for example; recall this was done for the tree traversal algorithms.

Analysis: Assume a graph with n vertices and an adjacency list representation. In either algorithm, there is Q(n) work to initialize the *num* array, but this is overshadowed by the work to traverse the adjacency lists of the graph. There are n adjacency lists, so the amount of work is at least Q(n) because each adjacency list must be checked, even if it turns out to be empty. Because there are e edges, the work in traversing the total length of all the adjacency lists is at least Q(e). Therefore the work is Q(max(n, e)). If there are more edges than nodes (the usual case), then Q(max(n, e)) = Q(e).

Hash Function

A hash function should have four properties:

- The same input should generate the same output every time. But this is inherit in the definition of a **function** (any element in the domain maps to a unique element in the codomain). So no further discussion on this matter.
- The "cost" (i.e., work, time) of finding what input value(s) produce a given hash value should be prohibitive. This is a security problem rather than a search problem, so we'll put that off for now.
- It should be simple to compute, i.e., Q(1)
- It should give an even distribution of data throughout the hash table so as to attempt to minimize collisions. However, whether this goal is achieved depends on the character of the data, which is often not known in advance.

Hash functions almost always involve performing some arithmetic operations on the key value, so if the key is originally not an integer, it must be passed through some function to map it to an integer representation. For example, using string data one approach would be to take the ASCII values of each character and add them up, then work with the resulting integer.

There are a number of hash function approaches:

1. Truncate the key value, or combine some of its digits
2. "Hash" the integer by chopping it into pieces and somehow recombine the pieces
3. Apply to the key value a modulo function relative to the hash table (array) size. (Remember that the modulo n function applied to an integer gives the integer remainder after dividing by n.) The result will be a value between 0 and table-size - 1, exactly the range of indices in the hash table. This sort of hash function gives the most even distribution throughout the table if the array size is a prime number.

These approaches can be mixed and matched, i.e., the key could be hashed and then a modulo function applied. Because the hash function has to result in a hash table index value, the modulo function is almost always used as the last step.

Example: A simple hash function is f(x) = x mod (hash table size).

So for hash table size = 10 (not a prime)

Value	Hashes to
43	3
57	7
22	2
30	0

and after inserting these values the hash table would look like:

30		22	43				57		
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

But

Value	Hashes to
40	0
50	0
20	0
30	0

Here we have lots of collisions.

The data could be hashed before the mod function is applied. For example, suppose in the second table, the digits are first reversed and then the mod function is applied. We get

Value	switch	Hashes to
40	04	4
50	05	5
20	02	2
30	03	3

which eliminates collisions. But with other data, this approach could also result in collisions.

Note that none of these schemes destroy the data, they are only used to find the array index; the original data is stored in the array.

A **perfect hash function** is one that results in no collisions. Note that this means that the hash table size must be at least the size of the data set so that each item of data can potentially have its own index in the array. But - the size of the data set is seldom known ahead of time. And even if it is, constructing a perfect hash function is nearly impossible. So...on to collision resolution.

Security

Another application of hashing quite apart from searching has to do with security. For example, you go to log on to a computer and the operating system requests your user name and password. It checks that the password you enter is on the list of authenticated users and also that the password you entered matches the password for that user ID. Options:

1. Keep the password file stored in the system in plaintext.

> But - If someone cracks into the system and steals the password file, all security is lost.

2. Keep the user ID in plaintext (customary) but encrypt the password. When the user attempts to log on, the operating system finds the user ID in the password file, unencrypts the password, and sees whether it matches the password the user entered.

> But - this means that the legitimate password for that user ID is momentarily exposed in unencrypted form and potentially could be captured. Also, it requires that the key to decrypt passwords be stored somewhere on the system, another security issue, although it is presumably not stored in the same file!

3. Keep the user ID in plaintext but encrypt the password. The operating system takes the password the user entered, encrypts it, and compares it with the password file.

> This is the most secure, although if the encryption system is something like DES, this still requires use of a key.

4. Best option - use option 3, but encrypt using a hash function. This requires no key, only application of the hash function. Many algorithms for hashing are well-known, but - assuming they are "one-way" hash functions, knowledge of the hash function used and the encrypted password, should it be discovered, does not help decryption. This is where the 2nd desired characteristic of a hash function comes into play. Furthermore, the encrypted password file need not be protected, since it is supposedly of no use to anyone.

What if 2 different passwords hash to the same value? Or 2 users (A and B) choose the same password? As far as ordinary usage is concerned, it doesn't matter because A is still authenticated as a legitimate user. But if A stole the password file, he/she would see that the hashed values are the same and would have a very good guess at B's password. If A and B do indeed have the same password, then A can log in as B and do damage.

Solution - most systems throw in a time-stamp, generally called a "salt," at password creation. The salt is combined with the password and the result is hashed. A and B, with the same password, will (likely) have different hash table entries. The system stores the time stamp (in the clear) for each user ID in the password file and when it reads the entered password, it looks up the user ID, appends the timestamp, hashes the result and compares it to the table. Knowledge of the timestamp does not help the hacker who has the file.

Hashing Overview

Hashing is yet another search algorithm. As has been the case throughout our work with search algorithms, whatever data structure we are using stores records that may be large, but each record is uniquely identified by a key value. So as usual, we'll just worry about finding a key value. The work unit is still comparison of the target against the key values (we are again assuming that all data can be stored internally, so we are not worried about disk reads).

Hashing is useful when all you want to do is search for individual items of data and ordering is not important. (We won't be able to retrieve the elements in sorted order, for example, as we can do in a BST by doing an in-order traversal). But we should be able to do a search with Q(1) work units, far better than anything we've found so far. This says that any key value can be found (or determined to be not present) with a constant amount of work. Hashing involves an array, called a **hash table**, in which the records are stored.

The constant amount of work says that given the target, we can go more or less directly to the exact location in the array where that key value would be found. Remember that one of the advantages of arrays is that they are *random-access data structures*. Given an array index, you can immediately retrieve what is stored at that index, which is never true for linked structures where you have to traverse the (list, tree, whatever) from some fixed starting point.

Consider a very simple case, where the key values 0, 1, 2, …99 are stored in a 100-element array, with a[0] = 0, a[1] = 1, etc. Given a key value, we immediately know the array index where the record with that key can be found. We have a mapping (the identity function) from key values to array indices.

f: {Key values} ® {Array indices}
f(x) = x

A **hash function** is a generalization of this idea. Given a key, I should be able to apply the hash function to that key and find the array index where the record with that key will be stored if it is present.

f(key) = index

As with the tree structures, inserting a key into a hash table and searching for a key in the hash table are virtually identical operations. Take the key and find where in the array to insert it, or take the key and find where in the array it *would have been put* and look there to see if it is present.

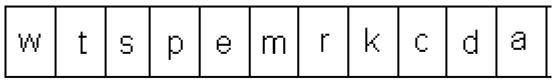
But here is the problem. There may be many more potential key values than slots in the array. In this case, the hash function will not be one-to-one, and multiple keys will map to the same index. So when you go to insert a new key value, you may apply the hash function, go to the array, and find that "your slot" is already occupied. This results in a **collision**.

So the 2 problems connected with hashing are:

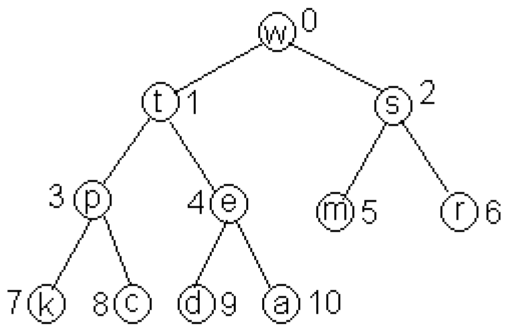
- how to choose the hash function
- how to handle **collision resolution**

Heapsort

Heapsort is a sorting method for data stored in an array. To understand how it works, however, you need to visualize the array items as representing nodes in a 2-tree. Any array can be visualized as a binary tree by making the array entries a level-order traversal of the tree. For example:



could be visualized as the binary tree



Conversely, any complete binary tree (a 2-tree where leaves are at most one level apart and the bottom level of leaves is filling from left to right) can represented as an array. You have to have the last level of leaves filling from left to right so there are no "empty slots" in the array.

In the above picture the nodes are also numbered in level order, starting with 0. Thus the node number corresponds to the array index. For any node k, its left child has the node number 2k + 1 and its right child has the node number 2k + 2. (For example, node 3 has a left child of 7 = 2*3 + 1 and a right child of 8 = 2*3 + 2.) This information enables you to find, given an array position, the array positions that represent the left and right child.

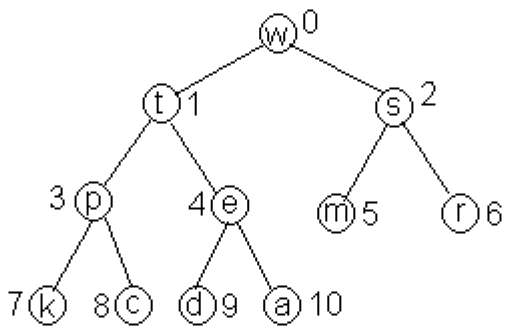
Definition: A **heap** is an array **a** in which, for any index k, the value at **a**[k] is ³ **a**[2k+1] and **a**[2k+2], provided that 2k+1 and 2k + 2 are valid array indices.

In the array above, you can see that the value at index 3, which is p, is ³ the values at indices 7 and 8, which are k and c, respectively. (Here ³ means the usual alphabetical ordering.) The same property holds for any other array index, so the original array is a heap. But this property is much easier to see in the binary tree version: the value at each node is ³ the value at its left or right child. This is why heaps are viewed as binary trees!

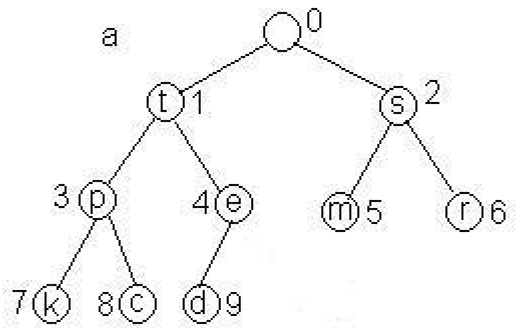
The Algorithm

Heapsort works as follows. Given an arbitrary array, think of it as representing a 2-tree. Rearrange the entries into a heap (more on how to do this later). The root (index 0) will be the maximum value. Put it in the last position of the array and take what was there (which in the tree is the rightmost leaf on the lowest level) and insert it in the rest of the array so as to preserve the heap property. Now the last element of the array is correct. Repeat on the first n – 1 array elements, etc. So you always throw the root to the end of the unsorted section and readjust the tree.

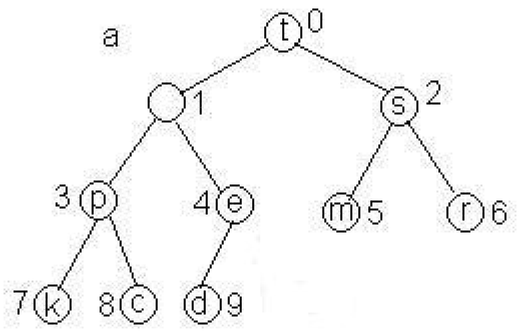
Example: Consider the tree below, which is a heap.



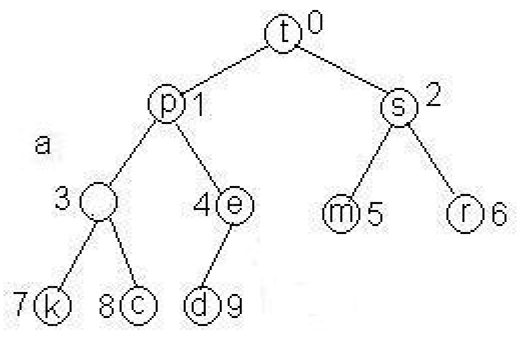
Root = w = maximum element. Put w at the end of the array and forget it. Thus w will displace a, and a will have to be inserted back into the tree.



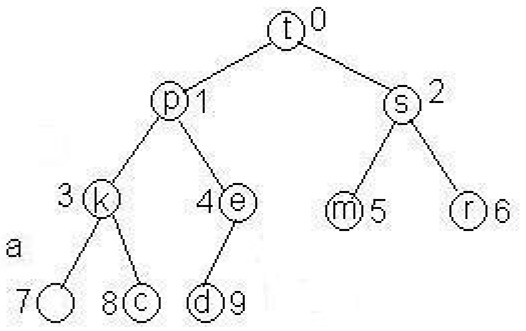
The tree now has no root, and the new root must be the maximum of the remaining elements. So take the maximum of three items: the two subtree roots and the displaced node. The displaced node was at the bottom of the tree, so it is relatively small and it will never win this three-way contest until near the very end. Here max (t, s, a) = t, make t the root. Note that comparisons of key values to each other must be done to determine this maximum, so here is where some of the work of the algorithm occurs.



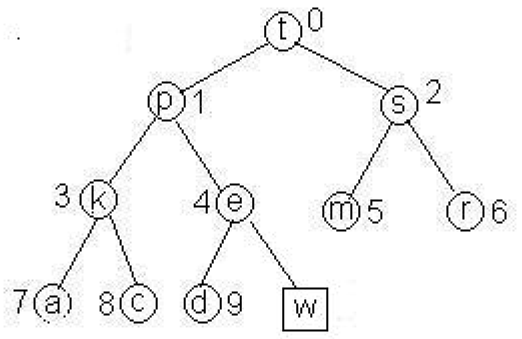
Repeat this process on the new rootless tree. Max (p, e, a) = p make it the root.



Repeat on new rootless tree. Max (k, c, a) = k, insert k.



Finally, max(NULL, NULL, a) = a. (In the code, if 2*(root index) + 1 > maximum index of the subtree, i.e., the root was a leaf, then the displaced value is inserted at the root. Thus no comparisons of key values to each other are done. There is a comparison of two array index values, but we are not counting that type of comparison.)

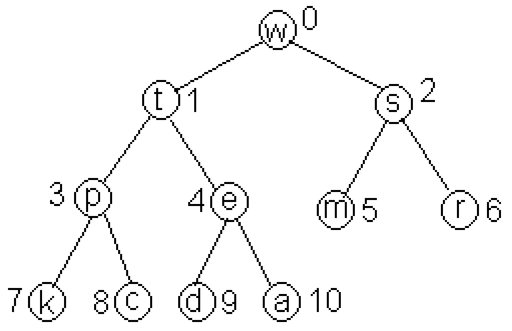


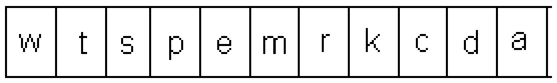
The array is once again a heap, and the process can be repeated.
End of Example

Now how does all this get carried out in the array itself? Function *moveDown*, shown below, will be invoked with *current* as the displaced item we want to insert, *low* as the tree root index – initially 0 – and *high* as the index of the end of the unsorted section of the array – initially n – 2, after the root has been thrown into the highest index, n - 1. (In the complete Heapsort algorithm, this same function will be used on sections of the array (subtrees of the whole tree), which is why the parameters are called "low" and "high".) Walk through this code for the above example. Note that by the way the indices are set, when you are working on a subtree you only see the indices for that subtree.

```
template <class List_entry>
void Sortable_list<List_entry>::moveDown(const List_entry& current, int low, int high)

//Pre: The entries of the array between indices low + 1 and high,
//inclusive, form a heap.
//Post: The entry current has been inserted into the array
//and the entries rearranged so that the entries between
//indices low and high, inclusive, form a heap.

{
    int large;                // position of child of entry[low] with the larger key

    large = 2 * low + 1;      // large is now the left child of entry[low]
    while (large <= high)
    {
        if (large < high && entry[large] < entry[large + 1])
            large++;                 // large is now the child of entry[low] with the largest key.
        if (current < entry[large])  // Promote entry[large] and move down the tree.
        {
            entry[low] = entry[large];
            low = large;
            large = 2 * low + 1;
        }
        else
            break;  //current is >= entry[large] and belongs in position low
    }
    entry[low] = current;
}
```

We have not yet addressed how you get the original random array organized into a heap in the first place. Consider again the heap as a tree and as an array.





w	t	s	p	e	m	r	k	c	d	a

The list has 11 elements, and 11/2 = 5 (using integer division in C++). The first 5 elements in the array are interior nodes, and the last 6 elements in the array are leaves. No relative order among the leaves is required in a heap. Hence we can back up from index 4 down to index 0 and insert each element x into the list from that point on to the rear. This has the effect of building the heap at the layer above the leaves and then working back up the tree.

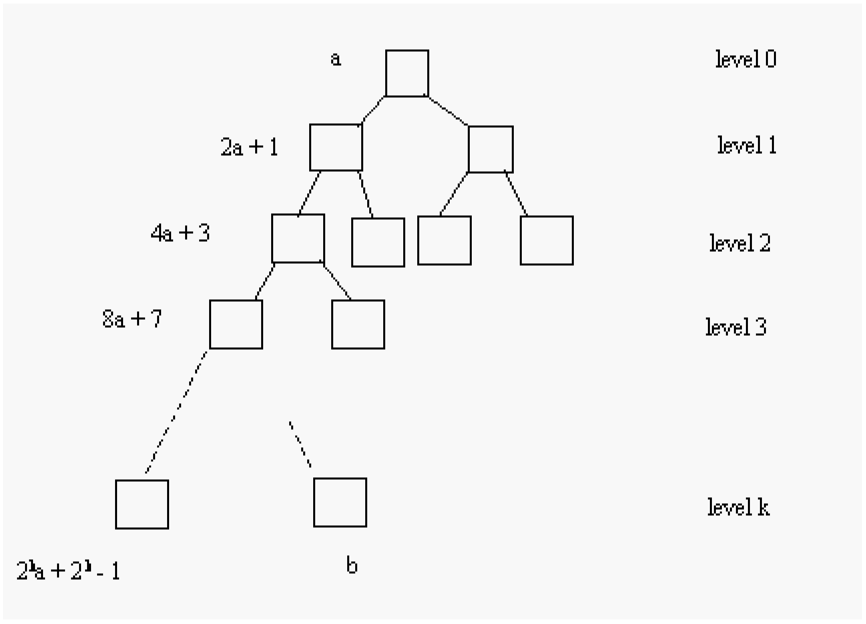
These insertions are the same operations we did when sorting the heap, and function *moveDown* can be used for this also.

Click here for an animation of heap sort.

Analysis of *moveDown*

The moveDown function is repeatedly used in heapsort, so we'll analyze this function first. The operations are once again comparisons (to find the maximum of the three values) and assignments (to move array entries around).

Consider a subtree rooted at index a with maximum index b



What is the height k of the tree? What is the largest k such that

2^ka + 2^k - 1 £ b

2^ka + 2^k £ b + 1

2^k(a + 1) £ b + 1

2^k = (b + 1)/(a + 1) (think of this as C++ integer division, which has the same effect as taking the floor function)

k = lg((b+1)/(a+1))

The *moveDown* function inserts item *current* in a subtree rooted at *low* with maximum index *high*. Such a tree has height approximately

lg((high+1)/(low + 1))

In the worst case of this function, if you have to go all the way to the bottom of the tree to complete the insertion process (as happened with "a" in the Example above) you must do 2 comparisons at each level of the subtree below the root (to find the maximum of the left child, the right child, and current), so the number of comparisons is at most

2*lg((high + 1)/(low + 1))

At most 1 assignment is made at each level of the subtree, including the root, to put the new root in place, so the number of assignments is at most

lg((high + 1)/(low + 1)) + 1

Analysis of *Heapsort*

We now know that in the worst case of the moveDown function, the number of comparisons is at most

2*lg((high + 1)/(low + 1))

and the number of assignments is at most

lg((high + 1)/(low + 1)) + 1

Now, how many times does the *moveDown* function get called in the Heapsort algorithm, and what are its parameters?

To build the heap, we begin at low = n/2 - 1 and go down to 0, invoking *moveDown* from low to the end of the array each time. So "high" is always n - 1. Therefore a given call of *moveDown* does

2*lg(n/(low + 1)) comparisons

giving a total number of comparisons of

$$\sum_{low=0}^{n/2-1} 2\lg(n/(low+1))$$

or, changing the index of summation, (k = low + 1)

$$\sum_{k=1}^{n/2} 2\lg(n/k) = 2\sum_{k=1}^{n/2} \lg(n/k)$$

Let m = n/2. Then

$$\sum_{k=1}^{m} \lg(n/k) = \lg\frac{n}{1} + \lg\frac{n}{2} + \lg\frac{n}{3} + \ldots + \lg\frac{n}{m}$$

$$= \lg\left(\frac{n}{1} * \frac{n}{2} * \frac{n}{3} * \ldots * \frac{n}{m}\right) = \lg\frac{n^m}{m!} = m\lg n - \lg m!$$

We can use an approximation (no proof) that says lg n! ~ n lg n, which we used earlier in establishing lower bounds for sorting using comparisons (reference), so we have

$$m\lg n - m\lg m$$

Because m = n/2, this expression becomes

$$\frac{n}{2}\lg n - \frac{n}{2}\lg\frac{n}{2} = \frac{n}{2}\lg n - \frac{n}{2}(\lg n - \lg 2) = \frac{n}{2}\lg 2 = \frac{n}{2}$$

This is approximately the number of assignments in the worst case, and the number of comparisons is twice this. So the process of building the heap is Q (n).

Now in the actual sorting part, the *moveDown* function is called with arguments current, 0, and k, where k goes from n - 2 down to 1. (This reflects the increasingly smaller subtree we work with as the unsorted part of the array.)

Using lg((high + 1)/(low + 1)) + 1, the number of assignments is then roughly

$$\sum_{k=1}^{n-2} \lg((k+1)/(0+1))$$

$$= \sum_{k=1}^{n-2} \lg((k+1)/1) = \sum_{k=1}^{n-2} \lg(k+1)$$

Changing base again, this equals

$$\sum_{k=2}^{n-1} \lg(k) = \lg 2 + \lg 3 + \ldots + \lg(n-1)$$
$$= \lg 1 + \lg 2 + \lg 3 + \ldots + \lg(n-1) = \lg(n-1)!$$

Using the same approximation as before,

lg(n-1)! » (n - 1)lg(n - 1) or Q (n lg n)

The number of comparisons is twice this.

So the work involved in sorting the heap, which is Q (n lg n), dominates the work involved to build the heap, which is Q (n).

This completes the worst case analysis. An average case analysis seems rather difficult to do, and at any rate the average case performance experimentally differs little from the worst case performance, so we'll stop here. We can now finish the sorting table:

Sorting Summary			
Algorithm	Worst	Average	Best
Insertion	???	random list	already sorted
compares	???	$0.25n^2$ + Q (n)	n - 1
assigns	???	$0.25n^2$ + Q (n)	0
Selection		any list	
compares	same as ->	$0.5n^2$ + Q (n)	<- same as
assigns	same as ->	3n + Q (1)	<- same as
Bubble	reverse sorted	random list	already sorted

compares	$0.5n^2$ + Q (n)	$0.5n^2$ + Q (n)	$0.5n^2$ + Q (n)
assigns	$1.5n^2$ + Q (n)	$0.75n^2$ + Q (n)	0
Smart Bubble	reverse sorted	random list	already sorted
compares	$0.5n^2$ + Q (n)	$0.5n^2$ + Q (n)	n - 1
assigns	$1.5n^2$ + Q (n)	$0.75n^2$ + Q (n)	0
Shell sort	reverse sorted	random list	already sorted
compares	---	Q ($n^{1.25}$) - experimental	---
assigns	---	---	0
Mergesort	interleaved sublists, i.e., 1, 3, ...2k –1, 2, 4, ...2k	random list	already sorted or reverse sorted
compares	Q (n lg n)	Q (n lg n)	0.5n lg n
assigns	2n lg n	2n lg n	2n lg n
Quicksort	reverse sorted or already sorted with pivot = first or last element	random list	
compares	$0.5n^2$ + Q (n)	1.39 n lg n + Q (n)	
assigns	$1.5n^2$ + Q (n)	2.07 n lg n + Q (n)	
Heapsort			
compares	Q (n lg n)+ Q (n)	---	---
assigns	Q (n lg n)+ Q (n)	---	---

Note that the average case for quicksort is equivalent to the worst case for heapsort. But the worst case for quicksort is much worse than the worst case for heapsort. So heapsort, relatively constant in its performance, avoids the unpleasant workload that quicksort can produce in certain cases.

Heapsort has the same order of magnitude as mergesort, but a higher coefficient, so it is not quite optimal. Unlike mergesort, however, heapsort has no space overhead for an array implementation. Heapsort is, all around, the best choice for large enough values of n to make the extra coding effort worthwhile.

Binary Tree Implementation

Although binary trees can be implemented in several ways, the most natural is to emulate our visualization of a tree in the language itself, allowing each node to contain not only data, but also pointers to a left and a right subtree. If a subtree is empty, the pointer has the value NULL. NULL is actually the integer 0, but using NULL emphasizes that you are talking about a pointer value, and not an integer value.

This means that a binary tree node must have three member variables holding (1) the data and (2 and 3) pointers to the left and right subtrees. Below is the interface for the Binary_node struct. Structs and classes are pretty much indistinguishable, except that the default access in a class is private whereas in a struct it is public. In practice structs are generally used when there is a data structure to be defined, but not much in the way of methods. Note that Binary_node is a template struct, so the kind of data to be stored at the node will be determined at run time.

```
//Binary_node struct interface
template <class Entry>
struct Binary_node {
//data members:

//Note that the default access in a
//struct is public, so these member
//variables are public. However,
//only private or protected member
//variables of trees will point to nodes.
Entry data;
Binary_node<Entry> *left;
Binary_node<Entry> *right;

//constructors:
Binary_node();
Binary_node(const Entry &x);
};
```

Some natural operations one might want to do on a tree structure are indicated in the Binary_tree class interface below. Note that the single member variable of a tree is a pointer to the root. That member variable and helper functions are protected rather than private, suggesting that we will have a tree class derived from this one. There are some natural operations not included here, such as node removal. These will be in the subclass, as will a new algorithm for insertion. Insertion is here to allow for testing the tree class and its various methods, otherwise you can't put anything into the tree.

Also note that most of these methods will be one-liners that invoke an auxiliary function that is recursive, again because of the recursive nature of the tree definition. Typically, the tree object invokes the method, and the method invokes the recursive function, passing to it the root of the tree and maybe some other information.

The following (partial) class definition shows all the public method declarations plus a few of the auxiliary function declarations. As you can see from the declaration comments, you will need other auxiliary functions as well, including the recursive_copy function..

```
//Binary tree class interface
template <class Entry>
class Binary_tree {
public:
Binary_tree(); //constructor
//Post: An empty binary tree has been created.
```

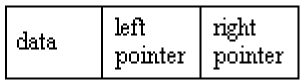
```
bool empty() const;
//Post: A result of true is returned if the binary tree is empty.
//Otherwise, false is returned.

void preorder(void (*visit)(Entry &));
//Post: The tree has been traversed in preorder sequence.
//Uses: The function recursive_preorder

void inorder(void (*visit)(Entry &));
//Post: The tree has been traversed in inorder sequence.
//Uses: The function recursive_inorder

void postorder(void (*visit)(Entry &));
//Post: The tree has been traversed in postorder sequence.
//Uses: The function recursive_postorder

int size() const;
//Post: Returns the number of nodes in the tree
//Uses: The function recursive_size

void clear();
//Post: All nodes of tree have been deleted, root set to NULL
//Uses: The function recursive_clear

int height() const;
//Post: Returns the height of the tree
//Uses: The function recursive_height

void insert(const Entry &);
//Pre: Data of type Entry has been passed for insertion -
//note that the prototype need not include parameter names,
//only data types
//Post: Parameter has been inserted into the shortest
//subtree or into the left subtree if equal height
//Uses: The function recursive_insert

Binary_tree (const Binary_tree<Entry> &original);
//copy constructor
//Post: creates a deep copy of tree original
//Uses: The function recursive_copy

Binary_tree & operator =(const Binary_tree<Entry> &original);
// overloaded assignment operator
//Post: The calling tree is reset as a deep copy of tree pointed to by original
//Uses: The function recursive_copy

~Binary_tree(); //destructor

protected:
//Auxiliary function prototypes

void recursive_preorder(Binary_node<Entry> *sub_root,
       void (*visit)(Entry &));

void recursive_inorder(Binary_node<Entry> *sub_root,
       void (*visit)(Entry &));
void recursive_postorder(Binary_node<Entry> *sub_root,
      void (*visit)(Entry &));

Binary_node<Entry>* recursive_copy(Binary_node<Entry>* sub_root);
//Pre: sub_root is NULL or points to a subtree of the Binary_tree
//Post: returns a pointer to a deep copy of tree pointed to by sub_root


//Single member variable
Binary_node<Entry> *root;
};
```

This is a good time to discuss the "fearsome threesome" - the destructor, copy constructor, and overloaded assignment operator. First, note that none of these are necessary if the class does not involve pointer variables. Hence we did not see them in our lists where we used a contiguous array implementation.

The Destructor

Objects that involve pointer variables lead to linked structures that grow dynamically by grabbing storage from the heap using the *new* operator. For example, a Binary_tree object has a single member variable that consists of a pointer to the root of the tree. The tree grows downward (via the *insert* method) by grabbing new storage for nodes and dropping left and right pointers down from leaves to point to these new nodes.

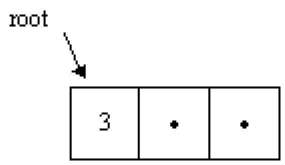
The following represents the data structure for a growing binary tree. Each node has the three member variables that are represented here as



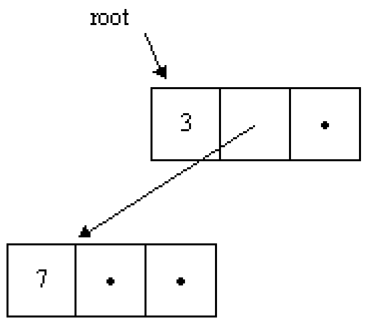
Empty tree

root �
(NULL pointer)

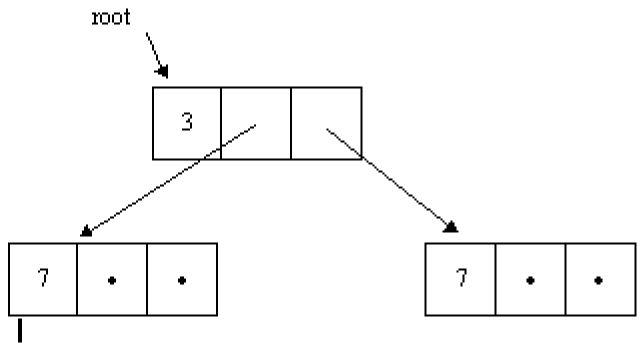
Tree with single (root) node



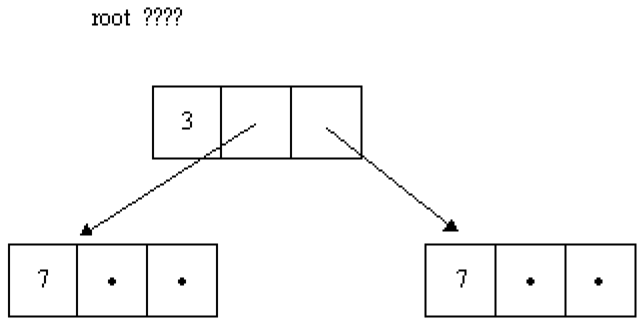
Tree with root node that has a left child



Tree with root node that has left and right child



Suppose a tree is created within a function. When the tree object goes out of scope (i.e., the function exits), then - just as is true for any local variable - the memory allocated for the tree object (its *activation record*) is returned to the heap and its data members are inaccessible. (Think of this as a default destructor at work). However, in any binary tree, the tree object's only member variable is the root pointer. The default destructor will kill the root pointer, but - see picture below - all nodes previously reachable through the root pointer have not been returned to the heap but nonetheless are now inaccessible. The destructor method does the necessary *garbage collection*. It applies the *delete* function to each node in the tree, thereby returning it to the heap, so that by the time the root is destroyed, it isn't pointing anywhere anyway.

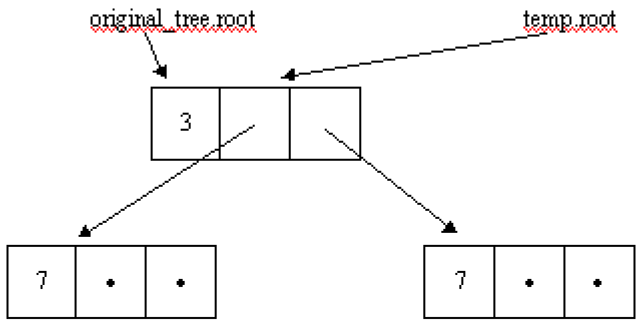


Copy Constructor

Objects, because they may represent a lot of storage, are usually passed by reference to avoid making a local copy, as is done with pass-by-value. But should you happen to pass an object by value, and without a copy constructor, here's what happens. Suppose the Binary_tree object *original_tree* is passed by value as a parameter to some function whose header is

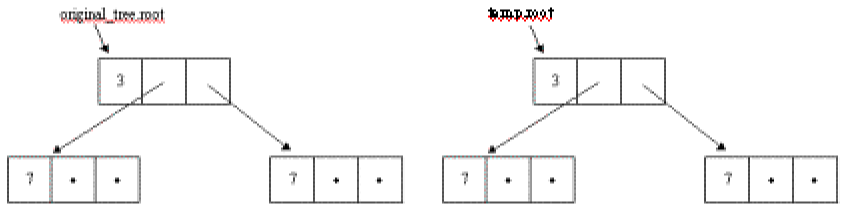
```
void some_function(Binary_tree temp)
```

The default copy mechanism sees that a local copy *temp* of *original_tree* is needed. *Original_tree*'s only member variable is the root pointer, so the default copy constructor will dutifully copy the value of the pointer variable root into a new pointer variable. The value of a pointer variable is an address, however, so that means that *temp's* root is now a pointer variable containing the same address (pointing to the same place) as *original_tree's* root.



Changes made to the *temp* tree are now made to *original_tree* as well, despite the fact that the tree was passed by value. (The pass-by-value has only protected *original_tree's* root pointer variable from change.) When the program exits *some_function*, the local object *temp* is destroyed. If you have written the destructor you were supposed to, as described above, the entire tree is destroyed, meaning that *original_tree* no longer exists. Clearly both of these side effects are undesirable.

A copy constructor creates an entire independent copy of the object. With a copy constructor, invocation of *some_function* would result in the following picture:



Now any changes to the temp tree affect only the copy, and only the copy is destroyed upon function exit. This sort of copy, where the entire structure is recreated, is sometimes called a **deep copy**. You may think you don't need a copy constructor if you never pass an object by value, but you can't prevent the client code from doing that.

Overloaded assignment operator

The system provides a default assignment operator for objects. Without an overloaded assignment operator, if tree1 and tree2 are two trees, the statement

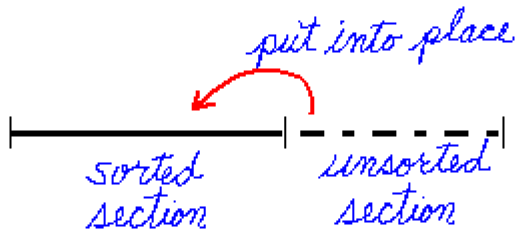
```
tree1 = tree2;
```

will assign the value of *tree2*'s only member variable, its root pointer, to *tree1*'s root pointer. *Tree1*'s root will now point to the *tree2* structure, just as *temp's* root pointed to the *original_tree* structure above. Changes to *tree1* will also change *tree2*. Once again, we want the result of such an assignment to result in *tree1* being a deep copy of *tree2*. Hence you get the idea that the copy constructor and the overloaded assignment operator will use very similar code. The only difference is that in the assignment operator, before you make a deep copy, you first need to get rid of the original *tree1* structure so that none of it remains to get mixed up with the new structure. Also consider the assignment of a tree to itself as an error and write out an error message.

Insertion Sort

The Algorithm

Here is a snapshot of insertion sort at work:



The sorted subsection of the list grows from front to back. We must have an array index to mark the beginning of the unsorted section. Pick up the next unsorted element and compare from the end of the sorted subsection to the beginning to find where to insert it. As you compare, move items 1 slot right (or the empty cell 1 slot left) so the empty cell is there to store the new item when you find where to put it.

Click here for Insertion Sort animation

The array implementation uses a derived class of the template List class that you used in Program 4. A List object has two member variables:

an array called *entry* storing data of the generic type List_entry
an integer called *count* that tells the number of meaningful values in *entry*

The derived class is called Sortable_list, and it allows additional member functions that will be the various sorting algorithms, plus any auxiliary functions.

The code for insertion sort is below. Because we are counting record comparisons and record assignments (which in our case will be key comparisons and key assignments), I have highlighted where these occur. Blue = comparison, red = assignment.

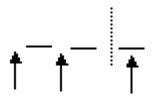
```
template<class List_entry>
void Sortable_list<List_entry>::insertion_sort()
/*
Post: The entries of the Sortable_list have been rearranged so that
      the keys in all the  entries are sorted into nondecreasing order.
*/
{
   int first_unsorted;    //  position of first unsorted entry
   int position;          //  searches sorted part of list
   List_entry current;    //  holds the entry temporarily removed from list

   for (first_unsorted = 1; first_unsorted < count; first_unsorted++)
      if (entry[first_unsorted] < entry[first_unsorted - 1]) {
         position = first_unsorted;
         current = entry[first_unsorted];         //  Pull unsorted entry out of the list.
         do {                          //  Shift all entries until the proper position is found.
            entry[position] = entry[position - 1];
            position--;                           //  position is empty.
         } while (position > 0 && entry[position - 1] > current);
         entry[position] = current;
      }
}
```

Each pass through the outer **for** loop inserts one element into its proper place. On any one pass, there is one initial comparison at the **if** condition. Assuming this condition is true, there is one assignment before the **do-while** loop and one assignment after the loop. This is a constant amount of work. Within the loop itself, there is one assignment within the loop body, and one comparison that is part of the loop termination condition. So for any one pass, the work is determined by the number of times the **do-while** loop is executed, which in turn is the number of positions you have to move forward in the list to find where to insert the first unsorted element.

Analysis

Average case: Assume the list is in random order. Consider item i, which is about to be inserted into its correct position relative to the sorted section of the list, which is of length i - 1. Eventually item i can end up in any of i positions - stay where it is, or go before any of the (i - 1) already sorted elements.

Example:



If the third item is about to be inserted, it can end up in any of 3 positions:

- stay where it is
- go before the second element
- go before the first element

end of Example

Let's first assume we have to move item i. When item i has to move, the do-while loop is executed once for each move forward. So the loop could be executed

- 1 time
- 2 times

....

- (i - 1) times

Because we are assuming that the list is totally random, any of these cases is equally likely, and has a probability of occurring of 1/(i - 1). The average number of times the loop is executed is therefore

$$\frac{1 + 2 + \ldots + (i - 1)}{i - 1} = \frac{(i - 1)i/2}{i - 1} = \frac{i}{2}$$

(making use of the formula 1 + 2 + 3 +... + n = n(n+1)/2, which you remember from CSCI 34000.) There is 1 comparison done within the loop condition, and 1 comparison done outside the loop, so to move item i requires, on the average, i/2 + 1 comparisons. There is one assignment done within the loop body, and 2 assignments done outside the loop, so to move item i requires, on the average, i/2 + 2 assignments.

For the case where item i remains in place, a single comparison is done (in the **if** condition) and no assignments are done.

To compute the average work to insert element i into its correct position, we do the usual average work formula:

Work =

$$\sum_{\text{all cases}} (\text{work for this case}) * (\text{probability of this case})$$

We have two cases: i does not have to move and i does have to move. Because there are i eventual locations i can occupy, the probability that i does not move is 1/i; the probability that i does have to move is (i - 1)/i. Therefore to insert element i into its correct position requires

Comparisons:

$$\overset{\text{no move}}{1*\frac{1}{i}} + \overset{\text{move}}{\left(\frac{i}{2}+1\right)*\frac{i-1}{i}} = \frac{1}{i}+\frac{i+2}{2}*\frac{i-1}{i} =$$

$$\frac{2+(i+2)(i-1)}{2i} = \frac{2+i^2+2i-i-2}{2i} = \frac{i^2+i}{2i} = \frac{i}{2}+\frac{1}{2} = \frac{1}{2}i+\Theta(1)$$

Assignments:

$$\overset{\text{no move}}{0*\frac{1}{i}} + \overset{\text{move}}{\left(\frac{i}{2}+2\right)*\frac{i-1}{i}} = \frac{i+4}{2}*\frac{i-1}{i} = \frac{i^2+3i-4}{2i}$$

$$= \frac{1}{2}i+\frac{3}{2}-\frac{2}{i} = \frac{1}{2}i+\Theta(1)$$

This is the amount of work to insert the ith term, and we have to insert items 2 through n. The total work is therefore (for either comparisons or assignments)

$$\sum_{i=2}^{n}\left[\frac{1}{2}i+\Theta(1)\right] = \left(\sum_{i=2}^{n}\frac{1}{2}i\right)+\Theta(n) = \frac{1}{2}\sum_{i=2}^{n}i+\Theta(n)$$

$$= \frac{1}{2}\left(\frac{n(n+1)}{2}-1\right)+\Theta(n) \quad \text{(using summation formula again)}$$

$$= \frac{1}{4}n^2+\Theta(n)$$

Conclusion: for an "average" list, the work (either comparisons or assignments) to sort by insertion sort grows as the SQUARE of the number of elements in the list.

Best case: It is easy to see from the code that the least amount of work is done if the do-while loop never executes, i.e., no elements have to be moved. This means that every element is already in its correct relative position, i.e., the list is already sorted. In this case, no assignments need to be done. There will be n - 1 comparisons (1 for each element after the first to see that it is indeed in its correct position). So insertion sort can check that a list is sorted using n - 1 comparisons. It is also true that insertion sort is very fast if the list is nearly in sorted order.

Note that any algorithm that determines whether a list is sorted by comparing pairs of elements in the list has to do at least n - 1 compares. Otherwise there is some element that is not examined and that, therefore, could be out of place.

Conclusion: The best case of insertion sort is **optimal**. [Recall that this does not mean just the best among algorithms we study, it means that insertion sort, given the right conditions, does the minimum possible amount of work of *any* sorting algorithm, known or unknown, that uses comparisons between list elements.]

Worst case:

There are two questions to address.

1. What condition(s) causes insertion sort to do the maximum amount of work?
2. How much work is done under that condition(s)?

These are questions to be answered later.

Sorting Summary

We will add to the following table as we study more sorting algorithms.

Sorting Summary			
Algorithm	Worst	Average	Best
Insertion	???	random list	already sorted
compares	???	$0.25n^2$ + Q (n)	n - 1
assigns	???	$0.25n^2$ + Q (n)	0

Lower Bounds

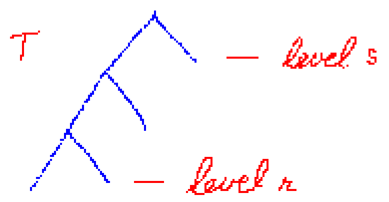
We have analyzed Binary 1 and Binary 2 searches for worst case and for average successful and unsuccessful cases. We learned that Binary 1 is more efficient than Binary 2. Our tool was a decision tree, which can be used to tell us something about *any* search algorithm, not just binary search, that is based on comparisons of the target to key values. After examining decision trees in general, we will be able to tell just how good the Binary 1 algorithm is.

To review facts about any such decision tree:

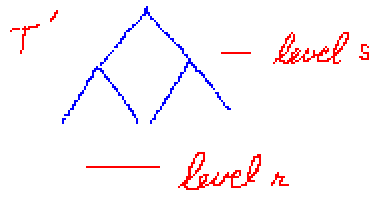
- The interior vertices represent comparisons of target against keys
- The leaves represent the search outcome (success or failure). [Remember that our Binary 2 tree was a compressed view of the tree, and that in the expanded view, the successes are indeed leaves.]
- The tree is a 2-tree, where each comparison results one of two outcomes. [Again, take the expanded view of the Binary 2 tree.]

FACT 3: In a 2-tree with k leaves, the minimum height h and the minimum external path length E occur when all the leaves are on one level or two adjacent levels. [Note that this was true for the Binary 1 tree.]

Proof: Suppose the original tree has some leaves at level s and some at level r > s + 1 (i.e., more than one level bigger.)



Clip two leaves from level r of the tree and hang them from a leaf at level s to make a new tree T'.



In this picture, the height of T' is less than the height of T. In general, this process will not always reduce the height, but it certainly won't make it any bigger, so h(T') £ h(T).

Also,

$$E(T') = E(T) \underbrace{-\ 2r}_{\text{2 removed leaves at level r}} + \underbrace{(r-1)}_{\text{new leaf at level } r-1} - \underbrace{s}_{\text{old leaf at level s}} + \underbrace{2(s+1)}_{\text{2 new leaves at level } s+1}$$

= E(T) - r + s + 1

Since r > s + 1, it follows that s + 1 - r < 0, so the above expression is

E(T) + (negative number) < E(T)

Therefore E(T') < E(T), and the external path length has decreased.

Continue this process as long as possible (until all the leaves are at most one level apart), and you have the minimum h and E.
End of Proof

FACT 4: In a 2-tree with k leaves, the height h ³ é **lg k**ù .

Proof: If all k leaves are at height h, by Fact 1, k £ 2^h and h ³ é lg kù . If the leaves are at two adjacent levels, h and h - 1, then there are even fewer leaves for the same height (or a greater height for the same number of leaves), so it is still true that h ³ é lg kù . By Fact 3, if the k leaves are separated by more than one level, the height is even greater.
End of Proof

FACT 5: In a 2-tree with k leaves, the external path length E is ³ k (lg k).

Proof: Assume that the leaves are all at one level, or at level h and level h - 1, in particular, that x of the leaves (x might be 0) are on level h - 1 and (k - x) leaves are on level h. Each pair of leaves on level h has a single parent at level h - 1, so the number of vertices (leaves and parent nodes) at level h - 1 is x + (1/2)(k - x). By Fact 1,

$$x + \frac{1}{2}(k - x) \le 2^{h-1}$$

$$\frac{1}{2}k + \frac{1}{2}x \le 2^{h-1}$$

k + x £ 2^h

x £ 2^h - k (**)

For such a tree,

E = path to leaves at level h - 1 + path to leaves at level h

= (h - 1)x + h(k - x)

= hk - x

³ hk - (2^h - k) (from **)

= k(h + 1) - 2^h

We know that if all leaves were at level h - 1, there would be 2^{h-1} such leaves. Because at least some of the leaves are at level h, there are even more leaves, so

2^{h-1} < k

If all leaves are at level h, there would be 2^h of them. Consequently,

2^{h-1} < k £ 2^h

or, taking logarithms to the base 2,

$$h - 1 < \lg k £ h$$

Adding 1 to the left inequality gives

$$h < \lg k + 1$$

Combining this with the right inequality gives

$$\lg k £ h < \lg k + 1$$

and we can write

$$h = \lg k + e \text{ , where } 0 £ e < 1.$$

Using this expression for h in our previous inequality

$$E ³ k(h + 1) - 2^{h}$$

we get

$$E ³ k(\lg k + e + 1) - 2^{\lg k + e}$$

$$= k(\lg k + e + 1) - 2^{\lg k} * 2^{e}$$

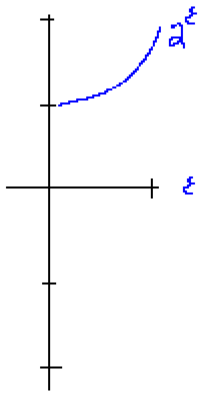
$$= k(\lg k + e + 1) - k * 2^{e}$$

$$= k(\lg k + e + 1 - 2^{e})$$

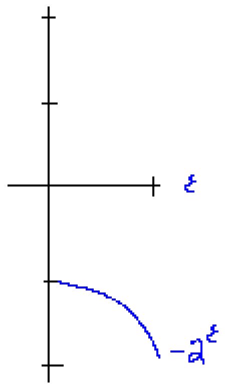
We can use a graphical argument to show that

$$e + 1 - 2^{e} ³ 0$$

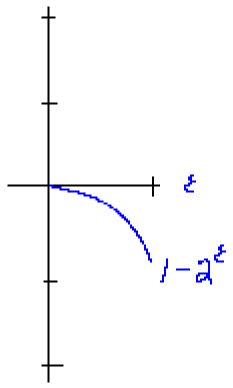
Set up a coordinate system where the horizontal axis is e . Then 2^{e} is an exponential curve; when e = 0, its value is 1, and when e is almost 1, its value is almost 2.



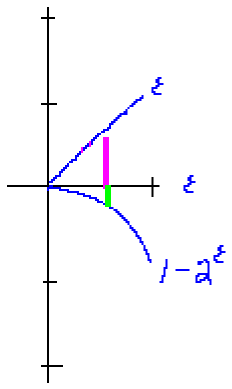
Now take the negative of this curve, getting -2^{e}



Add 1 to this curve, resulting in 1 - 2^{e}



Now add the 45° line that represents the graph of y = e .



The positive purple (e) is greater than or equal to the negative green (1 - 2^{e}), so that

e + 1 - 2 ^{e} ³ 0

and

E ³ k(lg k + e + 1 - 2^{e}) ³ k (lg k)

By Fact 3, if the tree has its k leaves separated by more than one level, then E is even greater.
End of Proof.

So what do we know? In a two tree with k leaves,

- h ³ é lg kù
- E ³ k(lg k) (and since there are k leaves, the average path length to any leaf is therefore ³ k(lg k) / k = lg k)

So any search algorithm based on comparisons has a decision tree with height h ³ é lg kù , where k is the number of leaves, and it must do at least é lg kù comparisons in the worst case. é lg kù is a **lower bound** on the worst-case number of comparisons. Any search algorithm based on comparisons has an average path length to a leaf that is ³ lg k, where k is the number of leaves. lg k is a **lower bound** on the average-case number of comparisons (for either successful or unsuccessful search).

Lower Bounds for Searching by Comparisons, k leaves	
Worst case	é lg kù
Average case	lg k

Now suppose that k = 2n. Then

worst case ³ é lg 2nù = é lg 2 + lg nù = é 1 + lg nù = 1 + é lg nù

average case ³ lg(2n) = lg 2 + lg n = 1 + lg n

On an n-element list, Binary 1 has 2n leaves (n success outcomes and n failure outcomes. Unlike Binary 2, Binary 1 does not tell us whether an unsuccessful target is less than or greater than the last element.) And these are the same values we came up with for Binary 1 worst case and average case.

Conclusion

Binary 1 is optimal. No algorithm can do better, under the assumptions we made that

1. Successful targets are equally likely to appear anywhere in the list, unsuccessful targets in any failure interval.
2. The algorithm is based entirely on comparisons of targets against keys. Surprisingly, there are other algorithms that do not require as many comparisons because they make use of special information about the data. Example: search by interpolation in a uniformly distributed list. There are comparisons, but many fewer required than binary search.

Lower Bounds

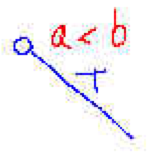
We already observed that any algorithm that works by comparing and rearranging keys must do n - 1 comparisons at a minimum in order to check that a list of n items is already sorted, and that insertion sort is optimal at this task.

What is the minimum amount of comparisons required to sort an unsorted list, as opposed to merely verifying that a sorted list is sorted? Once again, decision trees come to the rescue. This time, each internal node will represent comparisons of two key values. Leaves will represent the outcomes after the list is sorted.

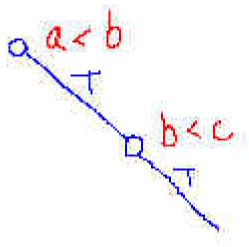
Consider a 3-element list:

a, b, c

where the relative size of these three items is unknown. Consider the version of selection sort that always puts the maximum element at the end of the unsorted section. The algorithm begins by setting a as the current maximum and then comparing a and b to see if b is larger. Suppose it is true that a < b.



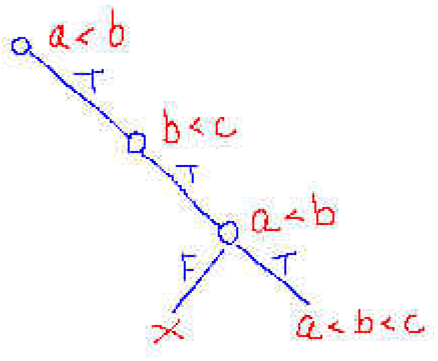
Then b is the new maximum, which will next be compared to c. Suppose it is also true that b < c.



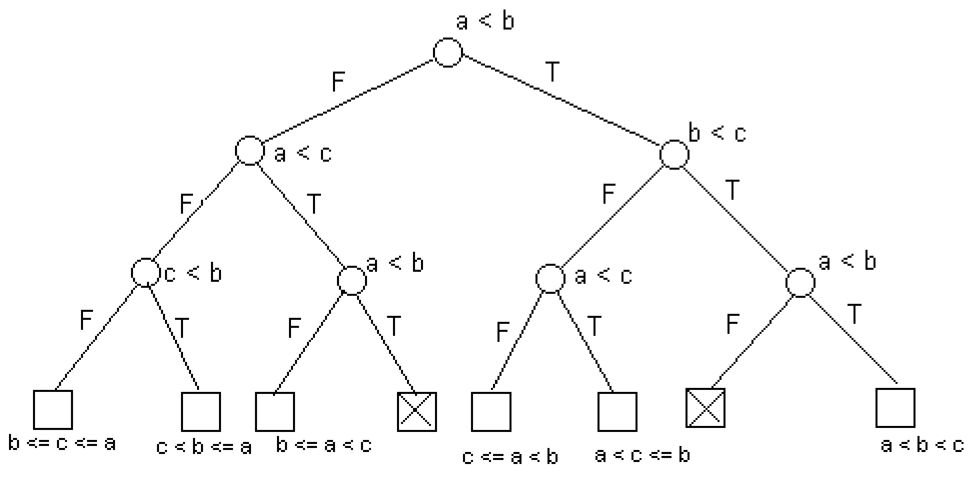
Then c is the maximum element of the three and will be swapped (with itself) into the last position in the list, so the algorithm now knows that the last (maximum) element is in its correct place.

a b | c

The algorithm begins again on the unsorted section of the list. The first element, a, is set as the current maximum and once more is compared with b. [Note the algorithm does not "remember" that it's already made this comparison once.] If it finds (again!) that a < b is true, then b is the maximum and is swapped (with itself) into the correct position. Only the single element a is left, and the list is sorted. The sorted order is a < b < c.



The X above marks an impossible case. Because a < b was found to be true at the top of the tree, then when a is again compared to b, it can't happen that a is no longer less than b. The complete decision tree for the case of a 3-element array is below, you can figure out how the rest of the branches work.



Any sorting algorithm where keys are compared can be visualized using such a tree. An algorithm may do lots of unnecessary comparisons (which we saw to some extent in selection sort). But what would the minimum number of comparisons be?

These decision trees are binary trees. The minimum height will be

$$\lceil \lg k \rceil$$

where k is the number of leaves, and the minimum external path length is k lg k (reference). The average path to a leaf is therefore at least lg k.

The leaves of the decision tree represent all possible outcomes of the sorting task, that is, all possible orderings of the n elements. There are n! possible ordered arrangements (permutations) of n elements (recall this from Discrete Math). Notice that the selection sort decision tree above has 6 = 3! leaves (not counting the impossible cases).

Summary:

Lower bounds on comparisons	
To verify that list is sorted	n - 1

	Worst case	Average case
To sort	$\lceil \lg n! \rceil$	lg n!

We can use an approximation (no proof) that says lg n! ~ n lg n. Because lg n is a smaller order of magnitude than any power of n, we can see that insertion sort, selection sort, and the two bubble sorts, all of which are $Q(n^2)$ on the average, are clearly not optimal. Hence we will continue a search for a more efficient sorting algorithm.

Mathematics Background

Introduction

This page is a review (we hope!) of topics from Discrete Structures, CSCI 34000. These topics are:

Induction

Recurrence Relations

Order of Magnitude of Functions

The P = NP Question

Induction

Mathematical induction is a method of proving some property that is true for all integers beyond some point, often for all positive integers. There are actually two induction methods.

First Principle of Mathematical Induction

$$\left.\begin{array}{l} 1.\ \ P(1) \text{ is true} \\ 2.\ \ (\forall k)[P(k) \text{ true} \rightarrow P(k+1) \text{ true }] \end{array}\right\} \rightarrow P(n) \text{ true for all positive integers } n$$

Notice that the major connective of this Principle is an implication. The conclusion is that the property (predicate) P(n) is true for all positive integers. The antecedent is the conjunction of two hypotheses, that is, both hypotheses must be true in order for the conclusion to follow. The first hypothesis is a simple statement, that the integer 1 has property P. The second hypothesis is also a statement, namely that for all (universal quantifier) integers k, a certain implication is true. The implication is that if integer k has property P, then so does integer k + 1.

To achieve the overall conclusion, one must prove that both the hypotheses are true. To prove the first hypothesis true is usually trivial. To prove the second hypothesis true, you must pick a completely arbitrary integer k, assume that k has property P, and then show that integer k + 1 has property P. This is often a source of confusion about induction. If I assume that k has property P, that is, I assume that P(k) is true, isn't this my overall conclusion? Haven't I assumed what I ultimately want to prove? No - because in hypothesis 2, you are trying to prove that an implication is true. You ASSUME that the antecedent, P(k), is true for some specific, although arbitrary k, and then prove the consequent P(k+1). This is the usual way to prove an implication, and certainly you have not assumed that P is true for all positive integers.

In doing a proof by induction, three steps are involved.

1. Prove P(1); this is called the **basis step** or **base case**.
2. Assume P(k) is true; this is called the **inductive hypothesis**.
3. Prove P(k+1) is true.

Completion of these three steps will prove that P(n) is true for every positive integer n, using the First Principle of Mathematical Induction.

One of the most common uses of induction is to prove some expression for a finite number of terms in a series. In such a proof, you must already have an expression in mind, obtained perhaps by observation; the proof verifies that your expression is correct.

Example 1: Prove that

$$1 + 2 + 2^2 + \ldots + 2^n = 2^{n+1} - 1$$

for any $n \geq 1$.

Using the First Principle of Induction, we prove the basis step. Write the above equation for n = 1.

$1 + 2 = 2^{1+1} - 1$ or $3 = 4 - 1$, which is true.

Assume P(k):

$1 + 2 + 2^2 + \ldots + 2^k = 2^{k+1} – 1$ [note that the left side is a sum of terms, it's not just 2^k]

Show P(k+1):

$$1 + 2 + 2^2 + \ldots + 2^{k+1} = 2^{k+1+1} - 1$$

Starting with the left side, we can discover the "k case" buried within the "k+1 case".

$1 + 2 + 2^2 + \ldots + 2^{k+1}$	left side of P(k+1)
$= 1 + 2 + 2^2 + \ldots + 2^k + 2^{k+1}$	writing the next-to-last term in the summation
$= 2^{k+1} - 1 + 2^{k+1}$	using the inductive hypothesis
$= 2*2^{k+1} - 1$	adding like terms
$= 2^{k+1+1} - 1$	adding exponents

This is now the right side of P(k+1), so we are done.
End of Example 1

The First Principle of Induction can be used whenever it's possible to deduce the truth of P(k+1) from the truth of P(k). But sometimes the truth of P(k+1) depends not on P(k) but on knowledge about integers "farther back" than just the previous integer k. Then you can use the Second Principle of Induction.

Second Principle of Mathematical Induction

$$\left.\begin{array}{l} 1'.\ \ P(1)\text{ is true} \\ 2'.\ \ (\forall k)[P(r)\text{ true for all } r, \\ \quad\ 1 \le r \le k \rightarrow P(k+1)\text{ true }] \end{array}\right\} \rightarrow P(n)\text{ true for all positive integers } n$$

Compared with the First Principle, the overall conclusion is the same and so is the base case. But hypothesis 2 is different. Here you get to assume that property P is true for all integers between 1 and k, and you can use any or all of those assumptions to prove P(k+1).

The two Principles are equivalent, but this is not immediately apparent because the Second Principle seems to give us so much more "ammunition" to prove P(k+1).

Example 2: Prove that any amount of postage greater than or equal to 8 cents can be built using only 3-cent and 5-cent stamps.

Here we let P(n) be the statement that only 3-cent and 5-cent stamps are needed to build n cents worth of postage, and prove that P(n) is true for all $n \ge 8$. The basis step is to establish P(8), which is done by the equation

$8 = 3 + 5$

But, this statement about 8 being a sum of 3's and 5's doesn't help at all to prove that 9 is a sum of 3's and 5's. (Never mind that you know that 9 = 3*3 + 0*5, you can't prove that based on what you know about 8.) Knowledge about 8 also won't help you prove anything about 10. But it will help you prove that 11 is a sum of 3's and 5's because 11 is 3 more than 8. If you can write 8 as a sum of 3's and 5's, then adding one more 3 would give you 11. In fact, you can prove that any integer is a sum of 3's and 5's if you know that the integer 3 units back can be so written. Looking just one position behind you doesn't help, so we will need the Second Principle of Induction.

Because the first place where the induction idea begins to work is 11, w will need to establish two additional base cases, P(9) and P(10), by the equations

9 = 3*3 + 0*5

10 = 0*3 + 2*5

Now we assume that P(r) is true for any r, $8 \le r \le k$, and consider P(k + 1). We may assume that k + 1 is at least 11, since we have already proved P(r) true for r = 8, 9, and 10.

If k + 1 ≥ 11, then (k + 1) - 3 = k - 2 ≥ 8, and by the inductive hypothesis, P(k - 2) is true. Therefore k - 2 can be written as a sum of 3s and 5s, and adding an additional 3 gives us k + 1 as a sum of 3s and 5s. This verifies that P(k + 1) is true.

End of Example 2.

Recurrence Relations

A mathematical sequence consists of a first value S(1), a second value S(2), etc. Such a sequence is sometimes defined recursively by giving the value for S(1) and then giving the value for larger integers n in terms of the value of S(n-1).

Example 3: Here is a recursive sequence.

S(1) = 4	**base case**
S(n) = 2S(n-1) + 3 for n ≥ 2	**recurrence relation**

End of Example 3.

It's easy enough to list the first few values in the sequence S of Example 3. They are:

4, 2*4 + 3 = 11, 2*11 + 3 = 25, ...

4, 11, 25, ...

But what if you want to find the value of S(75)? You certainly don't want to compute all of the first 75 terms! What you want is a **closed-form solution**, a formula, into which you can just plug 75 and get the answer. The process of finding the solution is called **solving the recurrence relation**.

The recurrence relation of Example 3 matches the following form:

S(n) = cS(n-1) + g(n)

subject to the known base case

S(1) = some known value

This type of recurrence relation is known as a **linear, first-order recurrence relation with constant coefficients**. It is first-order because the next value of S(n) depends only on the previous value S(n-1). It is linear because S(n-1) appears only to the first power in the recurrence relation. It has constant coefficients because the coefficient of S(n-1) is a constant.

There is a formula for the solution of this type of recurrence relation. It is

$$S(n) = c^{n-1}S(1) + \sum_{i=2}^{n} c^{n-i} g(i)$$

You can look at a linear, first-order recurrence relation with constant coefficients, figure out the constant c and the function g(n), and plug values into this solution formula. However, this won't give you a closed-form solution. You'll have to be able to find the sum of the series, usually by induction, to get a true closed-form solution.

Example 4: Solve the recurrence relation of Example 3.

Here's the recurrence relation:

$S(n) = 2S(n-1) + 3$ for $n \geq 2$

and here's the linear, first-order recurrence relation pattern:

$S(n) = cS(n-1) + g(n)$

Comparing these, it is clear that the constant coefficient c = 2, and that the function g(n) is the constant function g(n) = 3. The fact that g(n) is a constant function means the value of g is 3, regardless of the argument value. We also know from Example 3 that the base case S(1) = 4. Now plug all this information into the solution formula

$$S(n) = c^{n-1}S(1) + \sum_{i=2}^{n} c^{n-i} g(i)$$

and you get

$$\begin{aligned} S(n) &= 2^{n-1}(4) + \sum_{i=2}^{n} 2^{n-i}(3) \\ &= 2^{n-1}(2^2) + 3\sum_{i=2}^{n} 2^{n-i} \\ &= 2^{n+1} + 3[2^{n-2} + 2^{n-3} + \cdots + 2^1 + 2^0] \end{aligned}$$

If this were the end of the story, you would be no better off than before, you'd still be counting up to the value you want. But, thanks to Example 1, you also know that

$$1 + 2 + 2^2 + \ldots + 2^n = 2^{n+1} - 1$$

which is the same as

$$2^0 + 2 + 2^2 + \ldots + 2^n = 2^{n+1} - 1$$

and this is the sum inside the square brackets, only turned around and with the highest power of 2 equal to n -2 instead of n. So you can write the complete closed-form solution:

$$S(n) = 2^{n+1} + 3[2^{n-1} - 1]$$

and if you want S(75), it is $2^{76} + 3[2^{74} - 1]$

End of Example 4.

There is another type of recurrence relation where the value of S(n) is not dependent on the value of S(n-1) but on a value halfway back in the series, S(n/2). [Assume that n is a power of 2 so it can be cut in half as many times as desired with an integer result each time.] The general form of such a recurrence relation is

$$S(n) = cS\left(\frac{n}{2}\right) + g(n) \text{ for } n \geq 2,\ n = 2^m$$

This is called a **divide-and-conquer recurrence relation** because it occurs frequently in the analysis of divide-and-conquer algorithms that solve a problem by breaking it into smaller versions, each half the size of the original. There is a solution formula for this type of recurrence relation as well:

$$S(n) = c^{\log n} S(1) + \sum_{i=1}^{\log n} c^{(\log n)-i} g(2^i)$$

Here log n means log to the base 2, so that $2^{\log n} = n$.

Example 5: Solve the recurrence relation

$$T(1) = 3$$
$$T(n) = 2T\left(\frac{n}{2}\right) + 2n$$

This is a match for the general divide-and-conquer recurrence relation, where c = 2 and g(n) = 2n. Therefore $g(2^i) = 2(2^i)$. Substituting into the solution equation gives the following, where we use the fact that $2^{\log n} = n$.

$$T(n) = 2^{\log n} T(1) + \sum_{i=1}^{\log n} 2^{\log n - i} 2(2^i)$$

$$= 2^{\log n}(3) + \sum_{i=1}^{\log n} 2^{\log n + 1} \quad \text{(adding exponents)}$$

$$= n(3) + (2^{\log n + 1})\log n \quad \text{(pulling the constant } 2^{\log n+1} \text{ out of the summation)}$$

$$= 3n + (2^{\log n} \cdot 2)\log n$$

$$= 3n + 2n \log n$$

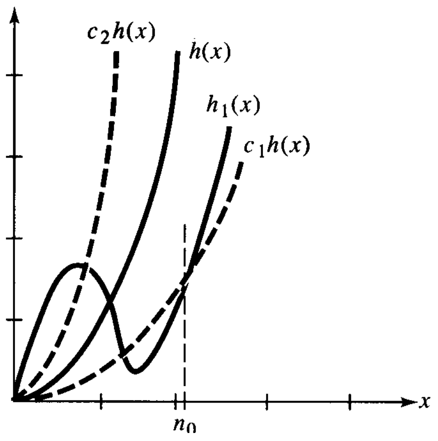
End of Example 5.

Order of Magnitude of Functions

Order of magnitude is a way of comparing the "rate of growth" of different functions. We know, for instance, that if we compute $f(x) = x$ and $g(x) = x^2$ for increasing values of x, the g-values will be larger than the f-values by an ever increasing amount. This difference in the rate of increase cannot be overcome by simply multiplying the f-values by some large constant; no matter how large a constant we choose, the g-values will eventually race ahead again. Our experience indicates that the f and g functions seem to behave in fundamentally different ways with respect to their rates of growth. If you were to graph the two functions, f(x) is a straight line and g(x) is (half of) a parabola. On the other hand, if $h(x) = 3x^2$, then g(x) and h(x) grow at approximately the same rate; if you were to graph these two functions, they have the same parabolic "shape". They are the same order of magnitude.

Let f and g be functions mapping nonnegative reals into nonnegative reals. Then f is the same **order of magnitude** as g, written $f = \Theta(g)$, if there exist positive constants n_0, c_1, and c_2 such that for $x \geq n_0$, $c_1g(x) \leq f(x) \leq c_2g(x)$. [Θ is the capital Greek letter theta]

What does this definition mean? It says that beyond some point n_0, the shape of the two curves is roughly the same. In the figure below, h(x) and $h_1(x)$ are the same order of magnitude. Ignore what happens for values of x less than n_0. The coefficients c1 and c2 create an "envelope" around the h(x) function, following its general shape, and beyond n_0, the function $h_1(x)$ stays within this envelope and so must follow h(x)'s shape as well.



Example 6: Prove that $f(x) = 3x^2$ and $g(x) = 200x^2 + 140x + 7$ are the same order of magnitude. Using the definition, we must find values for n_0, c_1, and c_2 that work. It's easy to get

$$f(x) \leq c_2g(x)$$

by just picking $c_2 = 1$ because

$$3x^2 \leq 200x^2 + 140x + 7$$

everywhere. For

$$c_1 g(x) \le f(x)$$

we have to pick a small c_1 so as to overcome the coefficient of 200 in g(x). Here's one choice: pick $c_1 = \frac{1}{100}$. Then

$$\frac{1}{100}(200x^2 + 140x + 7) = 2x^2 + 1.4x + 0.07, \text{ which is } \le \ 3x^2 \text{ for } x \ge 2.$$

So let $n_0 = 2$, $c_1 = \frac{1}{100}$, and $c_2 = 1$. Then for $x \ge 2$,

$$\frac{1}{100}(200x^2 + 140x + 7) \le 3x^2 \le (1)(200x^2 + 140x + 7)$$

But the values for n_0, c_1, and c_2 are not the only ones that work. We could pick $n_0 = 1$, $c_1 = \frac{1}{200}$, and $c_2 = 1$. Then for $x \ge 1$,

$$\frac{1}{200}(200x^2 + 140x + 7) \le 3x^2 \le (1)(200x^2 + 140x + 7)$$

End of Example 6

In general there is an infinite number of triples (n_0, c_1, c_2) that will work if f = Θ(g). There is no "best" set of values, you just need one set that works.

But there's an easier way to prove that f = Θ(g) by using some ideas from calculus. If f(x) and g(x) grow at the same rate beyond some point, then

$$\lim_{x\to\infty}\frac{f(x)}{g(x)} = p \quad \text{where p is a positive real number}$$

As an aid in finding the limit of a quotient, if $\lim_{x\to\infty} f(x) = \infty$ and $\lim_{x\to\infty} g(x) = \infty$ and f and g are differentiable functions, then L'Hôpital's rule says that

$$\lim_{x\to\infty}\frac{f(x)}{g(x)} = \lim_{x\to\infty}\frac{f'(x)}{g'(x)}$$

So for $f(x) = 3x^2$ and $g(x) = 200x^2 + 140x + 7$,

$$\lim_{x\to\infty}\frac{3x^2}{200x^2+140x=7}=\lim_{x\to\infty}\frac{6x}{400x+140}=\lim_{x\to\infty}\frac{6}{400}=\frac{6}{400}$$

which is a positive constant.

The Louvre in Paris has a huge central courtyard where, on the surrounding buildings, there are sixty-six statues of famous Frenchmen from the ages. Here is the picture I took on a 2010 trip of the statue of L'Hopital:

Some other order of magnitude relationships exist. $f = O(g)$ [f is **big oh** of g], if there exist positive constants n_0 and c such that for $x \geq n_0$, $f(x) \leq cg(x)$. This means that g(x) eventually grows at least as fast as f(x). For example, $3x = O(x^2)$. Many textbooks in this area talk a lot about Big-O, but this relationship is not particularly useful unless it's the only thing you've got. Thus, $3x = O(x^5)$, but so what? This is like saying that every man in Indianapolis is less than 13 feet tall – true but not useful. Similarly, $f = \Omega(g)$ if there exist positive constants n_0 and c such that for $x \geq n_0$, $f(x) \geq cg(x)$. This means that f(x) eventually grows at least as fast as g(x). If $f = O(g)$, then $g = \Omega(f)$.

There is also notation to tighten up these bounds. If $f = O(g)$ but we definitely know that g(x) eventually grows faster than f, then we say that $f = o(g)$ [f is **little o** of g]. The relationship between big oh and little oh is this: if $f = O(g)$, then either $f = \Theta(g)$ or $f = o(g)$. Some limit tests can be used for big oh and little oh as well. The limit tests are summarized in the table below:

$\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0$	f is o(g) [f is lower order of magnitude than g]
$\lim_{x\to\infty} \frac{f(x)}{g(x)} = p$, $0 < p < \infty$	f is Θ(g) [f and g are same order of magnitude]
$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \infty$	g is o(f) [g is lower order of magnitude than f]
$\lim_{x\to\infty} \frac{f(x)}{g(x)} = q$, $0 \leq q < \infty$	f is O(g) [f is lower or same order of magnitude as g]

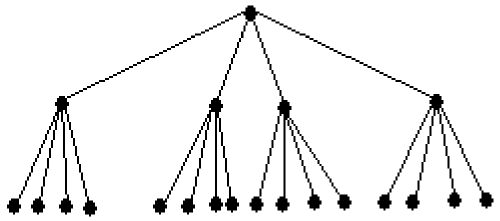
Why do we care about order of magnitude of functions? In analysis of algorithms, we identify some "unit of work" that an algorithm performs and try to find a function f(n) that expresses how many units are required on an input of size n. If we have only one algorithm to perform a given task, analysis is not very useful because we're stuck with what we've got. But if we have two algorithms to perform the task, we analyze both, and one is a lower order of magnitude than the other, we would generally prefer the algorithm of lower order of magnitude. See the following table, to see how much difference order of magnitude can make! Here there are three algorithms, of differing orders of magnitude, to do the same task. The assumption is that each step in a computation takes 0.0001 seconds. Note that exponential algorithms, $\Theta(2^n)$, are terribly inefficient.

Total Computation Time				
		Size of Input n		
Algorithm	Order	10	50	100
A	n	0.001 second	0.005 second	0.01 second
A'	n^2	0.01 second	0.25 second	1 second
A"	2^n	0.1024 second	3570 years	4×10^{16} centuries

The P = NP Question

Problems with no polynomial-time solutions (for example problems with only exponential solution algorithms) are called **intractable** because, while an algorithm to solve the problem exists, it is so inefficient as to be useless.

The kind of algorithms we usually deal with have a prescribed sequence of steps, so they are **deterministic algorithms**. The **class P** consists of all problems for which a deterministic, polynomial-time solution algorithm exists. A **nondeterministic algorithm** has a choice of actions at any step. It's perhaps best to think of such an algorithm in terms of parallel processing; at each point in which there is a choice of action, each choice is pursued in parallel. Or, one can think of a branching tree, such as



which shows 16 possible courses of action after two steps, each with 4 choices. The **class NP** consists of all problems for which a nondeterministic, polynomial-time algorithm exists; basically this says that if you pursued all possible courses of action in parallel, one of them (the "right choices") would lead to a solution in polynomial time (but if you "serialized" this process and followed each path sequentially, it could be exponential time). Another way to view NP problems is that a proposed solution can be checked in polynomial time – you follow the prescribed path until you get to a solution or you don't.

It is clear that P $\subseteq$ NP. If you can solve a problem deterministically in polynomial time, that's a trivial case of solving it nondeterministically in polynomial time (you just have one course of action to follow). The Big Question is, does P = NP? Does every problem that can be solved in polynomial time nondeterministically actually have some polynomial deterministic algorithm waiting to be discovered? The prevailing wisdom is that the answer is no, although no one has yet been able to prove this (see NEWS FLASH below…).

Supporting evidence for this conclusion has to do with NP-complete problems. A problem A is **NP complete** if it is in NP, but also has the property that an instance of any other NP problem B can be turned into (reduced to) an instance of A in polynomial time. Thus if a polynomial-time solution were ever found for A, B would also have a 2-part polynomial-time solution: reduce B to A (takes polynomial time), then apply A's polynomial time solution. Because many NP-complete problems have been identified and no polynomial time solution has ever been found for any of them, it is believed (but not proven) that P $\subset$ NP. This is the major outstanding question is computer science.

Here is a good description of the difference between *finding* a solution and *checking* a solution. The URL for this paragraph is http://www.claymath.org/millennium/P_vs_NP. The Clay Mathematics Institute has offered a $1 million (US) prize to the person who solves the P vs NP problem.

> Suppose that you are organizing housing accommodations for a group of four hundred university students. Space is limited and only one hundred of the students will receive places in the dormitory. To complicate matters, the Dean has provided you with a list of pairs of incompatible students, and requested that no pair from this list appear in your final choice. This is an example of what computer scientists call an NP-problem, since it is easy to check if a given choice of one hundred students proposed by a coworker is satisfactory (i.e., no pair taken from your coworker's list also appears on the list from the Dean's office), however the task of generating such a list from scratch seems to be so hard as to be completely impractical. Indeed, the total number of ways of choosing one hundred students from the four hundred applicants is greater than the number of atoms in the known universe! Thus no future civilization could ever hope to build a supercomputer capable of solving the problem by brute force; that is, by checking every possible combination of 100 students. However, this apparent difficulty may only reflect the lack of ingenuity of your programmer. In fact, one of the outstanding problems in computer science is determining whether questions exist whose answer can be quickly checked, but which require an impossibly long time to solve by any direct procedure. Problems like the one listed above certainly seem to be of this kind, but so far no one has managed to prove that any of them really are so hard as they appear, i.e., that there really is no feasible way to generate an answer with the help of a computer. Stephen Cook and Leonid Levin formulated the P (i.e., easy to find) versus NP (i.e., easy to check) problem independently in 1971. [1]

NEWS FLASH:
On August 8, 2010, Dr. Vinay Deolalikar, Principal Research Scientist in HP's Storage and Information Management Platforms Lab in Palo Alto, sent an e-mail to colleagues which began with

" Dear Fellow Researchers,
I am pleased to announce a proof that P is not equal to NP…"

One can imagine the storm that ensued over the claim that the attached 100+ page paper contained a solution to this very famous problem. In addition, the e-mail contained a quote from the aforementioned Stephen Cook that said: "This looks like a serious effort." Although even this quote seems controversial, as he may have said "This appears to be a relatively serious claim to have solved P vs NP", which is a bit more equivocal.

[1] I attended a technical conference on computer science theory not many years after this. In a room of fifty or so computer scientists, a lively discussion arose that went something like "Stephen Cook said xxx", "No, what he said was yyy," and continued for a few moments. Then a very quiet and unassuming man stood up and said "My name is Stephen Cook and was I actually said was zzz." I was very impressed by this.

The modern world being what it is, not only was this announcement made via an e-mail message, much of the ensuing discussion took place via blogs on the Internet. See

http://rjlipton.wordpress.com/2010/08/23/einstein-and-the-p%E2%89%A0np-question

for an overall discussion written by Richard Lipton, a famous computer scientist in his own right. You can follow the link to read his full blog.

As time went on, there appeared to be problems with the proposed proof. From http://www.nytimes.com/2010/08/17/science/17proof.html?_r=1:[2]

> "By the middle of last week, although Dr. Deolalikar had not backed away from his claim, a consensus had emerged among complexity theorists that the proposed proof had several significant shortcomings.
>
> "At this point the consensus is that there are large holes in the alleged proof — in fact, large enough that people do not consider the alleged proof to be a proof," Dr. Vardi said. "I think Deolalikar got his 15 minutes of fame, but at this point the excitement has subsided and the skepticism is turning into negative conviction."

Also see

http://www.technologyreview.com/computing/26086/?a=f

for an interesting short article called

"What Does 'P vs. NP' Mean for the Rest of Us?
Subtitle: A proposed "proof" is probably a bust--but even failed attempts can advance computer science."

So, for the moment, it seems that this Big Question is still unanswered.

[2] There is a statement in this article that says NP problems are those that are impossible for computers to solve, but for which the solutions are easily recognizable. This is misleading; NP problems are indeed solvable, it's just that their known algorithms take a huge amount of time to carry out. So they are "impossible for computers to solve" in a practical sense in that the solutions take too long.

Mergesort

The Algorithm

Mergesort cuts the list in half, recursively sorts each half, and then knits the two sorted halves back together. The recursion stops when the list has been reduced to size 1. The body of the recursive mergesort looks like this:

```
{
  if (low < high)
 { // Otherwise, no sorting is needed.
      mid = (low + high) / 2;
      mergesort(low, mid);
      mergesort(mid + 1, high)
      merge(low, high);
  }
}
```

This follows the general divide-and-conquer form. Note that Mergesort is somewhat misnamed, since it operates as "sort-merge", not "merge-sort." The merging comes after the sublists have been sorted.

Click here for mergesort animation

By the time you get to the merge function, you have an array A with two sorted halves A1 and A2. You want to build a merged, sorted array out of A1 and A2 by comparing the next two elements of each half and putting the smaller one into its proper position in the growing final array. The only problem is, you can't just put this element back into A or you've potentially written over data you need later. The simplest way to avoid this problem is to merge A1 and A2 into a new array B, then copy the results back into A. (You actually saw this process in the animation.) This makes mergesort a space-inefficient algorithm, doubling the array size required.

There is some good news, however; in the merge process, once one of the halves runs out of elements, the entire remainder of the other half can be copied directly into B with no further comparisons required.

Analysis

Comparisons

We want an expression for the number of comparisons required for mergesort to sort an n-element array. We can represent this symbolically by C(n). Comparisons are done entirely in the merge function. In the final merge, each comparison results in copying one value from one of the two sorted halves A1 and A2 into the new sorted array B being created. The number of comparisons is therefore bounded above by the sum of the lengths of the two halves, which is n. (The number of comparisons is actually less then this because once one of the halves runs out of elements, no more comparisons are required.) As an upper bound, then, we can write the number of comparisons by the following recurrence relation:

```
C(1) = 0                    //no work to sort a 1 element array
C(n) = 2*C(n/2) + n        //the number of compares to sort an n-element array is the n [at most]
compares
                             //needed for the final merge, plus 2 times however many compares it
takes to sort each half
```

For simplicity, assume that n is a power of 2. The above is then a divide-and-conquer recurrence relation of the form

$$S(n) = cS(n/2) + g(n)$$

which has a solution of the form

$$S(n) = c^{\log n} S(1) + \sum_{i=1}^{\log n} c^{(\log n)-i} g(2^i)$$

where the log n is log to the base 2 (see Mathematics Background course content page), usually written lg n. In our case c = 2 and g(n) is n. So the solution is

$$\mathrm{C(n)} = 2^{\lg n}\mathrm{C}(1) + \sum_{i=1}^{\lg n} 2^{\lg n-i}2^i = 0 + \sum_{i=1}^{\lg n} 2^{\lg n-i}2^i = \sum_{i=1}^{\lg n} 2^{\lg n-i}2^i = \sum_{i=1}^{\lg n} 2^{\lg n} = \sum_{i=1}^{\lg n} n = \mathrm{n\ lg\ n}$$

We said earlier that in the worst case, any sorting algorithm that acts by comparison of key values would have to do at least é lg n!ù comparisons. We also noted that lg n! is close to n lg n, so that

é lg n!ù ~ é n lg nù = n lg n for n = 2^k (which makes k = lg n)

So it appears that mergesort, in the worst case, is nearly optimal. The "nearly" comes about because lg n! is actually a little less than n lg n.

But we can improve on our worst case estimate for mergesort. No more comparisons are done after one of the two sublists being merged becomes empty, and one list is always empty just before the last element is inserted. Therefore every time the merge function is called, we can reduce the count on the number of comparisons by 1. This gives the recurrence relation

C(1) = 0
C(n) = 2*C(n/2) +(n - 1)

The solution here is

$$\mathrm{C(n)} = 2^{\lg n}\mathrm{C}(1) + \sum_{i=1}^{\lg n} 2^{\lg n-i}(2^i - 1) = 0 + \sum_{i=1}^{\lg n} 2^{\lg n-i}2^i - \sum_{i=1}^{\lg n} 2^{\lg n-i} = \sum_{i=1}^{\lg n} 2^{\lg n} - \sum_{i=1}^{\lg n} 2^{\lg n}2^{-i}$$

$$= \mathrm{n\ lg\ n} - \sum_{i=1}^{\lg n} n2^{-i} = \mathrm{n\ lg\ n} - \mathrm{n}\sum_{i=1}^{\lg n} 2^{-i}$$

A proof by induction shows that

$$1 + \frac{1}{2} + \frac{1}{2^2} + \cdots \frac{1}{2^k} = 2 - \frac{1}{2^k}$$

so that

$$\frac{1}{2} + \frac{1}{2^2} + \cdots \frac{1}{2^k} = 1 - \frac{1}{2^k} = 1 - \frac{1}{n}$$

and

$$n\left(\frac{1}{2} + \frac{1}{2^2} + \cdots \frac{1}{2^k}\right) = n(1 - \frac{1}{n}) = n - 1$$

Therefore we can reduce our previous upper bound by this number of comparisons, giving

n lg n - (n - 1) = n lg n - n + 1

which is even closer to the lower bound. Again, the worst case of mergesort is very nearly optimal.

The best case for mergesort occurs when one of the two sorted sublists runs out really early, for example, if everything in one list precedes everything in the other list. This would occur if the original list is already sorted (or reverse sorted). Then the number of comparisons is roughly the length of just the one sublist instead of the sum of the two sublists. This gives roughly half of what we had before, namely about

$$\frac{n}{2}\lg n$$

Both the worst case and the best case are Q(n lg n), so the average case has this same order of magnitude.

Assignments

Assignments also occur in the merge function by copying the n elements in arrays A1 and A2 (in sorted order) into array B, then copying B back into A. Representing the number of assignments to sort an n-element array by A(n) results in the following recurrence relation:

A(1) = 0 //no work to sort a 1 element array
A(n) = 2*A(n/2) + 2n //the number of assignments to sort an n-element array is the 2n assignments
//needed for the final merge, plus 2 times however many assignments it takes to sort each half

which has the solution 2n lg n. There is no best case or worst case here because these assignments occur under all circumstances, even if the original list is already sorted.

Summary: Mergesort is clearly superior in the average case for the number of comparisons to anything else we've seen (why not, it's nearly optimal), and is also low in the number of assignments. So why isn't mergesort the overall "best" algorithm? Because the array implementation is space inefficient. The alternative is to do a linked list implementation. The number of comparisons stays exactly the same; merging the two sorted halves becomes a matter of adjusting pointers, generally a much less expensive operation than copying records. But one could argue that the switch to a linked list implementation is more complex than the simple array structures we've used for all our other sorts.

Sorting Summary

Sorting Summary			
Algorithm	Worst	Average	Best
Insertion	???	random list	already sorted
compares	???	$0.25n^2$ + Q (n)	n - 1
assigns	???	$0.25n^2$ + Q (n)	0
Selection		any list	

compares	same as ->	$0.5n^2$ + Q (n)	<- same as
assigns	same as ->	3n + Q (1)	<- same as
Bubble	reverse sorted	random list	already sorted
compares	$0.5n^2$ + Q (n)	$0.5n^2$ + Q (n)	$0.5n^2$ + Q (n)
assigns	$1.5n^2$ + Q (n)	$0.75n^2$ + Q (n)	0
Smart Bubble	reverse sorted	random list	already sorted
compares	$0.5n^2$ + Q (n)	$0.5n^2$ + Q (n)	n - 1
assigns	$1.5n^2$ + Q (n)	$0.75n^2$ + Q (n)	0
Shell sort	reverse sorted	random list	already sorted
compares	---	Q ($n^{1.25}$) - experimental- depends on increment used	---
assigns	---	---	0
Mergesort	interleaved sublists, i.e., 1, 3, …2k –1, 2, 4, …2k	random list	already sorted or reverse sorted
compares	Q (n lg n)	Q (n lg n)	0.5n lg n
assigns	2n lg n	2n lg n	2n lg n

Minimal Spanning Tree

A **spanning tree** for a graph is a tree whose set of nodes coincides with the set of nodes for the graph, and whose arcs are (some of) the arcs of the graph. A *minimal spanning tree* (one with minimum weight) would represent the minimum connectivity in a network, or the minimum set of routes for a hub-spoke system such as FedEx, which flies everything to Memphis and then redistributes it, or American Airlines, which flies everyone to Dallas.

Recall that depth-first search of a graph produces a spanning tree from the forward edges used in the traversal. But because that algorithm pays no attention to the weight of any edge, it's not necessarily a minimal spanning tree. As with the shortest-path problem, there are many algorithms for finding a minimal spanning tree. *Prim's algorithm* is very similar to Dijkstra's shortest path algorithm, so quite easy to understand.

There is a set S that initially contains one arbitrary node. There is a distance array. For a node v not in S, its distance is the minimum distance between v and **any** node in S. (In the shortest path algorithm, the distance of v is the minimum path length from the source vertex using only the nodes in S.) The next node to add to S is the one with minimum distance (with tie-breaking rules). The arc with this minimal distance becomes part of the minimal spanning tree. When a node is added to S, all other distances for nodes not yet in S must be readjusted. The algorithm terminates when all nodes are in S. Like Dijkstra's algorithm, this is also an Q (n^2) algorithm. It is also a greedy algorithm, looking at each step for the shortest arc to add to the partial spanning tree we have so far, and this "local" strategy produces an overall minimal spanning tree.

Collision Resolution - Open Addressing

Collision occurs in a hash table when you want to insert an item. You apply the hash function to the key value you want to insert, go to the array position given by the resulting index, and find another element already there.

The "open addressing" solution to collision resolution is simply to make the hash table large. Then when a collision occurs, you **probe** the hash table looking for an empty slot. Of course you have to do the probing in some organized fashion that is repeatable when you later want to search for this item.

Collision also occurs when you search for a key value. Again, you apply the hash function to the key value and look in the indicated array position. If another key element is found there, you have a collision; if your target key is in the list, then this same collision occurred when your target key was inserted, and you have to follow the same probing scheme to find where the target ended up. The general approach to searching a hash table is thus:

```
Apply the hash function
Until target is found (successful search) or empty array cell is found
(unsuccessful search) or overflow occurs (error)
     Apply the probing algorithm
```

Overflow occurs when you have searched all the slots your collision algorithm allows and they are all full.

The most frequently-used probing schemes are *linear probing*, *quadratic probing*, and *random probing*.

1. **Linear probing** - just bump up the array index one step at a time looking for the next open spot. When you get to the end of the array, wrap back to the beginning and keep looking.

   Suppose the hash function, applied to the target key K being searched for or to the key about to be inserted, results in the array index h(K). Then successive probes are made at locations

   h(K) + 1
   h(K) + 2
   h(K) + 3
   etc. (modulo the hash size [array size] for the wrap-around)

   Here the array index to search is *incremented* (changed) by a constant value of 1.

   Advantage: Because the increment is a constant value, it's simple to compute the next index.

   Disadvantage: as the hash table starts to fill, you begin to get long contiguous sections of the array that are occupied (a **clustering** effect). Within a cluster, probing for the next empty spot degenerates into sequential search.

   Overflow for linear probing occurs only when the hash table is completely full, i.e., when you've marched all the way through the whole array and are back to where you started. But the more full the table, the more likely clustering occurs, so you still want the hash size to be comfortably greater than the maximum number of records you expect to store.

2. **Quadratic probing** - here the successive probes spread farther and farther apart. Suppose the hash function, applied to the target key being searched for or to the key about to be inserted, results in the array index h(K). Then successive probes are made at locations

   $h(K) + 1^2 = h(K) + 1$
   $h(K) + 2^2 = h(K) + 4$
   $h(K) + 3^2 = h(K) + 9$

etc. (modulo the hash size for the wrap-around)

Advantage: Because the locations in the array get successively farther apart, the clustering effect is largely eliminated.

Disadvantages:

1. The index for the next probe is harder to compute than in linear probing. But it need not involve squaring. The successive locations are

probe #	0	1	2	3		i
index #	h(K)	h(K) + 1	h(K) + 4	h(K) + 9	...	h(K) + i^2

which can be written as

probe #	0	1	2	3		i
index #	h(K)	h(K) + 1	(h(K) + 1) + 3	((h(K) + 1) + 3) + 5	...	previous index + (2i - 1)

In other words, just keep a count *i* of which probe you are on and add 2*i* - 1 to the location of the previous probe to get the location of this probe.

2. Assuming the hash size is a prime number, overflow occurs when the array is only about half full. (If the hash size is not prime, overflow can occur even sooner.)

Proof:

Suppose two probe positions are identical: i < j and

h(K) + i^2 = h(K) + j^2 mod hash size

so that the two probe positions agree. When is the first time this can happen?

i^2 - j^2 = 0 mod hash size
(i + j)(i - j) = 0 mod hash size
i + j = 0 or i - j = 0 since hash size is a prime

For i - j = 0, j = i + hash size. This says after hash size probes, you end up where you started. But if i + j = 0, then i + j = hashsize; here you only get halfway through the possible values when you begin to repeat.

Example: suppose hash size = 7, h(K) = 5. Then

h(K) + 0 = 5
h(K) + 1 = 6
h(K) + 4 = 2

h(K) + 9 = 0
h(K) + 16 = 0
h(K) + 25 = 2
h(K) + 36 = 6
h(K) + 49 = 5

For i = 0, j = 7 - after 7 probes, I am back at back at cell 5, but where was I in the meantime? I only do 4 probes before the locations start to repeat at
(i = 3, j = 4).

Therefore the hash table should be kept less than half full so that you can expect to find an empty slot when you insert a new data item.

3. Random probing.

Suppose a collision occurs at index h(K). Generate a pseudorandom number i that is less than hash size, and perform the next probe at location
h(K) + i (mod hash size).

*Advantage*s: none I can think of

Disadvantages:

1. You need a random number generator
2. It's hard to determine the overflow condition

Notice that you have to use a pseudorandom number generator, not a truly random number generator, so the actions are repeatable when you go to search for a value.

4. Rehashing

Still another form of probing is to rehash the key value using a second hash function. If h_1(K) results in a collision, then probe using

h_1(K) + h_2(K), h_1(K) + 2*h_2(K), h_1(K) + 3*h_2(K), ... etc.

where h_2 is a second hash function.

Advantage: This may spread out the locations probed so as to reduce further collisions.

Disadvantages:

1. It's hard enough to think up one good hash function, much less two.
2. Rehashing could be more time-consuming than simple linear or quadratic probing.

Deletions

In open addressing, we know how to insert/search (virtually identical operations). What about deleting an element?

Here's the problem. Element A is inserted into the hash table. Later you insert element B, which hashes to the same index as A; you follow your probing algorithm and insert B somewhere else. Now you delete A from the array, making the array cell empty. Later you search for B. You apply the hashing function and detect an empty cell. What's your conclusion? That B is not in the array, which is incorrect.

Here's the solution: perform a **logical delete**. Put a special marker in A's location to show that A is no longer there, but something once was. Then when B hashes to this location, it will not terminate the search. Note, however, that logically deleted items still take up space in the hash table, leaving fewer locations to insert "real" data. If you allow new data to be inserted into logically deleted slots, then you lose the ability to prevent duplicate keys; you could insert another copy of B in the logically-deleted slot, and you would then have duplicate entries in the hash table, whereas in the rest of the code, duplication results in an error message.

Priority Queue

In an ordinary queue structure, new items are added to the back and removed from the front. In a **priority queue**, items are assigned a priority value, and are removed in order of their priority. Priority queues have many applications. For example, a multitasking operating system may choose to execute jobs with a high priority first. A network switching node may choose to forward message packets with a high priority first. An event-driven program may assign some events a higher priority than others and respond to them first.

Several choices exist for implementing priority queues. Each must carry out the two fundamental tasks of inserting a new item into the queue, and removing the top priority item from the queue.

- Maintain the queue as a linked list sorted in priority order. Keep the top priority item at the head of the list.

Remove is Q(1) because the next item to be removed is at the head of the list
Insert is Q(n). Use a short sequential search to find where to insert a new item. This is Q(n) in the worst case and Q(n/2) on the average. Note that a binary search would not improve things because a binary search works well only in an array implementation, and an array implementation would require shuffling to insert the new item in place.

- Maintain the queue as an unsorted linked list.

Insert is Q(1) because you can just insert the new item at the head of the list.
Remove is Q(n) in both the worst case and the average case because you have to search the entire list to find the highest priority item. Again, removing an item from an array would require shuffling, hence the linked list.

So both of these choices require (one way or the other) worst case work of Q(n) comparisons.

A better solution is to use a heap sorted by priority.

If we maintain an array as a heap, how can it serve as a priority queue? It's easy to see that the maximum array element is at the root of the tree (and it's the first element in the array), so to dequeue the highest priority element we grab the first array element. But this leaves a "hole" at the root of the binary tree that has to be filled in such a way as to maintain the heap property. But - we've already seen that this is exactly what the *moveDown* function does.

Dequeue (Remove)

The pseudocode to dequeue from a (nonempty) heap is

```
extract the element from the root (first array element - highest priority)
remove the last leaf (last array element), store it in temp
use moveDown to reorganize array as a heap with temp as the displaced value
return the highest priority element
```

At the end of this process, the array is once again a heap. Here the *moveDown* function is always invoked with array index arguments of 0 and count - 1, in other words, the *moveDown* function will be acting on the whole array (the whole tree - the root of the tree has been removed).

Also, when we enqueue an element, it must be inserted into the array (tree) in such a way as to maintain the heap property.

Enqueue (Insert)

To enqueue an element e into a heap, add it to the end of the array (assuming there is enough space left in the array). Now e may have to be moved up in the tree until it reaches a spot where its parent node has a value greater than or equal to e. Note that if e is not the only child, the parent node is already >= the other child. Here is an algorithm for this:

```
put e at the end of heap;
while e is not in the root and e > parent(e)
   swap e with its parent;
```

At the end of this process, e is in its proper spot and the array is once again a heap. (See "Enqueue into heap-priority queue.doc"in Code Archives for an example.).

This process reorganizes the array into a heap by working e back up from the leaf level toward the top of the tree (back to front of the array). Note that the reorganization to obtain a heap after adding e to the end could also be done using *moveDown* (throw the root to the back and insert e). This would be somewhat inefficient because your array is already a heap except for the new element, and you just put the maximum value at the end where it has to work its way clear back to the top, whereas e might find its place from bottom to top earlier.

Implementation

The following shows all of the class definition for a template priority queue as a heap.

```
#include "utility.h"

#ifndef PQ_H
#define PQ_H

//header file PQ.H for class pq.
//An array-based implementation of a priority queue,
//maintained as a heap.

//Because this is a template class,
//both interface and implementation
//are in the header file

const int max_list = 100;

template <class Entry>
class pq
{
//methods of the pq class
public:

pq();
//constructor
//postcondition: empty list has been created

int size() const;
//postcondition: returns number of entries in list

void enqueue(const Entry &current);
//Pre: The entries of the array between indices 0 and
//count - 1, inclusive, form a heap. Current is to be
//added to the heap. Queue is not full.
//Post: Current is in the array, count has been incremented,
//and the entries between indices 0 and count - 1
//(new value of count) form a heap. Heap is reorganized
//from back to front (bottom of tree to top)
//Uses swap

void write_out() const;
//write out contents of heap

Entry dequeue();
//Pre: the array is a heap, highest priority message
//at the front. Queue is not empty.
//Post: returns first (highest priority) element; this
//element has been removed from array and array reorganized
//as a heap.
//Uses moveDown

bool empty();
//returns true if queue is empty

bool full();
//returns true if queue is full

private:
void swap (Entry &x, Entry &y);
//swap the two values

void moveDown(const Entry &current, int low, int high);
//Pre: The entries of the array between indices low + 1 and high,
//inclusive, form a heap.
//Post: The entry current has been inserted into the array
//and the entries rearranged so that the entries between
//indices low and high, inclusive, form a heap.

//Member variables

int count;
Entry queue[max_list]; //heap array
};
#endif; //pq class
```

In the above class, we are comparing elements of the array, so we must have a comparison operator for type Entry that compares on priority.

Analysis

Remove (dequeue) would involve no work to find what to remove - the highest priority item is at the root (the first element in the array). The heap is rebuilt by invoking moveDown, which does

Q(lg((high + 1)/(low + 1)))

work. So, taking

low = 0 and high = n - 1

this would give Q(lg n) work in the worst case.

Insert (enqueue) puts a new element at the bottom of the tree and it works its way up to its proper position. In the worst case, this would move the new element clear up to the root position. The height of the tree, as before, is approximately lg n, and there is one compare (node vs. parent) and 3 assignments (swap node and parent) at each level. The total work is therefore Q(lg n).

Conclusion

Maintaining a priority queue as a heap would involve worst case Q(lg n) work for both operations. This would therefore be the method of choice.

Quicksort

The Algorithm

Quicksort picks a **pivot element** p and moves all elements < p to before p in the array, and all elements > p to after p in the array. Then it partitions the list at the pivot and sorts each half (recursively). When it is time to combine the sorted pieces, everything is in order and there is nothing to do.

The body of the recursive quicksort function looks something like this:

```
{
  if (low < high)
 { // Otherwise, no sorting is needed.
     do a partition around the pivot
     recursive_quick_sort(low, pivot_position - 1);
     recursive_quick_sort(pivot_position + 1, high);
  }
}
```

which follows the general divide-and-conquer form, except that the "combine" phase has been omitted.. The recursive function is passed the indices that mark the beginning and end of the section of the array to be sorted (*low* and *high*), and when low becomes ^{3} high, nothing happens. In other words, one-element lists or empty lists require no processing.

All of the work in quicksort is done in the partition process. In the below animation, the partitioning process is done within a *Partition* function. *Partition* works as follows: Pick the middle element of the list to be the pivot element, and swap it into position *low*. Move an index *i* through the rest of the list, from (*low* + 1) to *high*. *Last_small* marks the end of the stuff that is < p. Examine element *i*: if it is ^{3} p, then it's in the correct part of the array, so just leave it where it is and advance *i*. If it's < p, advance *last_small* (which may temporarily contain an element ^{3} p, and swap elements at *i* and *last_small.* Finally, swap *last_small* and *low* elements to put the pivot in its correct position in the array. Here is the body of the function code (taken from Kruse and Ryba, Data Structures and Program Design in C++, 1999, Prentice-Hall). Again, *low* and *high* are parameters for the beginning and end of the section of the array to be sorted:

```
{
   List_entry pivot;
   int i, // used to scan through the list
   last_small; // position of the last key less than pivot
   swap(low, (low + high) / 2);
   pivot = entry[low]; // First entry is now pivot.
   last_small = low;

   for (i = low + 1; i <= high; i++)
      if (entry[i] < pivot)
     {
         last_small = last_small + 1;
         swap(last_small, i); // Move large entry to right and small to left.
     }
   swap(low, last_small); // Put the pivot into its proper position.
   return last_small;
}
```

Click here for a Quicksort animation

The choice of the middle element as the pivot is arbitrary. We could choose the first element, BUT if the list is already sorted or sorted in reverse order, then it will partition into an empty list and a sublist only 1 item shorter than before. This loses the power of the divide and conquer idea, and is a worst-case scenario. Of course, given the right data, this could also happen when the middle element is chosen as the pivot, but this would be a more pathological scenario. Sometimes the median of the first, middle and last array elements is used, but this also does not guarantee the more or less even division we would like.

Analysis

Comparisons:

We want an expression for the number of comparisons required for quicksort to sort an n-element list. We can represent this symbolically by C(n). As we do the partition of the list, every element beyond position *low* gets compared to the pivot element. Initially, this will mean n - 1 comparisons, as we partition the original list. Now suppose the two sublists have lengths r and (n - 1) - r [the pivot element does not belong to either sublist]. Because quicksort is recursive, these two sublists must eventually be sorted. Using our notation, we can say that the number of comparisons to sort the r-length sublist is C(r), even though we don't know what that really means, and the number of comparisons to sort the other sublist is C(n - 1 - r). Therefore, the total number of comparisons required to sort the original list is

$$C(n) = (n - 1) + C(r) + C(n - 1 - r)$$

and of course C(1) = 0.

Worst case: In the worst case, the choice of pivot is such that one of the sublists is empty and the other sublist has size n - 1. Then the above expression becomes

$$C(n) = n - 1 + C(0) + C(n - 1) \text{ (zero comparisons to sort an empty list)}$$

$$= (n - 1) + C(n - 1).$$

This is a linear, first-order recurrence relation with constant coefficients. The solution can be found by the techniques of discrete mathematics, and it is

$$\sum_{i=1}^{n}(i-1) = 0 + 1 + \ldots + (n-1) = \frac{1}{2}(n-1)n = 0.5n^2 - 0.5n = 0.5n^2 + \Theta(n)$$

This is equivalent to the work done by Selection sort (which you will recall was bad in the number of comparisons).

Assignments:

Instead of counting assignments, let's count swaps. Let S(n) represent the number of swaps quicksort will do on an n-element list. These swaps occur in the partitioning process.

Worst case: In the worst case, every element encountered is less than the pivot and must be swapped. This happens if the pivot is the maximum element in the list. [This would happen in an implementation that used the pivot element as the first element in the list if the list is in reverse sorted order.] In addition to the n - 1 swaps to process the list, there is one swap to move the pivot into the first position of the list and then one final swap to move the pivot into its correct position. This worst-case scenario would also result in one of the two sublists being empty. Therefore

$$S(n) = (n - 1) + 2 + S(n - 1)$$

This is another recurrence relation. The solution is also

$$0.5n^2 + Q\ (n)$$

By the time you factor in the 3 assignments for each swap, the number of assignments is

$$1.5n^2 + Q\ (n)$$

This is worse than the average case of insertion sort, which has to do a lot of swaps, and indeed is worse than the worst case of insertion sort.

Summary: Quicksort can be (in certain cases) as bad as selection sort in comparisons and worse than insertion sort in assignments. So what's so "quick" about quicksort? Its average case behavior is good.

Average case

For the average case analysis, assume that all possible orderings of the list are equally likely.

Suppose that the pivot element, after being finally put in its correct position, ends up as the pth element in the list, with p - 1 elements to the left that have values < the pivot, and n - p element to the right that have values > the pivot element. In other words, unlike the worst-case analysis, the sublists that will be sorted on the next invocation of the recursion are both non-empty. Using symbolic notation as before, let

C(n, p) denote the number of comparisons
S(n, p) denote the number of swaps

required to sort the list where the pivot ends at position p. Ultimately we want to find C(n) and S(n), the average number of comparisons and swaps to sort the entire list where the pivot element might end up anywhere.

This time, let's deal with swaps first. This analysis is based on the *Partition* function described above. Recall that within the **for** loop of the partition function, every entry less than the pivot element precipitates a swap, so there will be p - 1 swaps for the p - 1 entries less than the pivot. In addition, there are the 2 swaps to move the pivot element to the front and then into its correct position. Therefore there are

$$(p - 1) + 2 = p + 1$$

swaps on the first pass, plus however many swaps are required to sort the two sublists. Therefore

$$S(n, p) = (p + 1) + S(p - 1) + S(n - p) \qquad (1)$$

Note that the number of swaps to sort the two sublists are represented using the average notation (one parameter instead of two) because we make no assumptions about where their pivot elements might be located.

So equation (1) represents the number of swaps for the case where the pivot element ends up at position p. To find the average work for all cases, use the familiar formula:

$$\sum_{\text{all cases}} (\text{work for this case}) * (\text{probability of this case})$$

Here the different "cases" are the different positions (1, 2, ..., p, ..., n) in the list where the pivot element could end up. Each is equally likely, so each has probability 1/n. Therefore

$$S(n) = [S(n, 1) + S(n, 2) + \ldots + S(n, n - 1) + S(n, n)] * \frac{1}{n}$$

Now substitute from Equation 1 into this equation, using various values for p of 1, 2, ..., n - 1, n.

$$S(n) = \frac{1}{n} * \begin{cases} (1 + 1 + S(0) + S(n-1) \\ + (2 + 1 + S(1) + S(n-2) \\ + \ldots \\ + n + S(n-2) + S(1) \\ + n + 1 + S(n-1) + S(0) \end{cases}$$

Combining like terms, this gives

$$\begin{aligned} S(n) &= \frac{1}{n} * [(2 + 3 + \ldots + n + 1) + 2(S(0) + S(1) + \ldots + S(n-1))] \\ &= \frac{1}{n} * \left[\frac{(n+1)(n+2)}{2} - 1 + 2(S(0) + S(1) + \ldots + S(n-1))\right] \\ &= \frac{1}{n} * \left[\frac{n^2 + 3n + 2}{2} - 1 + 2(S(0) + S(1) + \ldots + S(n-1))\right] \quad \text{now multiply out} \\ &= \frac{n}{2} + \frac{3}{2} + \frac{1}{n} - \frac{1}{n} + \frac{2}{n}(S(0) + S(1) + \ldots + S(n-1)) \\ \text{so } \; S(n) &= \frac{n}{2} + \frac{3}{2} + \frac{2}{n}(S(0) + S(1) + \ldots + S(n-1)) \qquad (2) \end{aligned}$$

Now Equation (2) is also a recurrence relation, but more complicated than we have seen before because it is not first-order, i.e., more than just the n - 1 term appears in the recurrence relation for S(n).

To solve this recurrence relation we must resort to some rewriting. Restate Equation (2) using n - 1 instead of n:

$$S(n-1) = \frac{n-1}{2} + \frac{3}{2} + \frac{2}{n-1}(S(0) + S(1) + \ldots + S(n-2)) \quad (3)$$

Now multiply Equation (2) by n and Equation (3) by n - 1 and subtract:

$$nS(n) - (n-1)S(n-1) =$$
$$\frac{n^2}{2} + \frac{3n}{2} + 2(S(0) + S(1) + \ldots + S(n-1)) - \left[\frac{(n-1)^2}{2} + \frac{3(n-1)}{2} + 2(S(0) + S(1) + \ldots + S(n-2)\right]$$

Most of the terms will cancel out:

$$nS(n) - (n-1)S(n-1) =$$
$$\frac{n^2}{2} + \frac{3n}{2} + 2(\cancel{S(0)} + \cancel{S(1)} + \cancel{\ldots} + S(n-1)) - \left[\frac{(n-1)^2}{2} + \frac{3(n-1)}{2} + 2(\cancel{S(0)} + \cancel{S(1)} + \ldots + \cancel{S(n-2)}\right]$$

so that

$$
\begin{aligned}
&nS(n) - (n-1)S(n-1) = \\
&\quad \frac{n^2}{2} + \frac{3n}{2} + 2S(n-1) - \left[\frac{(n-1)^2}{2} + \frac{3(n-1)}{2}\right] \\
&\quad = \frac{n^2}{2} + \frac{3n}{2} + 2S(n-1) - \frac{n^2-2n+1}{2} - \frac{3n-3}{2} \quad \text{cancelout term:} \\
&\quad = 2S(n-1) + n - \frac{1}{2} + \frac{3}{2}
\end{aligned}
$$

or

$$nS(n) - (n-1)S(n-1) = 2S(n-1) + n + 1$$

Now collect the S(n-1) terms

$$nS(n) = (n+1)S(n-1) + n + 1$$

and divide by n(n + 1)

$$\frac{S(n)}{n+1} = \frac{S(n-1)}{n} + \frac{1}{n}$$

This now is a first-order recurrence relation. S(n) has been expressed using only S(n - 1). However, the recurrence relation does not have constant coefficients, so we don't have a solution formula. We can do successive expansions:

$$
\begin{aligned}
\frac{S(n)}{n+1} &= \frac{S(n-1)}{n} + \frac{1}{n} \\
&= \left[\frac{S(n-2)}{n-1} + \frac{1}{n-1}\right] + \frac{1}{n} \\
&= \left[\frac{S(n-3)}{n-2} + \frac{1}{n-2}\right] + \frac{1}{n-1} + \frac{1}{n} \\
&\ldots \\
&= \frac{S(0)}{1} + \frac{1}{1} + \frac{1}{2} + \ldots + \frac{1}{n} \\
&= 1 + \frac{1}{2} + \ldots + \frac{1}{n}
\end{aligned}
$$

We already did an approximation of

$$\frac{1}{2} + \ldots + \frac{1}{n}$$

using a definite integral (see the Average BST page) . The result is approximately ln n. Therefore

S(n) = (n + 1)(ln n + 1) » n ln n +Q (n)

Converting ln n to lg n, we get

S(n) » 0.69n lg n +Q (n)

Finally, using 3 assignments per swap

assignments » 2.07n lg n +Q (n)

For comparisons, the partition function does n - 1 comparisons on an n-element list, so

C(n, p) = (n - 1) + C(p - 1) + C(n - p)

Proceeding as before, we get

C(n) » 1.39n lg n +Q (n)

(note that this is 39% more work than the number of comparisons required by the worst case of mergesort)

Sorting Summary			
Algorithm	Worst	Average	Best
Insertion	???	random list	already sorted
compares	???	$0.25n^2$ + Q (n)	n - 1
assigns	???	$0.25n^2$ + Q (n)	0
Selection		any list	
compares	same as ->	$0.5n^2$ + Q (n)	<- same as
assigns	same as ->	3n + Q (1)	<- same as
Bubble	reverse sorted	random list	already sorted
compares	$0.5n^2$ + Q (n)	$0.5n^2$ + Q (n)	$0.5n^2$ + Q (n)
assigns	$1.5n^2$ + Q (n)	$0.75n^2$ + Q (n)	0
Smart Bubble	reverse sorted	random list	already sorted
compares	$0.5n^2$ + Q (n)	$0.5n^2$ + Q (n)	n - 1
assigns	$1.5n^2$ + Q (n)	$0.75n^2$ + Q (n)	0
Shell sort	reverse sorted	random list	already sorted
compares		Q $(n^{1.25})$ -	

	---	experimental - depends on increment used	---
assigns	---	---	0
Mergesort	interleaved sublists, i.e., 1, 3, …2k –1, 2, 4, …2k	random list	already sorted or reverse sorted
compares	Q (n lg n)	Q (n lg n)	0.5n lg n
assigns	2n lg n	2n lg n	2n lg n
Quicksort	reverse sorted or already sorted with pivot = first or last element	random list	
compares	$0.5n^2$ + Q (n)	1.39 n lg n + Q (n)	
assigns	$1.5n^2$ + Q (n)	2.07 n lg n + Q (n)	

Best case

The best case for quicksort would occur if we could guarantee that each time the pivot element is put into place, it splits the list into two equal-sized sublists. This would require that the pivot be the median of the list elements. How much work is it to find the median of n elements so we could use it as the pivot element?

This turns out to be rather a lot of work. Unlike finding, say, the maximum or minimum, you can't just march through the list because there is too much to remember. Suppose we want the kth smallest element out of the n elements in the list. (For the median element, k will be én/2ù, i.e., if n = 7, then the median is the 4th smallest element - there are 3 values smaller, then this one, then 3 values larger.) Here's a rather neat divide-and-conquer approach that is very similar to the partition function. Pass parameters for the section of the array being worked on, and also pass k. On the first pass, you use the whole list. Pick a random number as a pivot element and partition the list just as is done for quicksort. Then count the number of elements in the first half to decide whether the kth smallest element occurs in the first half L_1or the second half L_2 (or is perhaps the pivot). That is, if k £ | L_1| then the kth element is in L_1 and indeed is the kth smallest element in L_1so you recursively call the function, passing $L_1$and k. If k = | L_1| + 1, then the pivot element is the median element - quit here. Otherwise, the kth smallest element occurs in L_2 and is the (k - | L_1| - 1)st smallest element in L_2 so you recursively call the function, passing L_2 and k - | L_1| - 1. When you get to a 1-element list, that element will be the median, which terminates the recursion.

But this process would have to be done for each of the recursive calls of quicksort in order to find the median for that sublist. Even if the lists split evenly in half, there are at least lg n "levels" of recursive calls, so n lg n work just to find the pivot points. What do we save by using the median value as the pivot point?

If the sublists split evenly in half, then the number of comparisons would be

$$C(n) = (n - 1) + 2C(n/2)$$

which, while not a first-order recurrence relation, can still be solved using techniques from discrete mathematics. The solution is

$$n \lg n + Q(n)$$

Compared with the average case, we have a lower coefficient - but only at the expense of all the work we had to do to find the pivots.

Conclusion: It's not worth it to try and force quicksort to optimize by always finding the median element to use as the pivot.

Radix Sort

All of the sort algorithms we have studied have made no assumptions about the data, i.e., they were general sort algorithms that apply to any data for which an ordering relation exists (obviously you can't sort without this property). In a **bucket sort**, the items are distributed into piles or "buckets" based on some knowledge about the data. The buckets are each sorted by one of our general algorithms, then the contents are recombined in such a way as to preserve the sorted order.

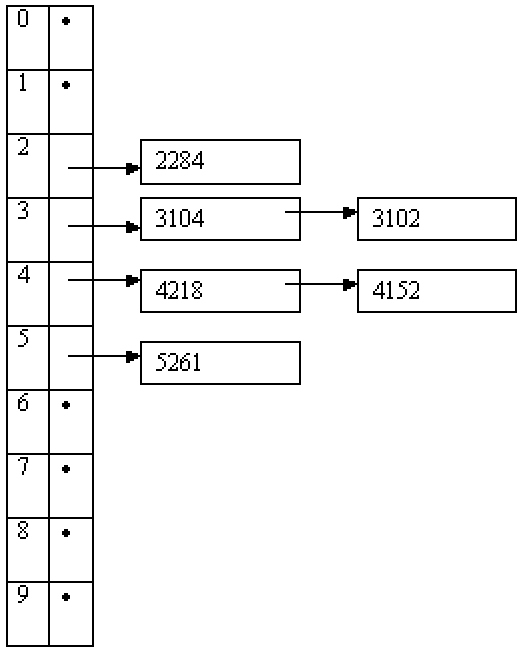
For example, if the keys are names, then the buckets could be the 26 letters of the alphabet. Distribute the keys based on the first letter, sort each bucket, recombine by concatenation. The distribution phase should involve little work, i.e., should be Q(n) - just look at each element and do a small constant amount of work to designate the bucket to which it is assigned. Similarly, recombining after sorting the buckets should involve little work if the buckets are chosen appropriately. Therefore most of the work is in sorting the buckets. We hope for an even distribution among the buckets. The worst case would have one bucket filled with the entire original list and the rest empty, which accomplishes nothing.

This sounds a bit like a divide and conquer approach where we partitioned the list into smaller sublists and then used recursion. Suppose we recursively call a bucket sort on each bucket, say by moving to the second letter in the name. We repeat this process until we end up with a base case of sorting a 1-element bucket. Problem: there is too much bookkeeping to keep track of which successive buckets an item belongs to, i.e., we must sort of manually manage the recursion stack. (A minor problem is that we may only end with small buckets, not necessarily 1-element buckets, but in this case everything in a bucket at the end is the same, so it does not need to be sorted.)

Consider the following data stored as a linked list.

$$\begin{tikzcd} 3104 \arrow[r] & 4218 \arrow[r] & 3102 \arrow[r] & 5261 \arrow[r] & 2284 \arrow[r] & 4152 \end{tikzcd}$$

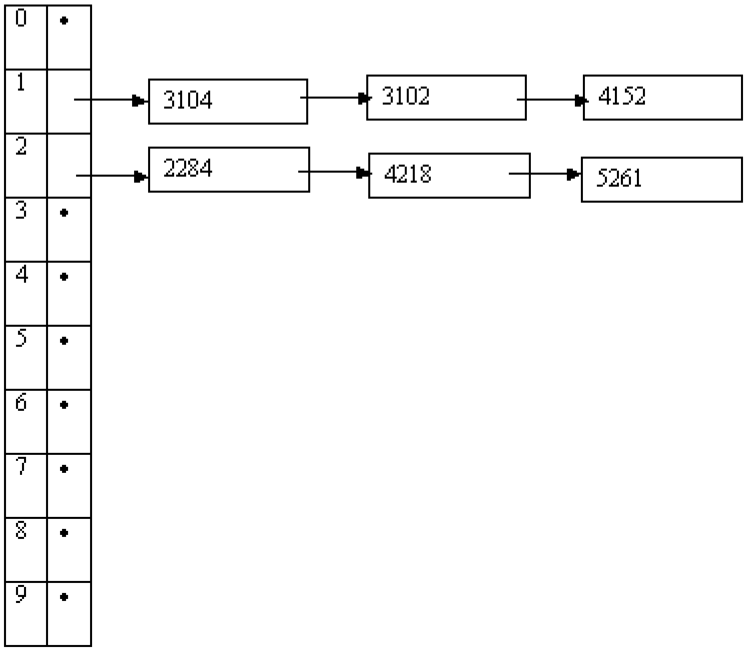
We wish to sort the linked list where the buckets are 0-9. Suppose we make each bucket a linked list also. Do a bucket sort on the leftmost (most significant) digit. Here we will add new elements to the end of the linked lists in order to keep them in their relative order. Ordinarily, adding to the end of the list is a lot of work as compared to adding at the head of the list, so we will keep an (invisible) pointer to the back of each list. (Hence the list can be thought of as a queue.)



Now we must distribute each bucket into buckets based on the digit second from left, involving at this point 10 * 10 = 100 linked lists. In order to avoid all this bookkeeping, suppose we recombine first by concatenating the bucket lists back together. The new single linked list is then

$$\begin{tikzcd}2284 \arrow[r] & 3104 \arrow[r] & 3102 \arrow[r] & 4218 \arrow[r] & 4152 \arrow[r] & 5261\end{tikzcd}$$

which is sorted with respect to the first digit. Now we again distribute into buckets based on the second digit.



and then recombine into a single linked list:

$$\begin{tikzcd}3104 \arrow[r] & 3102 \arrow[r] & 4152 \arrow[r] & 2284 \arrow[r] & 4218 \arrow[r] & 5261\end{tikzcd}$$

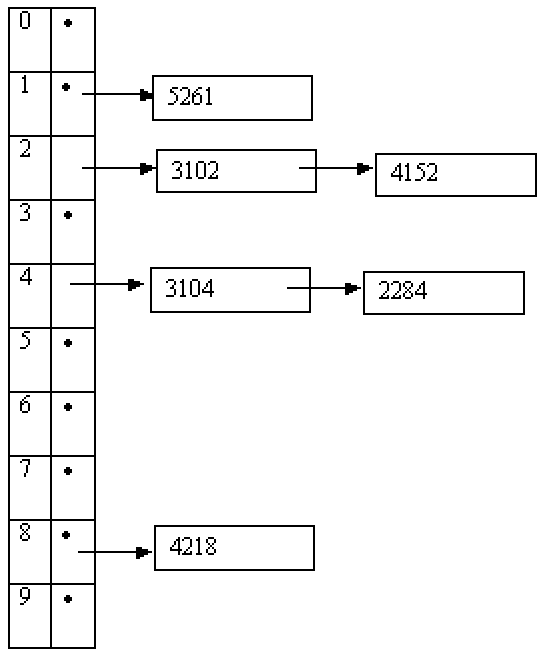
What is the problem? While our list is now correctly sorted on the second digit, we have lost all of the sorting on the first digit that we did on the previous pass. If we do four passes, one for each digit, then after the fourth pass, we can only guarantee that the numbers are sorted with respect to the fourth digit.

We can avoid this problem by sorting on the least significant digits first (not very intuitive). If two numbers differ in position i (from the right) as the leftmost digit in which they differ, i.e., they have the same digits left of i, they will be put in the correct sorted order at pass i and will remain in correct relative sorted order. This is called **radix sort.**

Example: Do a radix sort on the same linked list.

$$\begin{tikzcd}3104 \arrow[r] & 4218 \arrow[r] & 3102 \arrow[r] & 5261 \arrow[r] & 2284 \arrow[r] & 4152\end{tikzcd}$$

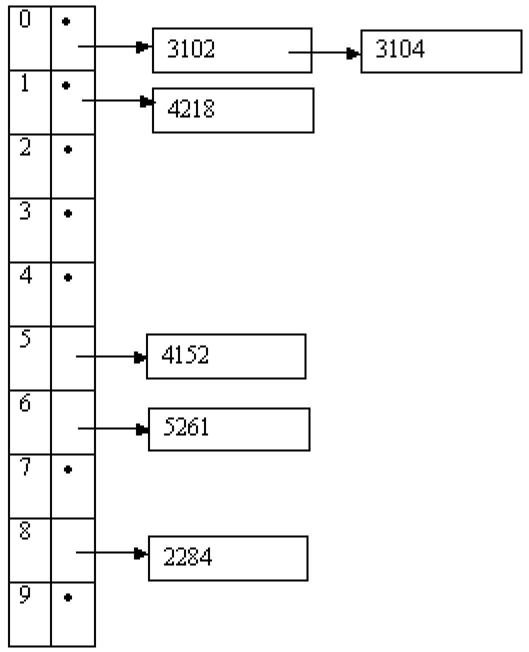
Pass One: Distribute into buckets based on rightmost digit.



and recombine:

$$\begin{tikzcd}5261 \arrow[r] & 3102 \arrow[r] & 4152 \arrow[r] & 3104 \arrow[r] & 2284 \arrow[r] & 4218\end{tikzcd}$$

Pass Two: Distribute into buckets based on second digit from right.

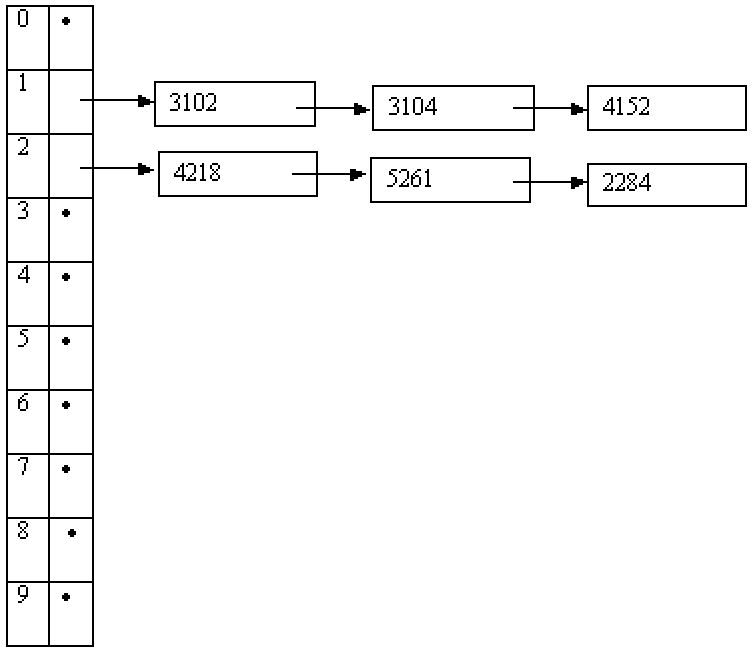


and recombine:

$$\begin{tikzcd}3102 \arrow[r] & 3104 \arrow[r] & 4218 \arrow[r] & 4152 \arrow[r] & 5261 \arrow[r] & 2284\end{tikzcd}$$

Note that at this point all numbers are sorted properly with respect to the last two digits, so at pass two, we did not lose the effect of pass one. Also, 3102 and 3104 (with the same first two digits) are now in the correct order relative to each other, and will so remain on succeeding passes.

Pass Three: Distribute into buckets based on third digit from right.

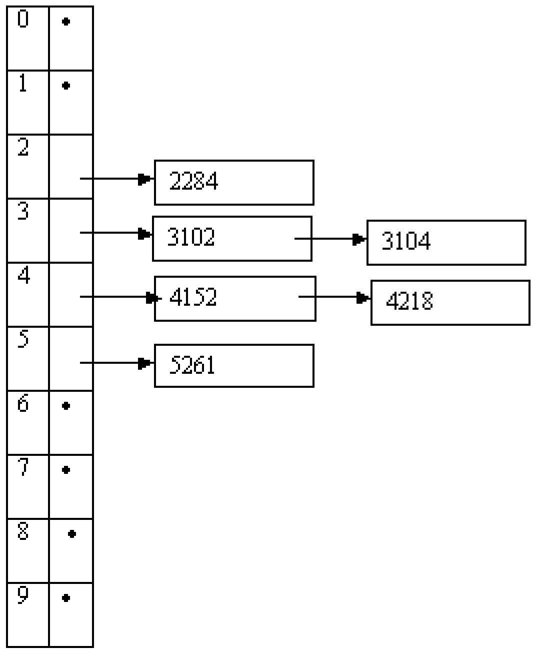


and recombine:

3102 → 3104 → 4152 → 4218 → 5261 → 2284

All numbers are sorted properly with respect to the last three digits.

Pass four: Distribute into buckets based on leftmost digit.



and recombine to get the list in final, sorted order:

2284 → 3102 → 3104 → 4152 → 4218 → 5261

Analysis: In radix sort, unlike all previous sorts, elements to be sorted are not compared against each other, so the "work unit" here is different.   The distribution phase of radix sort does Q(n) work, consisting of just examining each key and doing some constant amount of work with pointers to insert it at the end of the correct bucket list. The recombine phase does some constant amount of pointer manipulation for each pass to link the bucket lists back together. So each pass requires Q(n) work, and there are k passes (a fixed number, for example, the number of digits). The work is thus Q(n), giving a linear algorithm!!!!! This by far beats anything else we have seen. But before we get too excited, remember that this is not a general sorting algorithm, but instead makes use of special knowledge about the key values.

Recursive Sorts

The next two sorting algorithms, Mergesort and Quicksort, use the *divide- and-conquer* approach - break the task down into smaller copies of the same task until the tasks are so small that we can easily do them. Then reassemble the subtasks to complete the original task. This suggests recursion - a function calling itself with new (smaller) parameters. Upon each call, the variables associated with the current invocation are stored on a stack to be retrieved and used when backing out of the recursion. The final invocation occurs when the parameters have reduced the problem to the base case, which is trivial to solve. Recall that a recursive algorithm must have a base case as its stopping point in order to avoid **infinite descent** - recursion that keeps going forever, or rather until the system runs out of stack space and crashes with some error to that effect.

The general divide-and-conquer approach to sorting is:

```
void Sortable_list::sort()
{
   if the list has length greater than 1 {
      partition the list into lowlist, highlist;
      lowlist.sort();
      highlist.sort();
      combine(lowlist, highlist);
   }
}
```

Note that the base case occurs when the list has length = 1. A 1-element list is trivially sorted, so there is nothing to do.

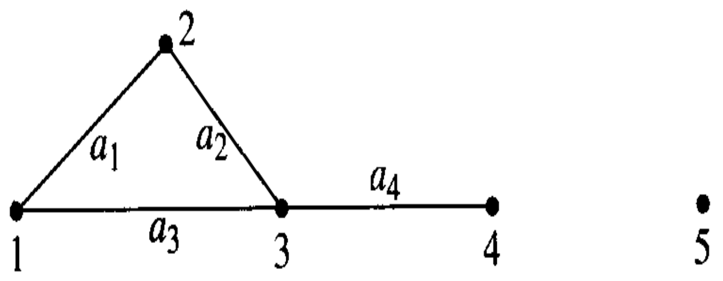
Mergesort does nothing sophisticated in partitioning the list, it just cuts it in half. Then the *combine* step is where the work is done. Quicksort works hard before partitioning the list to put all elements < some **pivot** ahead of the pivot, and all elements > pivot value behind the pivot. Then after the two sublists are sorted, combining is just concatenation.

Terminology and Representation

The following table is a quick review of graph terminology, which should be familiar from discrete structures:

Term	Meaning
Graph	Set of **vertices** (**nodes**) and **edges** (**arcs**) between vertices
Undirected graph (graph)	Arcs are represented by a set of endpoints {v, w}
Directed graph (digraph)	Arcs are represented by an ordered pair of endpoints (v, w)
Adjacent vertices v, w in a graph	An arc exists between v and w
Path in a graph	Sequence of distinct adjacent vertices and their corresponding edges
Cycle in a graph	Path that begins and ends at the same vertex but has no other repeating vertices.
Connected graph	Path exists from any vertex to any other vertex
Complete graph	Any two distinct nodes are adjacent
Degree of a vertex	Number of arc ends at that vertex
Weighted graph	A numerical value or "weight" is associated with each arc; it could represent $, distance, traffic congestion, etc.
Tree (free tree, nonrooted tree)	Acyclic, connected graph
Here a graph cannot have parallel arcs (two arcs with the same endpoints) or loops (an arc from a vertex to itself) in graphs. A *multigraph* allows parallel arcs and loops. Some authors define graphs to include parallel arcs and loops, and use the term *simple graph* if parallel arcs and loops are not allowed.	

Below is a graph with 5 vertices and 4 edges. The path 1-a1-2-a2-3-a3-1 is a cycle. The degree of node 3 is 3.



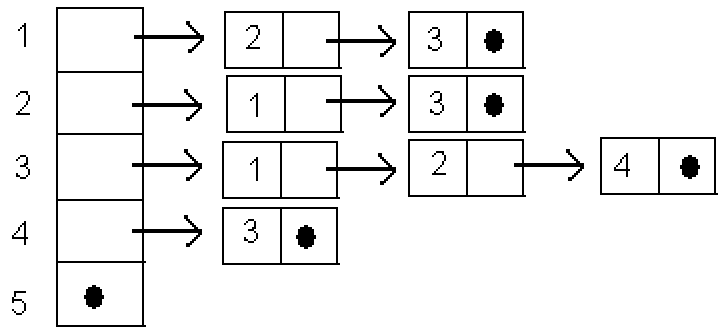
Although we think of a graph in some visual way, as in the above picture, we don't want to store it in the computer as an image because then we can't manipulate it. Instead we store the important elements that are part of the definition of a graph - what the nodes are and which nodes have connecting arcs. From this information a visual representation can be reconstructed if desired. The two common computer implementations of a graph are

1. **Adjacency matrix**. For a graph with n nodes, the adjacency matrix is an n x n matrix where entry a_{ij} = 1 if nodes i and j are adjacent, and a_{ij} = 0 if nodes i and j are not adjacent. The result is a Boolean matrix (entries of 0's and 1's). In a directed graph, a_{ij} = 1 if there is an arc from node i to node j. The adjacency matrix for a graph is symmetric, but that's not necessarily true for a directed graph. The adjacency matrix for the above graph is

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Note that the matrix is symmetric because this is a (simple) graph.

2. **Adjacency lists**. Most commonly implemented as an array of linked lists, where each array index represents a vertex, and the corresponding linked list contains the vertices adjacent to this vertex. Below is the adjacency list for our sample graph. Note that the adjacency list for node 3 contains 1, 2, and 4. These are all the nodes adjacent to node 3; don't be fooled into thinking the pointer from 2 to 4 means that 2 is adjacent to 4, which is not true.



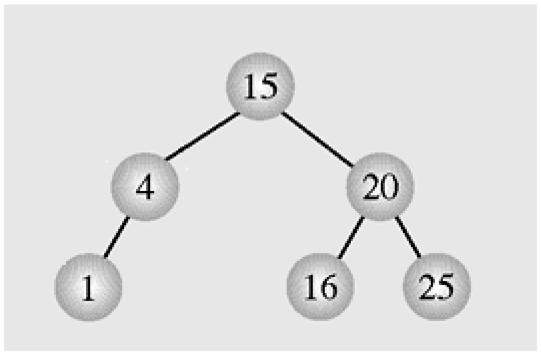
The advantages of each implementation are outlined below.

Implementation	Advantage
Adjacency matrix	If the graph has many arcs, avoids storage space for lots of pointers Allows direct answer to the question "is node i adjacent to node j"
Adjacency lists	Can save storage, even allowing for having to store pointers, over a sparse adjacency matrix (many 0s) Allows quicker answer to "find all nodes adjacent to node i"

Related information, such as arc weights, can be used as matrix entries instead of 0/1 values, or as extra member variables in objects stored in an adjacency list.

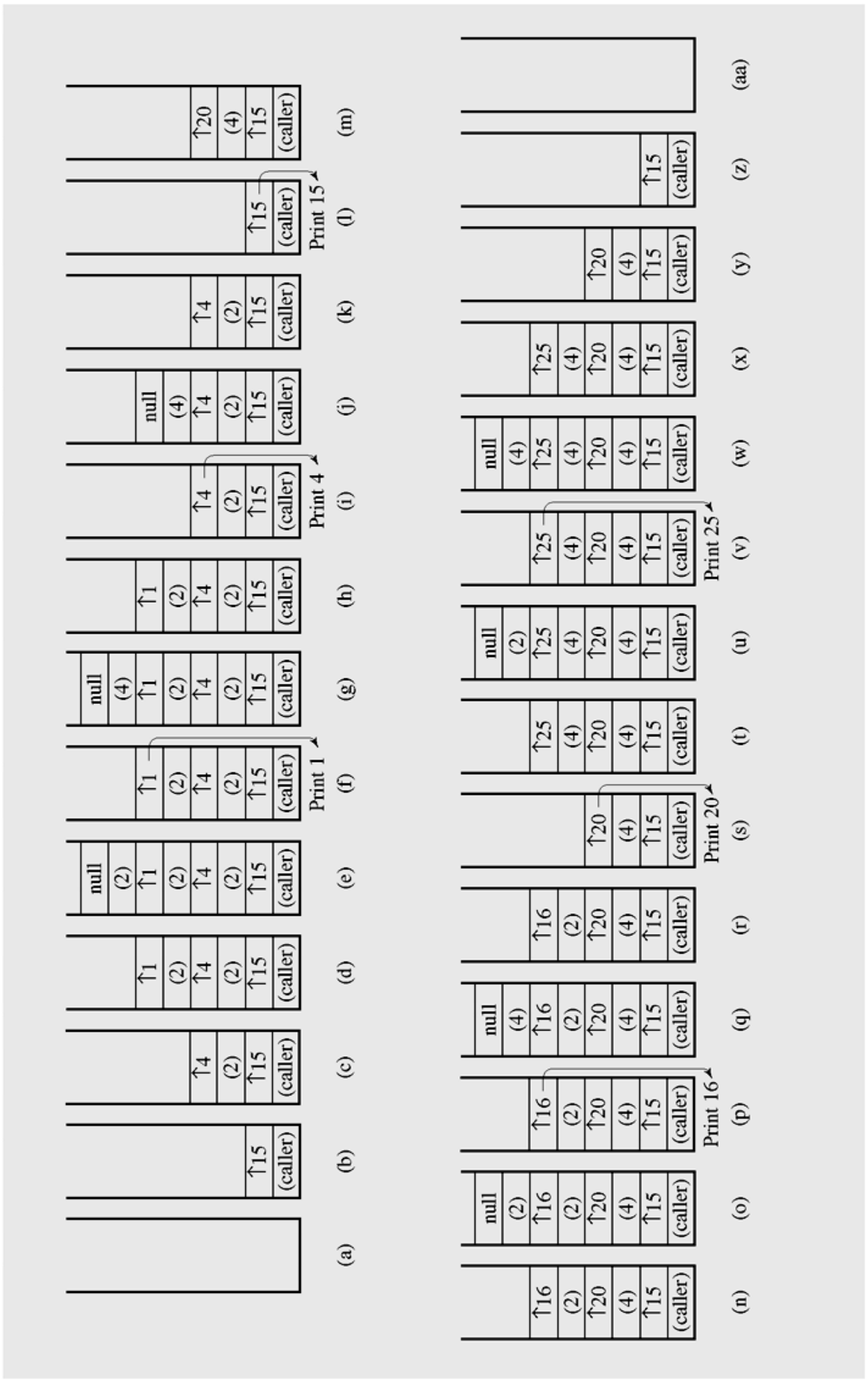
Run-time stack for in-order traversal

Consider an in-order traversal of the following binary tree:



The figure on the next page is used by permission of Cengage Learning from Data Structures and Algorithms in C++ by Adam Drozdek.

FIGURE 6.14 Changes in the run-time stack during inorder traversal.



Search Algorithms

The meta-structure (i.e., the larger view, not the implementation/data structure view) is that we are searching a "list" of records for a **target** key value. We expect a search function, upon receiving the list structure and the target value, to return some indication of whether the search was successful (the target was found in the list) and, if so, where in the list the target was found. (Recall our assumption that the target will occur in at most one place in the list.)

Discussions of four search algorithms appear below, along with a *very rough* pseudocode description of each algorithm. The pseudocode is meant to facilitate your understanding of the algorithm, not necessarily to be translated line for line into C++ code.

Sequential Search

The simplest search algorithm is sequential search. You begin at the beginning of the list and search sequentially through the list until the target is found (successful search) or you run out of the list (unsuccessful search).

Sequential Search Algorithm

```
//searches a random list for a target value

begin at the first position in the list
do
    compare the target against the current record's key
    if target = key, exit the search, returning
        indicator for successful search
        the current position
    if target ¹ key
        move on to the next list element if there is one
until end of list
return indicator for unsuccessful search
```

Successful sequential search animation

Unsuccessful sequential search animation

If the list is already sorted in increasing order by key values, then there are several other algorithm choices. A sequential search can quit if the key values exceed the target - once the key values are too big, the target will never be seen later in the list.

Short Sequential Search Algorithm

```
//searches an increasing sorted list for a target value

begin at the first position in the list
while not off the end of the list and target >  current record's key
   move on to next position
end while
if not off the end of the list and target = current key, exit the search, returning
    indicator for successful search
    the current position
else //either past the end of the list or target <  current key
    exit the search returning indicator for unsuccessful search
```

Successful short sequential search animation

Unsuccessful short sequential search animation

Binary Search

A binary search also assumes that the list has been sorted in increasing order of key values. The general idea behind binary search is to take a stab at the middle of the list and compare the key value there to the target. Based on the result of this comparison, you confine the search to the first (bottom) half or the last (top) half of the list, where you repeat the process again. "Repeating the process again" suggests recursion.

This is roughly the way we search the telephone directory (which the phone company has been kind enough to sort for us) for a given name. We certainly do not use sequential search, starting with "Aardvark, Abe" and going down the list until we find the name we want. We open the book somewhere more or less at random, check a name, and then go forward or backward in the phone book. If we are looking for "Martin" and we open the book to a page with all "K" names, we never have to look in the section of the book before that page. The "binary" part means that we are continually cutting the list in half.

The first binary search algorithm keeps reducing the list until it consists of one item, which may or may not be the target. In other words, this algorithm waits until the end before it decides whether the target is or is not in the list. Notice that when this algorithm cuts the current sublist "in half" and looks at the top half, it does not include the former midpoint, but when it looks at the bottom half, it does include the former midpoint.

Binary Search 1

```
//searches an increasing sorted list for a target value

sublist initially equals entire list
if more than one element in current sublist
   get key value at midpoint position of sublist
   if key < target, repeat search on top half of list above midpoint
   else repeat search on bottom half of list midpoint and below
if current sublist is empty
   exit the search returning indicator for unsuccessful search
if current sublist is one element
   if key = target, exit the search returning
       indicator for successful search
       the current position
   else exit the search returning indicator for unsuccessful search
```

Successful binary search 1 animation
(Note that even though the target equals the first list element compared against, the algorithm keeps going. Once the target occurs at the midpoint position, the midpoint becomes the top of the list and the bottom increases until a one-element list is achieved.)

Unsuccessful binary search 1 animation

The more usual binary search algorithm finds the target element as soon as possible by checking whether each midpoint key equals the target. When this algorithm cuts the current sublist "in half," the former midpoint is never included in the new sublist.

Binary Search 2

```
//searches an increasing sorted list for a target value

sublist initially equals entire list
if current sublist not empty
   get key value at midpoint position of sublist
```

```
        if key = target, exit the search returning
            indicator for successful search
            the current position
        if key < target, repeat search on top half of list above midpoint
        else repeat search on bottom half of list below midpoint
    if current sublist is empty, exit the search returning indicator for unsuccessful search
```

Successful binary search 2 animation

Unsuccessful binary search 2 animation

Searching and Sorting - Overview

Searching and sorting are two very common tasks, especially in data processing applications, so there has been a lot of effort devoted to finding efficient algorithms.

Typically we have a collection of records, and each record has an associated key value (usually one of the member variables of a record object). We have a key value and want to find all or some of the rest of the information in the record with that key. For example, we have a name and want the corresponding address and/or phone number, or we have an employee ID number and we want the employee name.

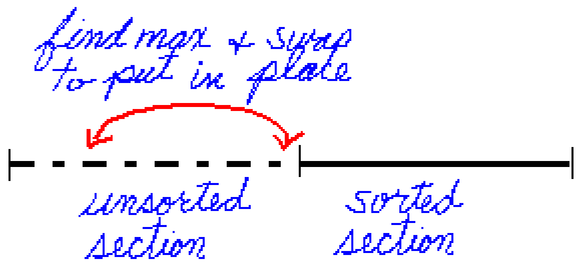
Assumptions:

1. Key values can be compared for equality
2. An ordering relation can be imposed upon key values so that records can be sorted by key values.
3. Records are stored internally in main memory, although they may have been read in from an external file stored on a disk. Because the size of memory is limited, this imposes a limit on the number of records we can handle. This number is a function of both internal memory capacity and the amount of data in each record. Procedures for external searching and sorting - where not all records can be brought into main memory at once and reads from/writes to the external store are a part of the processing - will be touched upon at the end of this course.
4. There is at most one record with a given key value. This means there may be no record with a given key value, but there will never be records with duplicate keys. None of the algorithms we will discuss would fail with duplicate keys, but they also might not behave quite as we might expect. Questions about searching records with duplicate keys would include: if there is more than one record with a given key value, should the program always return the first such record encountered? Should it always return the last such record encountered? Should it return all such records? Should it return some sort of error message? Questions about sorting records with duplicate keys would include: should the sort maintain the original relative order of duplicate records? Because these questions are not addressed and different algorithms will produce different results for duplicate key records, let's just assume there are none.

Selection Sort

The Algorithm

Here is a snapshot of selection sort at work:



In this view, the sorted subsection of the list grows from back to front. We must have an array index to mark the end of the unsorted section. Within the unsorted section, we find the maximum element and swap it with the last element in the unsorted section, which then puts it in its correct location. A variation is to grow the sorted subsection of the list from front to back by finding the minimum element in the unsorted section and swapping it with the first element in the unsorted section, which then puts it in its correct location. The work done by these two versions is identical.

Click here for Selection Sort animation

The implementation is

```
template <class List_entry>
void Sortable_list<List_entry>::selection_sort()
/*
Post: The entries of the Sortable_list have been rearranged so that
      the keys in all the entries are sorted into nondecreasing order.
Uses: max_key, swap.
*/
{
   for (int position = count - 1; position > 0; position--) {
      int max = max_key(0, position);
      swap(max, position);
   }
}
```

with auxiliary functions for finding the maximum key and for swapping the maximum key with the last element in the unsorted section.

The **for** loop is executed n - 1 times (when the unsorted section gets down to one element, that element is in its correct position so there is nothing to do). The work gets done in the auxiliary functions. Swapping, of course, requires three assignments, so we may hope that selection sort does less assignment work than insertion sort, which may have had to shuffle elements through the whole sorted section of the list. On the other hand, selection sort must compare the entire unsorted section of the list on each pass in order to find the maximum element. The unsorted section ranges from size n down to 2 (again, we don't have to do anything when there's only one "unsorted" element). Insertion sort has to go along the sorted subsection (which ranges in size from 1 to n - 1) until it finds where to put the element it is trying to insert. This sounds like equivalent activities, but remember that insertion sort only has to find where to put the element, so it may be able to quit early. So we expect that insertion sort may require fewer compares.

A **for** loop, by its very nature, gets executed a fixed number of times. In addition, scanning for the maximum element depends only on the length of the sublist to be scanned, not the order of its elements. So regardless of the condition of the list - sorted, random, nearly sorted, etc., selection sort's actions are always the same. This simplifies our analysis because there is no best case or worst case.

Analysis

To count assignments, note that selection sort executes its **for** loop n - 1 times. There is a swap on each pass (even if the element is only "swapped" with itself), requiring 3 assignments. Total assignments:

3(n - 1) = 3n + Q(1)

The size of the unsorted section goes from n down to 2. For a list of length k, it takes k - 1 comparisons to find the maximum element (current maximum is initially the first record, and the current-maximum-record gets compared with every other record). Total comparisons:

$$(n-1)+(n-2)+\ldots+1 = \frac{1}{2}(n-1)n = \frac{1}{2}n^{2}+\Theta(n)$$

Sorting Summary

Sorting Summary			
Algorithm	Worst	Average	Best
Insertion	???	random list	already sorted
compares	???	$0.25n^2$ + Q (n)	n - 1
assigns	???	$0.25n^2$ + Q (n)	0
Selection		any list	
compares	same as ->	$0.5n^2$ + Q (n)	<- same as
assigns	same as ->	3n + Q (1)	<- same as

As expected, selection sort, on the average, requires fewer assignments (in fact, a lower order-of-magnitude), while insertion sort requires fewer comparisons (a smaller coefficient). But selection sort, with its uniform behavior, really loses out to insertion sort's best case. Moral so far: use insertion sort if the list already tends toward being sorted, use selection sort for average lists where the records are large and you want to minimize assignments.

Sequential Search Analysis

Best Case

The best case for sequential search is when the target value is found at the first position in the list. This requires exactly 1 comparison.

Worst Case

The worst-case analysis of sequential search is fairly straightforward. Recall that sequential search begins at the front of a list of n elements and checks one-by-one for the target value until it finds the value or runs off the end of the list. What kind of input produces the worst case? There are two different possibilities:

1. The target is found at the very end of the list, in the last place the algorithm looks. How much work (how many comparisons) must be done? n comparisons, one for each list position.
2. The target is not in the list. Again, the algorithm must do n comparisons; the only difference between this case and the previous one is the outcome of the last comparison.

Conclusion: Sequential search of a list of n elements requires n comparisons in the worst case, for both a successful search and an unsuccessful search.

Average Case

What about the "average" number of comparisons done in searching an n-element list? We already know the answer for an unsuccessful search, because the best case, average case, and worst case are all the same - an unsuccessful sequential search will always require n comparisons.

For a successful search, the target value must appear somewhere in the list.

Assumption: the target is equally likely to be any one of the list elements.

This is a big assumption, and characteristic of the kinds of assumptions we must make to get a mathematical handle on "typical input" in order to do an average case analysis.

To see the danger in this assumption, suppose that 90% of the time we search a given list, the target value occurs at the first list position. Obviously, on the average, little work is required. (But is this a "typical" situation or an anomaly?)

We can set up the following table for the work done (number of comparisons made) for successful sequential searches, depending on where the target appears in the list. Note that "location" of the target here goes from 1 to n, as opposed to "position of a list element", which we've used in code, and which goes from 0 to n - 1.

Location of target in list	Number of comparisons required
1 (first element)	1
2	2
3	3

...	...
n (last position in list)	n

To find the average work done, we use the following formula:

$$\text{Average work} = \sum_{\text{all cases}} (\text{work for this case}) * (\text{probability of this case})$$

Under our assumption that the target is equally likely to be found in each of the n positions in the list, the probability of each case is 1/n. The formula for average work then gives us Equation (1) below. (Note that this is how you would find the average of three equally weighted tests, for example - you would add the three scores and divide by 3.)

$$1*\frac{1}{n} + 2*\frac{1}{n} + \cdots + n*\frac{1}{n} = \frac{1+2+\cdots+n}{n} = \frac{1}{2}(n+1) \qquad (1)$$

Equation 1 can be proved true for all n > 0 by mathematical induction.

Shell Sort Analysis

Recall that no general Shell sort analysis exists; instead, the increments are decided on, and an analysis *for those increments* is done. We will look at two choices of increments, and analyze each of these special cases.

Shell increments

Symbolically, we'll denote the successive increments used by

$h_t, h_{t-1}, \ldots, h_1 = 1$

These increment values decrease; the last increment is always 1 because the final pass of Shell sort is just a regular insertion sort on the entire list.

Suppose the list contains n elements. The Shell increments are:

$$h_t = \left\lfloor \frac{n}{2} \right\rfloor, \quad h_{t-1} = \left\lfloor \frac{h_t}{2} \right\rfloor, \quad \text{etc.}$$

In other words, the first increment is half the list size, and successive increments keep getting cut in half.

Example:

Suppose the list to be sorted is

12 3 6 4 82 54 5 9 7 15 61 43 30 21 15 29

Then n = 16. The Shell increments would be

$h_4 = 8, h_3 = 4, h_2 = 2, h_1 = 1$

The initial sublists with $h_4 = 8$ would be:

12 3 6 4 82 54 5 9 7 15 61 43 30 21 15 29

(color-coding - 8 lists, each only 2 elements long)

The second pass, using increment $h_3 = 4$, would involve the four sublists

7 3 6 4 30 21 5 9 12 15 61 43 82 54 15 29

and so forth.
end of Example

Observe that with this choice of increments, on subsequent passes we keep revisiting a lot of the same sets of values (such as "12" and "7" in the above example) that we already put in order, so we are wasting a lot of effort.

Using these increments and this observation, we can construct a specific case where lots of work must be done. First, remember that in insertion sort, both assignments and comparisons required $0.25n^2 + \Theta(n)$ or, on a more gross scale, $\Theta(n^2)$. The n^2 term was driven by the number of times we had to execute the do-while loop looking for the place to fit in the next unsorted element. The assignments and comparisons outside the loop added a constant amount of work as each element got inserted; the sum of all these constant values got absorbed into the Q(n) part of the expression. Hence we will just count the number of "element moves" to insert a new item as the driving work unit.

Now let n be a power of 2, so that all increments are even numbers except h_1 = 1. The example above, with n = 16, has this property. (For the purposes of this discussion, it's probably easiest to suppose that array indices start at 1 instead of 0.) Suppose the original array has the n/2 largest elements stored in all the even array indices and the n/2 smallest elements stored in all the odd indices. (Pathologic case, but possible.) All increments h_k, k ¹ 1, are even, so for any index i, the other numbers in its sublist, i.e., the ones with which it gets sorted, are of the form

$$\text{i} \pm (\text{multiple of } h_k) = \text{i} \pm \text{even} = \begin{cases} \text{even if i is even} \\ \text{odd if i is odd} \end{cases}$$

Therefore the large elements have only been sorted against other larges, and the smalls against other smalls. The large elements are still in even indices, the small elements are still in odd indices, when we come to the final pass with h_1 = 1.

Now consider the ith small number, i £ n/2. It is currently sorted with respect to other smalls, so it is in location 2i - 1. But it belongs in location i, so on the final pass it will have to be moved i - 1 places in the array. This holds for all n/2 small elements, so the number of moves is

$$\sum_{i=1}^{n/2}(i-1) = \underset{\substack{\uparrow \\ \text{position 1}}}{0} + \underset{\substack{\nwarrow \\ \text{position 3}}}{1} + 2 + \ldots + (n/2-1) = \frac{1}{2}\left(\frac{n}{2}-1\right)\left(\frac{n}{2}\right) = \Theta(n^2)$$

Ignoring the constant coefficient, this is the same order of magnitude as regular insertion sort, or selection sort. This is a bad case; is it a *worst* case? Yes; we will see that Q (n^2) is an upper bound for Shell sort, *for these increments*.

First, for ANY set of increments, an increment h_k will result in h_k sublists to sort, each with

~n/h_k

elements. (See the example above, with n - 16 and h_k = 8; there are 8 sublists, each with 2 elements.) For any one insertion sort, the worst case amount of work is Q (input length)^2, i.e., quadratic. Then at increment h_k, there are h_k lists each requiring Q ($(n/h_k)^2$) work. The total work is at most

h_k Q($(n/h_k)^2$) = Q (h_k $(n/h_k)^2$) = Q (n^2/h_k)

If there are t passes, that is, t different increment values, then the total work is at most

$$\sum_{k=1}^{t}\Theta\left(\frac{n^2}{h_k}\right)=\Theta\left(n^2\sum_{k=1}^{t}\frac{1}{h_k}\right)$$

Now, back to Shell increments. Here,

$$\sum_{k=1}^{t}\frac{1}{h_k}=1+\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\ldots+\frac{1}{2^{t-1}}$$

This is a geometric series with first term = 1, common ratio = 1/2. The formula for the sum of the first t terms of a geometric series with first term "a" and common ratio "r" is

$$\frac{a-ar^t}{1-r}$$

which becomes

$$\frac{1-1\left(\frac{1}{2}\right)^t}{1-\frac{1}{2}}=\frac{1-\left(\frac{1}{2}\right)^t}{\frac{1}{2}}=2\left(1-\left(\frac{1}{2}\right)^t\right)=x$$

From the expression for x, it is clear that 1 <= x < 2. Therefore

$$\Theta\left(n^2\sum_{k=1}^{t}\frac{1}{h_k}\right)$$

is Q(n^2).

Conclusion: Shell sort with these increments can do no worse than Q(n^2) , but it can be made to do this badly.

Shell Sort Analysis, 2

Using Shell increments and rigging the data, we can make Shell sort work really hard, producing no better results than our previous sorts. But with seemingly trivial modifications to the increments, we can greatly improve the performance.

Hibbard increments

Now consider increments of the form

1 3 7 15 31 63 �

(which sound pretty close to the Shell increments of 1, 2, 4, 8, 16, 32, 64�)

Here increment

$$h_k = 2^k - 1$$

But unlike the case with Shell increments, **successive increments do not share a common factor.**

Proof: Consider successive increments h_k and h_{k+1}. The larger increment, h_{k+1}, can be written as

$$h_{k+1} = 2^{k+1} - 1 = 2(2^k - 1) + 1 = 2h_k + 1$$

so

$$h_{k+1} = 2h_k + 1 \qquad (1)$$

Suppose that $h_{k+1} = jm$ and $h_k = qm$, that is, these two increments share a common factor of m. We also assume m > 1 because 1 is always a trivial common factor. Then from Equation (1),

$$jm = 2qm + 1$$

or

$$(j - 2q)m = 1$$

Because m and j - 2q are integers with m > 1, this gives a nontrivial factorization of 1, which is a contradiction.

end of proof

The fact that successive Hibbard's increments have no common factors will be key.

We already know that any one pass of Shell sort does h_k sublists of length $\sim n/h_k$. We choose t (the maximum number of increments) so that $n/h_t \sim 2$, that is, so that the shortest lists (the ones that occur with the largest increments, on the first pass) have at least 2 elements. This guarantees that there is something to sort. Then

$$\frac{n}{h_t} = \frac{n}{2^t - 1} \approx 2 \Rightarrow \frac{n}{2} \approx 2^t - 1 \Rightarrow \frac{n}{2} \approx 2^t \Rightarrow \sqrt{\frac{n}{2}} \approx 2^{t/2} \approx h_{t/2}$$

(Assume that t is even so that t/2 is an integer.) This says that the increment size halfway through the process is approximately $\sqrt{n/2}$.

Example:

Suppose n = 32, and the increments are 1, 3, 7, 15 = h_t. Then the first sublist sorted with increment 15 contains elements at 1, 16, and 31, but the last sublist sorted with increment 15 contains elements 15 and 30, so each sublist has at least 2 elements. Here t = 4, t/2 = 2, and the increment size halfway along the list is $h_2 = 3 \approx \sqrt{32/2}$.
end of example

We also know that for any increment, any one pass requires at most Q(n^2/h_k). We'll use this upper bound on those passes for the t/2 increments in the second half of the list of increments, but find a better upper bound on the first half of the increment list (which is used *later*). The upper bound on the second half work is therefore

$$\sum_{k=t/2+1}^{t} \Theta\left(\frac{n^2}{h_k}\right)$$

Now consider that we have done passes h_{k+2}, h_{k+1}, and are about to do h_k. Consider elements list[i] and list[j], i < j. If j - i is a multipleof h_{k+2} or h_{k+1}, then list[i] � list[j], i.e., these list elements have been relatively sorted because they were in the same sublist earlier. But if j - i is a linear combination of h_{k+2} and h_{k+1}, these list elements are still relatively sorted.

Example:

Suppose we have done a 15-sort and a 7-sort and are about to do a 3-sort. Consider list[100] and list[152]. Here j = 152, i = 100, and j - i = 52 = 1*7 + 3*15. Then list[100] � list[152] because

list[100] � list[107] (7-sort)

� list[122] (15-sort)

� list[137] (15-sort)

� list[152] (15-sort)

end of example

Because h_{k+2} and h_{k+1} share no common factors, the key point proved earlier, it turns out that every integer

$$\geq 8h_k^2 + 4h_k$$

is a linear combination of h_{k+1} and h_{k+2}. Although we won't go through a proof of this, we can at least illustrate it with an example.

Example:

Suppose h_{k+2} = 15, h_{k+1} = 7, and h_k = 3. Then

$$8h_k^2 + 4h_k = 8*9 + 4*3 = 84$$

The claim is that every integer � 84 is a linear combination of 15 and 7. Beginning with 83, which does not meet the criterion,

83 - cannot be written as a linear combination of 15 and 7
84 = 0*15 + 12*7
85 = 1*15 + 10*7
86 = 2*15 + 8*7
87 = 3*15 + 6*7
88 = 4*15 + 4*7
89 = 5*15 + 2*7
90 = 6*15 + 0*7
91 = 0*15 + 13*7
92 = 1*15 + 11*7
etc.

end of example

What this fact gives us is that at the pass using increment h_k, whenever any item list[j] is to be inserted, it only has to look at indices that are multiples of h_k before j AND that are closer than

$$8h_k^2 + 4h_k = (8h_k + 4)h_k$$

(Item list[i] at an index i that is farther away is already sorted with respect to list[j] because j - i is a linear combination of h_{k+2} and h_{k+1}.) This gives at most $8h_k + 4$ places to look to insert list[j]. The first item in each of the h_k sublists does not need to be inserted, that is, items list[1] - list[h_k] are the heads of their respective sublists. Therefore at this pass there are at most ($n - h_k$) items to be inserted, each requiring at most ($8h_k + 4$) element moves. The work at this pass is at most

$$(8h_k + 4)(n - h_k) = 8nh_k + 4n - 8h_k^2 - 4h_k) = \Theta(nh_k)$$

The work using the first half of the increment list is

$$\sum_{k=1}^{t/2} \Theta(nh_k)$$

The total work (first half of the increment list plus second half of the increment list) is

$$\sum_{k=1}^{t/2} \Theta(nh_k) + \sum_{k=t/2+1}^{t} \Theta\left(\frac{n^2}{h_k}\right) = \Theta\left(n\sum_{k=1}^{t/2} h_k\right) + \Theta\left(n^2 \sum_{k=t/2+1}^{t} \frac{1}{h_k}\right) \qquad (2)$$

To evaluate Term #1 in Equation (2), note that

$$\sum_{k=1}^{t/2} h_k = \sum_{k=1}^{t/2}(2^k - 1) = \sum_{k=1}^{t/2} 2^k - \sum_{k=1}^{t/2} 1 = \sum_{k=1}^{t/2} 2^k - \frac{t}{2}$$

The summation represents a geometric series with common ratio r = 2 and first term a = 2. We want the sum of the first t/2 terms in this series, so again using the formula for the sum of terms in a geometric series, the above expression becomes

$$\frac{2-2*2^{t/2}}{1-2}-\frac{t}{2}=2(2^{t/2}-1)-\frac{t}{2}=2h_{t/2}-\frac{t}{2}$$

Therefore

$$\Theta\left(n\sum_{k=1}^{t/2}h_k\right)=\Theta(n\left[2h_{t/2}-\frac{t}{2}\right])=\Theta(nh_{t/2}-n)=\Theta(n\sqrt{\frac{n}{2}}-n)=\Theta(n^{3/2})$$

To evaluate Term #2 in Equation (2), note that

$$\sum_{k=t/2+1}^{t}\frac{1}{h_k}\approx\sum_{k=t/2+1}^{t}\frac{1}{2^k}=\sum_{k=0}^{t}\frac{1}{2^k}-\sum_{k=0}^{t/2}\frac{1}{2^k}$$

This gives us the sums of two geometric series, for which again we can apply a formula, resulting in

$$2-\frac{1}{2^t}-\left[2-\frac{1}{2^{t/2}}\right]=-\frac{1}{2^t}+\frac{1}{2^{t/2}}=\frac{1}{2^{t/2}}-\frac{1}{2^t}\approx\frac{1}{2^{t/2}}\approx\frac{1}{h_{t/2}}$$

so that

$$\Theta\left(n^2\sum_{k=t/2+1}^{t}\frac{1}{h_k}\right)=\Theta\left(n^2*\frac{1}{h_{t/2}}\right)=\Theta\left(n^2*\frac{1}{\sqrt{\frac{n}{2}}}\right)=\Theta(n^{3/2})$$

The total work from Term #1 and Term #2 is

$$\Theta(n^{3/2})+\Theta(n^{3/2})=\Theta(n^{3/2})$$

This is an upper bound (for these increments) that breaks the Q(n^2) barrier!!!

Much experimental evidence gives (for these increments) an average work value of Q($n^{1.25}$), but there is no proof (analysis) to back this up.

Shell Sort

The Algorithm

Selection sort is low in assignments because it does 1 swap per pass (swapping the maximum in the unsorted list with the back of the unsorted list). So one move accomplishes a big jump. It is high in comparisons because it keeps looking for the maximum value of the unsorted section, and in the process may repeat comparisons it has made earlier.

Insertion sort minimizes comparisons because it can quit early, but it does a lot of assignments moving the "empty slot" to where it needs to be to insert the item. The moves only occur between adjacent positions in the list. We can keep the relative efficiency of insertion sort with respect to comparisons and improve its efficiency with respect to assignments if we move things in bigger steps or increments. In basic insertion sort, the increment is 1 - we consider the total list, and the items are 1 cell apart.

Shell sort (1959) is named after Donald Shell. An increment (larger than 1) is selected, and the original list is broken down into a series of sublists whose members are *increment* cells apart. Insertion sort is done on each of these sublists. Here the shuffle moves (swaps) take bigger bites of the original list and the items are moving somewhat toward sorted order. Although we are sorting multiple lists, each is short compared to the original, so the sort will go faster. This process is repeated with smaller increments - sublists get longer, but start closer to sorted order, for which insertion sort is quick. Finally, the increment size = 1, i.e., the process finishes with a regular insertion sort.

Example: The original list is

7 4 2 8 10 34 54 23 6 5 12 9 17

The first increment is chosen to be 5. There will be 5 sublists - each shown as a different color:

7 4 2 8 10 34 54 23 6 5 12 9 17

After all the blues are sorted in relative order, all the greens are sorted in relative order, etc., the list is

7 4 2 6 5 12 9 17 8 10 34 54 23

You can see that some of the big numbers are drifting to the back and the small numbers to the front. The next increment is chosen to be 3. There will be 3 sublists:

7 4 2 6 5 12 9 17 8 10 34 54 23

Once these sublists have been sorted we have

6 4 2 7 5 8 9 17 12 10 34 54 23

This is closer to being sorted. On the final pass, the increment = 1, and regular insertion sort is done, resulting in the fully sorted list

2 4 5 6 7 8 9 10 12 17 23 34 54

end of Example

A critical feature of the Shell sort algorithm is that items sorted on one pass remain in sorted order relative to each other on the next, so work is not undone in successive passes. Progress is always made.

Shell Sort Analysis Summary

In practice, Shell sort seems to be quite efficient, but the efficiency varies with the choice of increments. No general complete analysis of Shell sort has been done. Special cases have been examined for specific choices of increments, and even these analyses are rather difficult. We will shortly do two such special analyses, for two sets of increment choices. This will illustrate how the efficiency can change based on the increments chosen.

Overall, what you want to remember is:

- a general analysis of Shell sort has not been done
- the performance has been analyzed only for specific increment choices
- the performance varies considerably depending on the increments chosen
- you want to avoid increments where lots of elements that were in the same sublist on one pass are in the same sublist on another, later pass with a smaller increment. The reason is you put these elements in relative sorted order the first time, and you are just wasting time trying to do it again
- if you avoid the case just described, then empirical evidence gives a work estimate for Shell sort of Q ($n^{1.25}$). The reason this is pretty exciting is that it's the first algorithm we've seen that has "broken the n^2 barrier" for the average amount of work required to sort an n-element list

Sorting Summary

Algorithm	Worst	Average	Best
Insertion	???	random list	already sorted
compares	???	$0.25n^2$ + Q (n)	n - 1
assigns	???	$0.25n^2$ + Q (n)	0
Selection		any list	
compares	same as ->	$0.5n^2$ + Q (n)	<- same as
assigns	same as ->	3n + Q (1)	<- same as
Bubble	reverse sorted	random list	already sorted
compares	$0.5n^2$ + Q (n)	$0.5n^2$ + Q (n)	$0.5n^2$ + Q (n)
assigns	$1.5n^2$ + Q (n)	$0.75n^2$ + Q (n)	0
Smart Bubble	reverse sorted	random list	already sorted
compares	$0.5n^2$ + Q (n)	$0.5n^2$ + Q (n)	n - 1
assigns	$1.5n^2$ + Q (n)	$0.75n^2$ + Q (n)	0
Shell sort	reverse sorted	random list	already sorted
compares		Q ($n^{1.25}$) - experimental -	

	---	depends on increment used	---
assigns	---	---	0

Shortest Path

Problem: Given a weighted graph (or digraph), find the shortest path (the path of least weight) from vertex x to all other vertices.

The shortest path problem has many applications. Any routing problem - sending packets over computer networks, passengers over plane routes, goods over truck routes, etc. - is a candidate for a shortest path solution. Although there are several shortest-path algorithms, the best-known is *Dijkstra's algorithm*, named for Edsgar Dijkstra, author of the famous "Go to statement considered harmful" letter. Again we are assuming that this will be a member function in a graph class, and the graph could be implemented as an adjacency list or as an adjacency matrix. An advantage goes to the adjacency matrix for two reasons. First, instead of a_{ij} recording 1 or 0 (adjacent nodes or not), a_{ij} can be the weight of the arc between node i and node j, so we get the weight information without any additional overhead. For this algorithm, where there is no arc, instead of setting a_{ij} to 0, it is set to some "infinity" value (something larger than any legitimate weight). Second, and more important, in Dijkstra's algorithm we need to know the weight of an arc, if it exists, between two arbitrary nodes i and j, and the adjacency matrix is a direct access structure, so we can get this information in one step.

A set S is maintained. S consists of nodes whose shortest distance from x using only nodes in S is known. Intially, S contains only node x. S grows by choosing next the node (incorporating some tie-breaking rule if necessary) whose distance from x using only nodes in S is minimal.

Dijkstra's algorithm falls into the class of algorithms known as **greedy algorithms**. Such algorithms try to optimize their situation at any point in time using only the "local" information known at that time. Then when all is said and done, this strategy turned out to produce an overall optimal solution. Thus for the shortest path algorithm, we bring in the node with the shortest path using only the partial paths we know so far (i.e., using only nodes in S), and this turns out to produce the overall shortest path in the end. A boolean array *IN* can be used to keep track of which nodes are in S. A *distance* array maintains information for all nodes on the minimum distance from x using only nodes in S. The *distance* array is initialized to the arc length from x to each node if such an arc exists, otherwise the infinity value is used. When S contains only node x (the initial case), the only path from x using only nodes in S is a direct arc from x. Whenever a node v is brought into S, the distance of all the other nodes not in S needs to be checked to see if a shorter path can be found using this new node v. The algorithm terminates when all nodes have been brought into S. At this point S = G, so the overall shortest path has been found. But you don't just want to know what the shortest path length from x to some node y is, you want to know what the actual path is. Hence you should keep track of a "parent" node so you can trace the nodes along the shortest path. When the distance of some node u is reduced because node v has been added to S, then v is the "parent node" of u , that is, the arc v-u would be part of u's shortest path.

Here's the general description, assuming the adjacency matrix A has been initialized as described above and that x has been given as the source node:

```
For all nodes v
    IN[v] = 0                    //S is initially empty
    distance[v] = $a_{xv}$            //weight of arc from x to v
    parent[v] = x                //all paths at this point have source as parent
IN[x] = 1                        //add source node to S
distance[x] = 0                  //"0-length" path from x to x

While there are nodes not in S
    Find the node v not in S such that dist[v] is minimum
    IN[v] = 1                    //add v to S
    For all nodes u with u not in S
        if (distance[u] > distance[v] + $a_{vu}$)   //shorter path through v
                distance[u] = distance[v] + $a_{vu}$  //readjust distance for shorter path
                parent[u] = v    //path goes from v to u
```

Note that if the graph is not connected, some distance values will remain set to infinity, which conveys the information that there is no path (of whatever distance) from x to such nodes.

If you want the shortest path from the source node x to some one specific destination node y, you can stop as soon as y is brought into S.

Analysis: The analysis is fairly simple because of the loops in the code. The first for loop initializes *IN* and *distance* for each node, so the first part of the code is Q(n). When the while loop is first encountered, there are n - 1 nodes not in S and one node is removed from S on each pass, so the while loop is executed n - 1 times. Finding the minimum distance node not in S requires walking through all the nodes, as does the next for loop. So the second part of the code has two sequential Q(n) loops within the while loop, resulting in an Q (n^2) algorithm.

But - Dijkstra's algorithm does NOT work if negative weights are allowed.

All pairs shortest path: It turns out there is a very simple algorithm (Floyd's Algorithm) to compute the distance of the shortest path between any two nodes. Again assume that the graph is represented as an adjacency matrix with entries representing arc weights and a_{ii} = 0 for all i.

```
for k = 1 to n do
    for i = 1 to n do
        for j= 1 to n do
            if A[i, k] + A[k, j] < A[i, j] then
                A[i, j] = A[i, k] + A[k, j]
```

Initially, a_{ij} is the length of a 1-length path (if one exists) from i to j. For any fixed k, the two inner for loops look at all pairs of nodes to see if there is a shorter path going through node k.

There are several remarkable features of this algorithm. One is its simplicity, from which it's clear to see that it's an $Q(n^3)$ algorithm. [If you execute Dijkstra's algorithm once for each starting node x, it is also $Q(n^3)$.] Second, it works with negative weights. And finally, it operates on the adjacency matrix *in place*, that is, there is only one copy of the array but changing the value of a_{ij} in the middle of the algorithm occurs in such a way as to not mess up anything later on, and when the algorithm is finished, the entry in a_{ij} is now the distance of the shortest path from node i to node j. For example, consider i = 1, k = 3, j = 4; if $a_{13} + a_{34} < a_{14}$, then a_{14} will be changed. But on the previous pass through the outer loop, when k = 2, the value of a_{13} may have been changed. That's OK - a_{13} is the shortest path at this point using node 2, if we can now make a shorter 1-4 path using both nodes 2 and 3, so much the better.

Smart Bubble Sort

The straightforward version of bubble sort uses two nested **for** loops which, of course, execute a fixed number of times. Bubble sort can be improved upon by recognizing (through some sort of flag variable) whether an entire pass through the inner loop has been made where no swaps were required. Once that happens, the list is in sorted order and the algorithm can terminate.

The smart version of bubble sort would be able to recognize an already sorted list after just one pass through the outer **for** loop. This one pass would require n - 1 comparisons and no assignments, hence smart bubble sort would behave the same as the optimal insertion sort on already-sorted or nearly-sorted lists. Unless the list gets sorted early, however, smart bubble sort is no improvement over regular bubble sort.

Sorting Summary			
Algorithm	Worst	Average	Best
Insertion	???	random list	already sorted
compares	???	$0.25n^2$ + Q (n)	n - 1
assigns	???	$0.25n^2$ + Q (n)	0
Selection		any list	
compares	same as ->	$0.5n^2$ + Q (n)	<- same as
assigns	same as ->	3n + Q (1)	<- same as
Bubble	reverse sorted	random list	already sorted
compares	$0.5n^2$ + Q (n)	$0.5n^2$ + Q (n)	$0.5n^2$ + Q (n)
assigns	$1.5n^2$ + Q (n)	$0.75n^2$ + Q (n)	0
Smart Bubble	reverse sorted	random list	already sorted
compares	$0.5n^2$ + Q (n)	$0.5n^2$ + Q (n)	n - 1
assigns	$1.5n^2$ + Q (n)	$0.75n^2$ + Q (n)	0

Sorting - Overview

Now that we have spent weeks on searching, it is time to look at the other major task in information processing - sorting. As before, we assume that records are uniquely identified by a key value, and for simplicity, we'll just work with key values. We assume that key values can be ordered, otherwise what would sorted order mean? We assume also - unless otherwise noted - that the set of records is small enough to be stored entirely in main memory, either in an array or a linked list.

All our algorithms will perform operations of comparing key values against each other, and rearranging the key values within the list as needed. The main ideas of the algorithms are the same whether the data structure for the list is an array or a linked list, but rearranging key values is done differently. In an array, rearranging is done by shuffling records around in the array, using array assignments. In a linked list, rearranging is done by relinking pointers, usually a less expensive operation if the records are large. Our goal is to compare these various algorithms to each other and as long as we consider the same data structure for every algorithm, we'll get the same relative amounts of work whether that data structure is an array or a linked list.

That said, we'll proceed under the following conditions:

1. We'll use an array-based implementation.
2. We'll sort in ascending order.

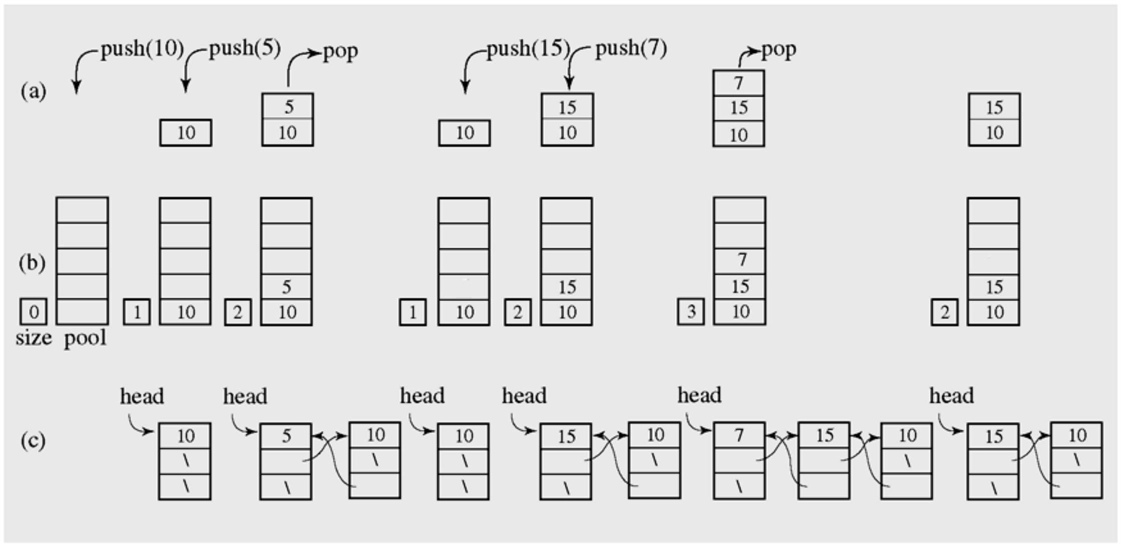
One might ask why there isn't just one "best" sorting algorithm that everybody uses? The answer is that the performance of the algorithm often depends on the initial configuration of the list - is it random? is it close to sorted order? is it in reverse sorted order? etc.

For each of our sorting algorithms, we will try to give a best case, an average case, and a worst case analysis.

The stack ADT and two implementations

The following figure is used by permission of Cengage Learning from Data Structures and Algorithms in C++ by Adam Drozdek.

FIGURE 4.6 A series of operations executed on an abstract stack (a) and the stack implemented with a vector (b) and with a doubly linked list (c).



Summary and Conclusions

The following table summarizes the general formulas we have found for searching n-element lists. Except we'll do some further simplifications and use

$$\lg n \text{ as an approximation to } \lg(n+1)$$
$$1 \text{ as an approximation to } \frac{n+1}{n}$$

These are quite close for large values of n; we don't need exact values, so we can be a little sloppy.

	Worst Case	**Average Successful**	**Average Unsuccessful**
Sequential	n	$\frac{1}{2}(n+1)$	n
Short Sequential	???	???	???
Binary 1	é 1 + lgnù	1 + lg n	1 + lg n
Binary 2	2*é lgnù	2lg n - 3	2 lg n

Note that unlike the sequential search, there is no difference in Binary 1 between the average work in a successful search or an unsuccessful search, and little difference in Binary 2.

Conclusions

1. If the list is unsorted, sequential search is the only choice.

2. The fact that the other three algorithms are an improvement on sequential search must be tempered with the requirement that the list has to be sorted. Sorting is an expensive operation, and if the list is to be searched only infrequently, then leave it unsorted and stick to sequential search.

3. Similarly, for small n, use sequential search. It is simple and straightforward to implement, and the difference between n and lg n is insignificant for small n. In fact, for very small n, sequential search is actually better.

4. For large n, either binary search beats sequential search (~lg n instead of ~n).

5. Binary 1 beats Binary 2, contrary to our original impression of Binary 1. This is because of the doubling factor in Binary 2. Binary 2 gives the opportunity for a few really early exits, but these occur at the relatively few nodes near the top of the tree.

Topological Sorting

Consider a directed graph with no cycles. Then it is always possible to find an ordered list (a sort) of the nodes of G such that if there is an arc from v to w, then v precedes w in the list. The result of a topological sort is a linear ordering that satisfies all "prerequisites." A topological sort for a given graph is not necessarily unique.

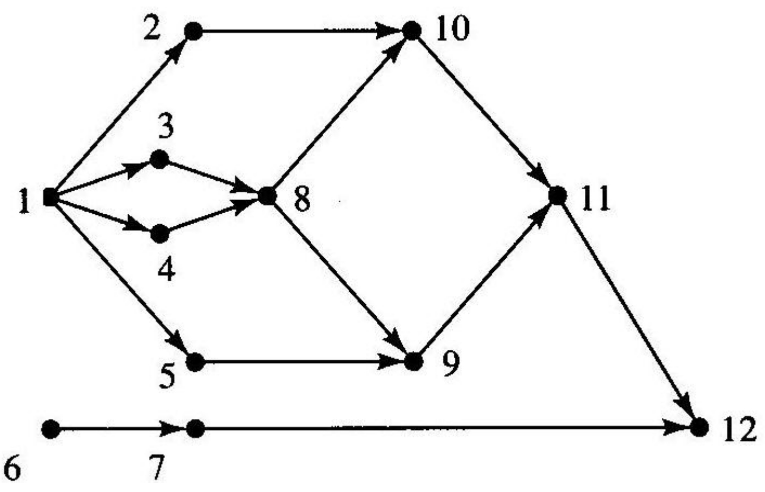
Topological sorting is good for any task list where some tasks must be completed before others can begin. Note that if you had modules of code running on multiple processors and you wanted to convert this into a "straightline" program running on a single processor, you could use topological sort.

Example: (from Gersting, *Mathematical Structures for Computer Science*, 6 ed, W.H. Freeman, 2007)

Ernie and his brothers run a woodworking shop in the hills of New Hampshire that manufactures rocking chairs with padded cushion seats. The manufacturing process can be broken down into a number of tasks, some of which have certain other tasks as prerequisites. The following table shows the manufacturing tasks for a rocking chair and the prerequisite tasks for each one.

Task	Prerequisite Tasks
1. Selecting wood	None
2. Carving rockers	1
3. Carving seat	1
4. Carving back	1
5. Carving arms	1
6. Selecting fabric	None
7. Sewing cushion	6
8. Assembling back and seat	3, 4
9. Attaching arms	5, 8
10. Attaching rockers	2, 8
11. Varnishing	9, 10
12. Adding cushion	7, 11

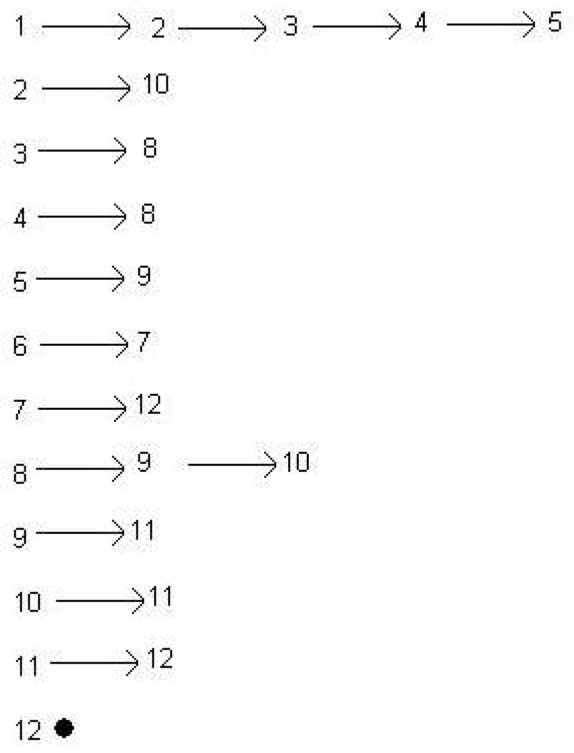
This set of tasks can be represented as an acyclic directed graph:



A topological sort relies on the idea of **successor** and **predecessor** nodes. In the above figure, node 10 is an immediate predecessor of node 11; node 8 is also a predecessor of node 11 because task 11 cannot be done until task 8 is completed. Likewise, node 11 is an immediate successor of node 10 and a successor of node 8. If we consider what a topological sort should look like, the end of the ordered list should contain a node that has no successors, called a **sink** or a **minimal vertex**. The front of the list should contain a node that has no predecessors.

An algorithm for finding a topological sort can be found using a modification of depth-first search traversal. Recall that depth-first search uses an adjacency list representation of the graph and, from an arbitrary start node, walks through all adjacent unvisited nodes, recursively visiting each of those nodes (and their adjacency lists) until a dead end is reached, then backtracking out of the recursion.

The adjacency list for the above graph is



An outline of the topological sort algorithm is given below. Because of recursion, there is a top-level function *TopologicalSorting* that calls a recursive function *TS*. The description here assumes that this process is done within a Graph class so a Graph object would invoke these functions and would not be passed as a parameter. However, the list to be sorted is passed as a parameter.

In the top-level function there is an array of vertices called *num* that is initialized to all 0's. A vertex with a *num* value of 0 is unvisited. An arbitrary unvisited node invokes the recursive TS function. After all the recursive calls have been unwound and all nodes have been visited, the list is printed out.

```
void Graph::TopologicalSorting(list <Vertex> &TheList)
{
    for all vertices v, set num[v] to 0;
    set i = 1
    while there is a vertex v with num[v] equal to 0
      TS(v, num, i, TheList)
    print out TheList
}

void Graph:: TS(Vertex v, int num[ ], int& i, list <Vertex> &TheList)
{
   set num[v] value to i, then increment i        //node v is now visited
   for all vertices u adjacent to v
      if num(u) equals 0
         TS(u, num, i, TheList)
   insert v at front of TheList                    //all successors of v are in the list
}
```

This approach builds the sorted list from back to front. At the back goes a vertex with no successors, i.e., the last vertex cannot be a predecessor of any other vertex. Such a vertex must exist, or the graph would have cycles. This is actually the base step of the recursion - any vertex with no successors stops the recursion and can go into the sorted list at once. (Information on the successors of a node comes for free with the adjacency list implementation; if w is on v's adjacency list, then v is a predecessor of w, or w is a successor of v. A node with no successors has an empty adjacency list.) The recursive part is to process and place into the sorted list all successors of a node, then the node itself.

In the Example above, after the initialization, we'll pass the (unvisited) node 1 into the TS function. Node 1 is marked as visited, then we walk down its adjacency list. Node 2 is unvisited, so pass 2 to TS. Walk down 2's adjacency list. Node 10 is unvisited, so pass 10 to TS. Walk down 10's adjacency list. Node 11 is unvisited, so pass 11 to TS. Walk down 11's adjacency list. Node 12 is unvisited, so pass 12 to TS (note we are now 5 layers deep in recursion). Node 12 has no adjacent vertices, the for loop condition in TS is immediately false, so 12 is inserted at the front of TheList, and the 12-invocation of TS is complete. Back in the Node 11 invocation, there are no more unvisited nodes on 11's adjacency list, so 11 goes into the list in front of 12. Backing up to node 10, it's list is also exhausted, put 10 at the front of the list. Likewise node 2 goes next, then we are back in node 1's list, moving along to unvisited vertex 3, which will cause another invocation of TS. Continue... After node 1 goes into the list, back up in the top-level function, there's another unvisited vertex (node 6), so invoke TS and pass node 6. Continue... At the end of the entire process, the final list is

6 7 1 5 4 3 8 9 2 10 11 12

Note that this ordering meets all "prerequisite" requirements, for example, tasks 2 and 8 are both completed before task 10. This is not the only topological sort of this data. Below is another list that meets our requirements.

1 3 4 2 5 6 8 10 9 11 7 12

As in the traversal algorithms, there is Q(n) work to initialize the visited array and Q(max(n, e)) to traverse all the adjacency lists.

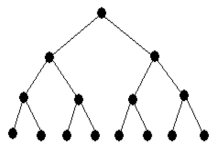
Where we are now

To summarize where we stand at this point:

- We can build (insert items into) binary search trees, remove items from binary search trees, and do binary tree searches on binary search trees.
- A binary tree search on an **ideally-shaped** BST has the same efficiency as our Binary 1 and Binary 2 searches - Q(lg n).
- Insertion and removal in a BST are just about the same amount of work as searching. To insert a node, we search for where it would be if it were there and then insert it (by adjusting a few pointers). To remove a node we search for it and then remove it (by adjusting a few pointers); actually there is a little more work here if we have to search for the node's immediate predecessor, but at most going down one branch of the tree to a leaf.
- Maintaining a BST - depending again on the shape of the tree - is more efficient than maintaining a sorted array (used for Binary 1 and Binary 2), where inserts and deletes can be expensive because a possibly large number of array elements must be shuffled. Again, tracing one branch of a tree - even with two comparisons at each node to decide which way to go - is generally less work than moving n elements over.
- The shape of a BST for a given set of data elements depends on the order in which the elements are inserted.

The shape of the BST is therefore critical to the amount of work that is required for searching (our main concern) and also for maintenance.

So what can we say about the shape of the tree? The best case is that we get a full binary tree (minimum height for the number of nodes).

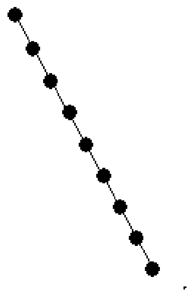


The tree would then look and act like the decision tree for Binary 2 searching, so the average number of comparisons, as we learned before, would be roughly

success: 2 lg n - 3
unsuccess: 2 lg n

or approximate 2 lg n for either case.

The worst case is that the tree turns into a "chain".



On such a tree, the binary tree search degenerates into something similar to short sequential search, but it's even worse because it requires two comparisons (test for equality first), rather than one, to decide that the target is not this node and to go on to the next node. At any rate, the work is Q(n).

If the data being inserted into the tree is relatively "random", then the shape of the tree is going to be somewhere between the best case and the worst case. However, the worst case will happen more often than one might expect. Suppose the data is stored on a file in *sorted order* (not unusual); inserting sorted data into a BST is exactly what causes the chaining effect.

Throwing out the extreme cases of best and worst, is there any estimate we can make for the AVERAGE work to search an AVERAGE binary search tree (one where we make no assumptions about the shape)?