Chapter 1 Review of Prerequisite Topics

1.1 Mathematical Preliminaries

This is the math that you are required to know for the remainder of the course. It is extremely important that you commit these formulae to memory and know how to apply them in practice, because they do arise often in the study of algorithmic analysis, and in computer science in general.

1.1.1 Series Summations

Arithmetic Series The formula for an *arithmetic series* is

$$\sum_{k=0}^{n} k = \frac{n(n+1)}{2} \tag{1.1}$$

Equation 1.1 is easily proved by mathematical induction. From this formula you can solve the more general arithmetic series sum of the form

$$\sum_{k=0}^{n}(ak+b) \tag{1.2}$$

by writing it out directly and redistributing the terms of the series:

$$\begin{aligned}\sum_{k=0}^{n}(ak+b) &= a\sum_{k=0}^{n} k + \sum_{k=0}^{n} b \\ &= \frac{an(n+1)}{2} + (n+1)b \end{aligned} \tag{1.3}$$

If it appears that the series does not start at 0, but say, at c, then observe that

$$\begin{aligned}\sum_{k=c}^{n}(ak+b) &= (a(c+0)+b) + (a(c+1)+b) + ... + (a(c+n-c)+b) \\ &= \sum_{k=0}^{n-c}(a(c+k)+b) &(1.4)\\ &= \sum_{k=0}^{n-c}(ac+ak+b) &(1.5)\\ &= \sum_{k=0}^{m}(ak+d) &(1.6)\end{aligned}$$

where $m = n - c$ and $d = ac + b$. In other words, it can always be viewed as starting at 0.

To illustrate, suppose you need to find the sum of the sequence $9, 13, 17, 21, 25, 29, 33, ..., 257$. You observe that the difference between each pair of terms is the constant $a = 4$, from which you can conclude that this is an arithmetic series. The first term is 9, so you can take $b = 9$. So you know that $a = 4$ and $b = 9$. Now you need to know the value of n. The last term is $an + b = 257$; so $4n + 9 = 257$. Solving for n we get $n = (257 - 9)/4 = 62$. Therefore, applying Equation 1.3, the sum is $4 \cdot 62 \cdot 63/2 + 63 \cdot 9 = 133 \cdot 63 = 8379$.

Quadratic Series The formula for the sum of the sequence of squares 1, 4, 9, 16, and so forth, is

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6} \tag{1.7}$$

This can be proved by mathematical induction, which we do below.

Higher Order Series The quadratic series is the special case of $m = 2$ in the more general series

$$\sum_{k=1}^{n} k^m \approx \frac{n^{m+1}}{m+1},\ m \neq -1 \tag{1.8}$$

A discrete sum such as the above can be viewed as an approximation to the definite integral that defines the area under the curve x^m between the points $x = 0$ and $x = n$. From calculus we know that $\int_0^n x^m dx = n^{m+1}/(m+1)$. When $m = -1$, the denominator in Eq. 1.8 is zero and the right hand side is undefined. In this case there is a different approximation, which can also be seen through the perspective of calculus:

$$H_n = \sum_{k=1}^{n} \frac{1}{k} \approx \int_1^n \frac{1}{x} dx = \ln n \tag{1.9}$$

.

The sum in Equation 1.9 for each value of n is given a name, the ***harmonic number*** of order n, H_n. The absolute difference between the n^{th} harmonic number H_n and the integral $\int_1^n \frac{1}{x} dx$, as $n \to \infty$, is ***Euler's constant***, $\gamma \approx 0.577216$.

Geometric Series The formula for the sum of a *geometric series* for $x \neq 1$, is

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1} \tag{1.10}$$

This is proved directly by showing that the product of the left-hand-side and the denominator of the right hand side is equal to the numerator of the right-hand-side. The left-hand side is the polynomial of degree n all of whose coefficients are 1. When it is multiplied by $x - 1$, all terms drop out except the x^{n+1} and the 1. When $0 < x < 1$, as $n \to \infty$, this converges to $1/(1 - x)$.

Another series that we will encounter quite a bit is

$$\sum_{k=0}^{n} k a^k \ 0 < a \tag{1.11}$$

where a is either 2 or 1/2. Although it is a bit simpler to derive the formula for this series when $a = 2$ or $a = 1/2$, we derive it for arbitrary a as follows, using Equation 1.10. First observe that

$$\begin{aligned} a\left(\sum_{k=0}^{n} ka^k\right) &= \sum_{k=0}^{n} ka^{k+1} \\ &= a^2 + 2a^3 + 3a^4 + ... + (n-1)a^n + na^{n+1} \end{aligned} \tag{1.12}$$

Second, expand the original sum in Eq. 1.11:

$$\sum_{k=0}^{n} ka^k = a + 2a^2 + 3a^3 + 4a^4 + ... + na^n \tag{1.13}$$

and subtract Eq. 1.12 from Eq 1.13, adding 1 and subtracting 1:

$$\begin{aligned} \sum_{k=0}^{n} ka^k - a\left(\sum_{k=0}^{n} ka^k\right) &= 1 + a + a^2 + a^3 + a^4 + ... + a^n - na^{n+1} - 1 \\ &= \left(\sum_{k=0}^{n} a^k\right) - na^{n+1} - 1 \\ &= \left(\frac{a^{n+1}-1}{a-1}\right) - \left(1 + na^{n+1}\right) \end{aligned} \tag{1.14}$$

The left hand side of this equation is

$$\sum_{k=0}^{n} ka^k - a\left(\sum_{k=0}^{n} ka^k\right) = (1-a)\left(\sum_{k=0}^{n} ka^k\right)$$

so we can divide both sides of Equation 1.14 by $(1-a)$ and, adjusting minus signs on the right-hand side, we get

$$\sum_{k=0}^{n} ka^k = \frac{\left(1 - a^{n+1}\right)}{(a-1)^2} + \frac{1 + na^{n+1}}{a-1} \tag{1.15}$$

When we substitute $a = 1/2$ in Equation 1.15, we have

$$\begin{aligned} \sum_{k=0}^{n} \frac{k}{2^k} &= \frac{\left(1 - 1/2^{n+1}\right)}{1/4} + \frac{1 + n/2^{n+1}}{-1/2} \\ &= 4 - \frac{1}{2^{n-1}} - 2 - \frac{n}{2^n} \\ &= 2 - \frac{1}{2^{n-1}} - \frac{n}{2^n} \end{aligned} \tag{1.16}$$

As $n \to \infty$, the right hand side approaches 2, since both $1/2^{n-1}$ and $n/2^n$ approach 0 as $n \to \infty$. Hence

$$\sum_{k=0}^{\infty} \frac{k}{2^k} = 2$$

When $a = 2$ in Eq. 1.15the result, which you can verify, is

$$\sum_{k=0}^{n} k2^k = 2 + (n-1)2^{n+1} \tag{1.17}$$

1.1.2 Modular Arithmetic (Congruences)

You should be familiar with the basic rules of modular arithmetic, but perhaps you have not heard the language associated with it. Computer science students write $a = b \,\%\, N$ to mean that a is the remainder of the integer division b/N. But actually, in mathematics, we say that two numbers a and b are ***congruent modulo*** N if their absolute difference, $|a - b|$, is divisible by N, or in other words, if there exists an integer q such that $q \cdot N = |a - b|$, and we write $a \equiv b \, mod \, N$, or $a \equiv_N b$. I will use both notations here.

When numbers are congruent modulo N, their remainders when divided by N are the same. E.g., $53 \equiv_8 29 \equiv_8 13$ because they all have remainder 5 when divided by 8. In general, modular arithmetic obeys these same rules as integer arithmetic:

- If $a \equiv b \, mod \, N$, then $a + c \equiv b + c \, mod \, N$ and
- If $a \equiv b \, mod \, N$, then $ad \equiv bd \, mod \, N$

With ordinary integer arithmetic, we know that if $ab = 0$, then either $a = 0$ or $b = 0$. But if $ab \equiv 0 \mod N$, it does not imply that one of a or b must be 0. For example, $ab \equiv 0 \mod 12$ can be true if $a = 3$ and $b = 4$.

Furthermore, with rational numbers, we know that for any number $a \neq 0$, $ax = 1$ has a unique solution called its multiplicative inverse. But with modular arithmetic this is not true; a number does not necessarily have a multiplicative inverse. For example there is no x such that $3x \equiv 1 \mod 12$.

When N is a prime number, however, which we will now write as p instead of N, the picture becomes much more interesting, because the set of numbers $0, 1, 2, ..., p - 1$ has been turned into a ***field***, specifically a ***finite field***, and the following statements are true:

- $ab \equiv 0 \, mod \, p$ implies that at least one of a or b is divisible by p. For example, if $ab \equiv 0 \, mod \, 53$ then either a or b is 0 or 53.
- If $ax \equiv 1 \, mod \, p$, then x is uniquely determined: there is a single integer x for which it is true, and it is called the ***multiplicative inverse*** of a. For example, if $8x \equiv 1 \, mod \, 11$, then $x = 7$ because $8 \cdot 7 = 56 \equiv 1 \, mod \, 11$.
- The equation $x^2 \equiv a \, mod \, p$ has either no solution or exactly two solutions if $0 < a < p$. For example, $x^2 \equiv 1 \, mod \, 11$ has the solutions $x = 1$ and $x = 10$, whereas the equation $x^2 \equiv 2 \, mod \, 11$ has no solutions.

1.1.3 Greatest Common Divisor

The ***greatest common divisor*** of two integers a and b is the largest integer that divides both a and b, which we denote by $gcd(a, b)$. For example, $gcd(24, 30) = 6$ and $gcd(5, 15) = 5$. Formally, d is the greatest common divisor of a and b if d divides both a and b and if any number c divides both a and b, then c divides d. When $gcd(a, b) = 1$, we say that a and b are ***relatively prime***, or ***co-prime***. For example, 5 and 8 are relatively prime. For the domain of integers, every pair of numbers has a unique greatest common divisor. For other domains, the *gcd* is not necessarily unique, but we restrict our discussion to the integers. An important result concerning greatest common divisors is the following theorem.

Theorem 1. *Let a and b be two integers and let $d = gcd(a, b)$. Then there exist two integers x and y such that $d = ax + by$.*

Proof. Let $\mathbb{N}$ be the set of all integer linear combinations of a and b. Formally, $\mathbb{N} = \{ra + sb \,|\, r, s \in \mathbb{Z}\}$. Observe that $\mathbb{N}$ is closed under addition (and subtraction) and multiplication, because for any integers $r_1, s_1, r_2, s_2, t \in \mathbb{Z}$, we have

$$(r_1a + s_1b) \pm (r_2a + s_2b) = (r_1 \pm r_2)a + (s_1 \pm s_2)b \in \mathbb{N}$$

since $(r_1 \pm r_2), (s_1 \pm s_2) \in \mathbb{Z}$. Also,

$$t(ra + sb) = (tr)a + (ts)b \in \mathbb{N}$$

since $(tr), (ts) \in \mathbb{Z}$.

Since $\mathbb{N}$ is a subset of the integers, it has some smallest positive member. Let d be the smallest positive number in $\mathbb{N}$. By the divisibility property of the integers, we know that, for any $n \in \mathbb{N}$ there are integers q and r such that

$$n = qd + r$$

where $r = 0$ or $0 < r < d$. Suppose $r \neq 0$. Then $0 < r < d$ and $r = n - qd = 1 \cdot n + (-q) \cdot d$. Since $n \in \mathbb{N}$ and $d \in \mathbb{N}$ and $\mathbb{N}$ is closed under addition, r must be in $\mathbb{N}$ as well. But then r would be a member of $\mathbb{N}$ smaller than d which is a contradiction, because we chose d to be the smallest positive member of $\mathbb{N}$. Therefore, $r = 0$. But then

$$n = qd$$

for some $q \in \mathbb{Z}$. This shows that $d|n$. Since we chose $n \in \mathbb{N}$ arbitrarily, this shows that for every $n \in \mathbb{N}$, d is a divisor of n. Let $r = 1$ and $s = 0$. Then

$$d \,|\, (ra + sb) \Rightarrow d \,|\, a$$

and letting $r = 0$ and $s = 1$

$$d \,|\, (ra + sb) \Rightarrow d \,|\, b$$

so d is a common divisor of a and b. Suppose that c divides a and c divides b. Then clearly c divides ra and c divides sb, so $c \,|\, (ra + sb)$. This implies that c divides all numbers in $\mathbb{N}$. In particular, c divides d, which implies that $c \leq d$ and therefore d must be a greatest common divisor of a and b. Is it unique? Since $d \in \mathbb{N}$, there are integers x and y such that $d = ax + by$. If there were another number d_1 that was a *gcd* of a and b then we would have to have $d \,|\, d_1$ by the definition of the *gcd*, and since $d = ax + by$ and d_1 divides a and b, $d_1 \,|\, d$. If $d \,|\, d_1$ and $d_1 \,|\, d$ then $d = d_1$. Therefore d is the unique *gcd* of a and b. □

This is a very important statement. The *gcd* of any two numbers is a linear combination of the two numbers. If a and b are relatively prime, their *gcd* is 1, and therefore there are x and y such that $1 = ax + by$. This also implies that every integer c can be written as a linear combination of a and b, because $c = c \cdot 1 = c \cdot (ax + by) = acx + bcy$. The x and y do not have to be positive; in fact it is not always possible to find two non-negative integers x and y in this theorem. But under certain conditions, we can:

Theorem 2. *Let a and b be two relatively prime positive integers. If $d \geq (a-1)(b-1)$ then there exist non-negative integers x and y such that $d = ax + by$.*

Proof. Since a and b are relatively prime, $1 = gcd(a, b)$. By Theorem 1, there exist integers x and y such that $1 = ax + by$. Therefore, $d = adx + bdy$. Let $x_0 = dx$ and $y_0 = dy$. Then $d = ax_0 + by_0$. Since d is a non-negative number, at least one of x_0 or y_0 must be non-negative. Let us assume that $x_0 \geq 0$ but $y_0 < 0$. Observe that for any x and y, if $d = ax + by$ then $d = a(x - b) + b(y - a)$, because $a(x - b) + b(y + a) = ax - ab + by + ab = ax + by = d$. In particular,

$$d = a(x_0 - b) + b(y_0 + a)$$

If $y_0 < 0$ it is not possible that $x_0 - b < 0$. To see this, suppose to the contrary that $x_0 - b < 0$. Then $x_0 < b$, implying that $x_0 \leq b - 1$. Also, since $y_0 < 0$, $y_0 \leq -1$. This would imply that

$$d = ax_0 + by_0 \leq a(b-1) + b(-1) = ab - a - b < ab - a - b + 1 = (a-1)(b-1)$$

which contradicts the premise that $d \geq (a-1)(b-1)$. Hence $x_0 - b \geq 0$. This shows that if we have a pair of numbers x and y such that $d = ax + by$ but that $y < 0$, we can find a second pair of numbers $x' = x - b$ and $y' = y + a$ such that $d = ax' + by'$ but for which $x' \geq 0$ and $y' > y$. This leads to the pair we need, as we now formalize.

Let $x_1 = x_0 - b$ and $y_1 = y_0 + a$. In general, let $x_{i+1} = x_i - b$ and $y_{i+1} = y_i + a$, for all $i \geq 0$. You should see that all pairs (x_i, y_i) satisfy $d = ax_i + by_i$. The sequence $y_0, y_1, y_2, \ldots$ is a strictly increasing sequence and so not all y_i are negative. Let k be the largest index such that y_k is negative. Then $y_{k+1} = y_k + a$ is non-negative. For this k, we have $d = ax_k + by_k$ with $y_k < 0$, so $x_{k+1} = x_k - b \geq 0$ by our preceding discussion. Therefore, we have found a pair, (x_{k+1}, y_{k+1}) such that $d = ax_{k+1} + by_{k+1}$ and both x_{k+1} and y_{k+1} are non-negative. □

1.1.4 Proofs

Proofs are important. When we make a claim that something is true, we have to prove that it is true. Although there are philosophical arguments about what constitutes a proof, we will restrict this discussion to methods of proof that no one disputes are valid. Three such methods of proof can be used to solve most problems:

1. proof by mathematical induction,
2. proof by contradiction, and
3. proof by counterexample.

We describe and give examples of each.

1.1.4.1 Proof by induction

Suppose a statement S can be formulated as depending on a single non-negative integer n. A proof of S by mathematical induction has two parts: a base case that establishes that $S(n)$ is true for a finite number of values of n, typically that it is true for $n = 0$, and an inductive argument that shows that if S is true for arbitrary k, then it is also true for $k + 1$. I.e., $S(k) \implies S(k + 1)$. We demonstrate this method by proving Equation 1.7 above using induction. We display the equation again for easy reference:

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$$

Base Case: n = 0 . The left side of the equation is 0 and the right hand side is also 0. Thus, the base case is true.

Inductive Hypothesis We assume the equality is true for $n = N$:

$$\sum_{k=0}^{N} k^2 = \frac{N(N+1)(2N+1)}{6}$$

Inductive Step We prove that it will be true when $n = N + 1$, given that it is true for $n = N$:

$$\begin{aligned}
\sum_{k=0}^{N+1} k^2 &= \sum_{k=0}^{N} k^2 + (N+1)^2 \\
&= \frac{N(N+1)(2N+1)}{6} + (N+1)^2 \\
&= (N+1)\left(\frac{N(2N+1)}{6} + \frac{6(N+1)}{6}\right) \\
&= (N+1)\left(\frac{2N^2+7N+6)}{6}\right) \\
&= (N+1)\left(\frac{(N+2)(2N+3)}{6}\right) \\
&= = (N+1)\left(\frac{(N+1+1)(2(N+1)+1)}{6}\right)
\end{aligned}$$

which shows that the equation is valid when $n = N + 1$.

1.1.4.2 Proof by Contradiction

In a proof by contradiction, the idea is to assume that the statement to be proved true is false and then show that the assumption that it is false leads to some contradiction. In symbolic language, let S be a statement to be proved true. Suppose that S implies that some statement R and its negation, $\sim R$ are both true. Then S implies (R and $\sim R$). Since (R and $\sim R$) must be logically false, S implies false, which by *Modus Tollens* implies that S is false. (*Modus Tollens states that if P implies Q and Q is false, then P is false.*)

Example 3. Proof that the square root of 2 is an irrational number.

Suppose that the square root of 2 is rational.

Then by the definition of a rational number, there are two integers p' and q' with $q' \neq 0$, such that $(p'/q')^2 = 2$.

This implies that there are two integers p and q that are relatively prime to each other such that $(p/q)^2 = 2$, because we can let $p = p'/\gcd(p', q')$ and $q = q'/\gcd(p', q')$ where gcd means "greatest common divisor".

Since $(p/q)^2 = 2$, $p^2 = 2q^2$ implying in turn that p^2 is even.

This means p is even because if it were odd, p^2 would be odd. So $p = 2m$ for some m, and $p^2 = 4m^2$. Hence $4m^2 = 2q^2$, implying $2m^2 = q^2$ implying q^2 is even which then implies q is even for the same reason that we stated above regarding p.

But then p and q are each even and cannot be relatively prime, which is a contradiction. So the supposition that square root of 2 is rational leads to a contradiction and therefore it must be false.

Example 4. This is another proof that the square root of 2 is an irrational number, but it is more intuitive; it is from Alexander Bogomolny. It is based on the *Fundamental Theorem of Arithmetic*, which states that every number is uniquely (up to the order of factors) representable as a product of primes. Assume the square root of 2 is rational and let $(p/q)^2 = 2$ for some integers p and q. Then $p^2 = 2q^2$. Factor both p and q into a product of primes. p^2 is factored into a product of the very same primes as p each taken twice. Therefore, p^2 has an even number of prime factors. So does q^2 for the same reason. Therefore, $2q^2$ has an odd number of prime factors. As $p^2 = 2q^2$, it cannot have both an even number of prime factors and an odd number of prime factors simultaneously, so this is a contradiction.

1.1.4.3 Proof by counterexample

There are two types of quantified statements: *universal statements* and *existence statements*. A universal statement is of the form, "For every such-and-such, P(such-and-such) is true" where P is some predicate involving such-and-suches. The symbol $\forall x$ means "for all x." For example,

"Every month has 31 days."

is a statement that says, "for every month x, 'has-thirty-one-days(x)' is true". This is clearly false and is easily proved by showing that there is some month that does not have 31 days, such as April. An existence statement is of the form, "There is a such-and-such for which Q(such-and-such) is true." This can be proved by finding a single such-and-such that makes Q true, but it is disproved by showing that for any possible such-and-such, Q(such-and-such) is false.

The proof that P is not universally true for every month is an example of a proof by counterexample. Proof by counterexample is used to prove that universal statements are false. It cannot be used to prove that universal statements are true.

Example 5. The *Fibonacci* numbers are defined recursively as follows:

$F_0 = 1$

$F_1 = 1$

$F_{n+2} = F_{n+1} + F_n$

F_k is the k^{th} Fibonacci number, which are important numbers that arise in unexpected places in nature and mathematics alike. Suppose we want to prove that the statement $(\forall k) F_k <= k^2$ is false. This is a universal statement, and to prove it false we need a witness to its falsehood, i.e., a counterexample. We can take $k = 11$: $F_{11} = 144 > 121$.

1.2 *C++* Review

This is a review of selected topics about *C++* that should have been covered in the prerequisite courses. They are here mostly for reference.

1.2.1 Functions with Default Arguments

C++, unlike *C*, allows any function to have default arguments for its parameters. A default argument is the value assigned to a formal parameter in the event that the calling function does not supply a value for that parameter.

Example 6. The declaration

```
void point2d(int = 3, int = 4);
```

declares a function that can be called with zero, one, or two arguments of type `int`. It can be called in any of these ways:

```
point2d(1,2);
point2d(1);
point2d();
```

The last two calls are equivalent to `point2d(1,4)` and `point2d(3,4)`, respectively.

The syntax for declaring default arguments is:

`return_type function_name (` $t_1\ p_1,\ t_2\ p_2,\ \ldots,\ t_k\ p_k = d_k,\ \ldots,\ t_n\ p_n = d_n$ `);`

where each t_k is a type symbol, p_k is a parameter symbol, and d_k is an initializing expression, which should be a constant literal or a constant global variable. It is not allowed to be a local variable. If parameter p_k has a default value, then all parameters p_i, with $i > k$ must also have default values. In other words, *trailing parameters*, the parameters to its right in the list of parameters must have default values also.

If a function is declared prior to its definition, as in a class interface, the defaults should not be repeated again – it is not necessary and will cause an error.

If default parameters are supplied and the function is called with fewer than n arguments, the arguments will be assigned in left to right order, as the `point2d()` example illustrated. As another example, given the function declaration,

```
void carryOn( int count, string name = "", int max = 100);
```

`carryOn(6)` is a legal call that is equivalent to `carryOn(6, "", 100)`, and `carryOn(6, "flip")` is equivalent to `carryOn(6, "flip", 100)`. If argument `k` is not supplied, then all arguments to the right of `k` must be omitted as well. The following are all invalid.

```
void bad1(int a = 2, int b    , int c);
void bad2(int a = 1, int b = 2, int c);
void bad3(int a    , int b = 3, int c);
```

The C++ standard also allows you to add default arguments to overloaded function declarations (not definitions) at a later time. The following program is valid code.

```
void f(int a, int b, int c);           // f() has no defaults
void f(int a, int b, int c = 1);       // c is given a default
void f(int a, int b = 1, int c);       // b is given a default
void f(int a = 1, int b, int c);       // a is given a default

int main()
{
   return 0;
}

void f( int a, int b, int c)
{
    //stuff here
}
```

Be warned though that there are not many good reasons to do this in your programs. The reason that default arguments are important is that they are particularly useful in reducing the number of separate constructor declarations within a class. One can write good programs without ever using default arguments for function parameters.

1.2.2 Member Initializer Lists

Consider the following class definitions.

```
class MySubClass
{
public:
    MySubClass(string s) { m_name = s; }
private:
    string m_name;
};

class X
{
public:
```

```
    X(int n, string str ): x(n), quirk(str) { }
private:
    const int x;
    MySubClass quirk;
    MySubClass& pmc;
};
```

The constructor

```
X(int n, string str ): x(n), quirk(str), pmc(quirk) { }
```

causes the constructors for the `int` class and the `MySubClass` class to be called prior to the body of the constructor, in the order in which the members *appear in the definition*[1], i.e., first `int` then `MySubClass`.

Member initializer lists are necessary in three circumstances:

- To initialize a constant member, such as `x` above,
- To initialize a member that does not have a default constructor, such as `quirk`, of type `MySubClass`,
- To initialize a reference member, such as `pmc` above. References are explained below.

1.2.3 Separation of Interfaces and Their Implementations

A class definition should always be placed in a ***header file*** (a `.h` file) and its implementation in an ***implementation file***, typically called a `.cpp` file. The header file should be thoroughly documented; it serves as a contract between the implementation of the class and the client code that uses it, and if you expect someone to use your class or read its interface, then you must describe in unambiguous, complete, and consistent language, what each member function and friend function does. This is nothing new.

If you would like to distribute this class to other users, you should distribute the thoroughly documented header file and the ***compiled implementation file***, i.e., ***object code***. The users do not need to see the source code for the implementation file. Of course, you can only do this if you know the target machine architecture and can compile for that architecture.

For a class, the implementation needs the interface, so you must put an `#include` directive in the `.cpp` file. Remember that the `#include` directive is executed by the preprocessor (*cpp*) and that it copies the included file into the position of the directive in a copy of the source file that it creates. You should always put a ***header guard*** into the header file. A header guard is a construction of the form

```
#ifndef __HEADERNAME_H
   #define __HEADERNAME_H

   the interface definitions here

#endif // __HEADERNAME_H
```

[1] Not the order in which they appear in the list!

where `__HEADERNAME_H` is a placeholder for a suitable name. Usually it is the name of the file in uppercase. It is used to prevent multiple inclusions of the same file, which would cause compiler errors. The first time it is encountered, the `ifndef` is true (because it is short for "if not defined"), the symbol `__HEADERNAME_H` is then defined, and the code is included. Subsequent attempts to include the file fail because the symbol is defined so `ifndef` is false.

For those wondering why we bother, if you do not do this, then multiple definitions of the same class will occur when the header file is included indirectly by other files, and this will cause a compile-time error.

Separating the interface and implementation is part of the process of organizing a program into distinct modules to make it easier to maintain the project, and for this reason it is important. It is better to create many small files than a few large ones, because changes are easier to control, compiling and relinking becomes faster, and debugging becomes easier.

Lastly, separation of interface and implementation is not just a good idea for classes, but for modules in general. If you are creating a program that has several utility functions that are needed by many other functions, but they are not really related to each other, you could create two files, `utilities.h` and `utilities.cpp` that have the function prototypes and their implementations respectively, and this will make the program easier to maintain.

1.2.4 Vectors and Strings

There have been some changes in the *C++* standard, *C++11*, that allow you to do things that you could not do before with vectors and with strings. These notes will not discuss the changes, but you are advised to review the textbook to see what they are. They are not of any importance. The only reason to be aware of them is so that you are not thrown for a loop, so to speak, when you see code that looks like

```
int sum = 0;
int squares[] = {0,1,4,9,16,25,36};
for ( int x : squares )
    sum += x;
```

or like this

```
int sum = 0;
int squares[] = {0,1,4,9,16,25,36};
for ( auto x : squares )
    sum += x;
```

1.3 *C++* Details

The assumption is that you know about pointers, but might need a reminder about a few things. Here it is.

1.3.1 Dynamic object creation

In *C++*, the `new` operator dynamically allocates memory on the heap and returns a pointer to the starting address of the created object, as in

```
myclass = new MyClass;
```

The `new` operator is overloaded to create arrays as well, as in

```
p = new int[100];
```

The old *C++* standard specified that, if it fails, it would return a `NULL` pointer. In the most recent standard, it will throw an exception that, unless it is handled, will terminate the program. Thus, if you plan on using *C++11*, you must catch the `std::bad_alloc` exception that `new` might throw, as in

```
#include <cstdio>
#include <new>
using namespace std;
int main()
{
    int sum = 0;
    try {
        int* p = new int[200000000000];
    }
    catch ( bad_alloc ) {
        printf(" too much memory requested \n" );
    }
    return 0;
}
```

The `delete` and `delete[]` operators free the storage associated with the pointer; the latter is used when the pointer is to an array. Assuming the objects are those created above, we release their resources with

```
delete myclass;
delete[] p;
```

If you fail to release memory when you are finished, you will be wasting memory, and guilty of causing ***memory leaks*** as in

```
while ( 1 ) {
    int* p = new int[1000000000];
    printf("la di da . The ship is leaking and I don\'t care.\n");
}
```

1.3.2 References Versus Pointers

The thing to remember is that a reference is another name for the same object, not an object containing its address[2]. Thus,

```
int x = 4;
int & y = x;      // y is another name for x, so y == 4
int z = y;        // z is not a reference
int* px = &x;     // px is a pointer to x;
z = 2;
int & m;          // illegal - m must be bound to an object when declared
y++;              // increments x
px++;             // increments px, which now points to something else.
```

1.3.3 Return Values

A function should generally return by value, as in

```
double sum(const vector<double> & a);
string reverse(const string & w);
```

`sum` returns a `double` and `reverse` returns a `string`. Returning a value requires constructing a temporary object in a part of memory that is not destroyed when the function terminates. If an object is large, like a class type or a large array, it may be better to return a reference or a constant reference to it, as in

```
const string & findmax(const vector<string> & a);
```

which searches through the string vector `a` for the largest string and returns a reference to it. It does not copy the string. Of course if the caller needs to modify it, then passing by `const` reference is not a solution. But in general, passing by reference can be error-prone – if the returned reference is to an object whose lifetime ends when the function terminates, the result is a runtime error. In particular, if you write

```
int& foo ( )
{
    int temp = 1;
    return temp;
}
```

then your function is returning a reference to `temp`, which is destroyed when the function terminates.

Usually you return a reference when you are implementing a member function of a class, and the reference is to a private data member of the class or to the object itself.

[2]This is a white lie. A reference *is* a pointer, but it is a special pointer that can be used with the same syntax as the thing that is pointed to. So if `y` is a reference to `x`, then `y` contains the address of `x`, but can only be used in the program as if it did not contain an address but was a substitute for the name `x`.

1.3.4 Constructors, Destructors, Copy Constructors, and Copy Assignment Constructors

1.3.4.1 Default Constructor

A ***default constructor*** is a constructor that can be called with no arguments. If a class does not have a user-supplied constructor of any kind, the compiler tries to generate a default constructor at compile time. If it has any kind of constructor, the compiler will not do this.

1.3.4.2 Destructor

A ***destructor*** is called when an object goes out of scope or is deleted explicitly by a call to delete. The compiler supplies destructors if the user does not. The reason to supply a destructor is to remove memory allocated by a call to `new`, to close open files, and so on. The destructor created by the compiler will be a ***shallow destructor***, meaning that it simply deletes the actual members of the class, and not any memory that members may point to, directly or indirectly.

1.3.4.3 Copy Constructor

A ***copy constructor*** is called in the following situations:

1. When an object needs to be constructed because of a declaration with initialization from another object of a type that can be converted to the object's class, as in

   ```
   IntCell C;
   IntCell B = C; // called here
   IntCell B(C);  // called here
   ```

   but not

   ```
   B = C;
   ```

   because `B` already has been constructed, so this is not a constructor call of any kind.

2. When an object is passed by value to a function.

3. When an object is returned by value from a function. If an object is returned by reference, then it is not copied. If by value, the object to be returned is constructed as a copy of the object within the function.

Again, *C++11* has made things more complex, by the inclusion of the ***move*** operator and ***call by rvalue-reference parameters***. It defines another type of constructor called a ***move constructor***. This is mentioned only briefly here. The move operation is what it sounds like – unlike an assignment such as `x = y`, which *copies* the value of `y` into `x`, the assignment `x = std::move(y)` does not perform a copy, but instead gives `x` the value stored in `y` and removes the value stored in `y`; it moves it from `y` to `x`.

1.3.4.4 Copy Assignment Operator

The copy assignment operator is called when two objects already exist and one is being assigned to the other. In *C++* it is `operator=`. Again, *C++11* has increased the complexity, as there are two different assignment operators, the ***copy assignment operator*** and the ***move assignment operator***. The copy assignment operator is called when the right hand side of the assignment is a lvalue, i.e., the name of an object. The move assignment operator is called in *C++11* when the right hand side is a temporary object that is about to be destroyed.

1.3.4.5 Using Defaults or Not

In general, you should either declare no destructors or constructors or assignment operators of any kind, or define all of them. If your data members include pointers, in general you should define all of these functions. Even if it does not include pointers, then whether or not you need to depends on whether any of the conditions described in Section 1.3.4.3 are true and you need to implement a copy constructor. *C++* does a great deal for you, but in turn you must understand its complex semantics if you are to avoid hard-to-diagnose bugs. This is why it is best to follow the simple rule of either all-or-nothing when it comes to these functions.

1.4 Templates

One difference between *C* and *C++* is that *C++* allows you to define templates for both classes and functions. It is easier to understand class templates if you first understand function templates.

Suppose that in the course of writing many, many programs, you find that you need a swap function here and there. Sometimes you want a swap function that can swap two integers, sometimes you want one that can swap two doubles, and sometimes you need to swap objects of a class. In *C*, you have to write a swap function for each type of object, or you can reduce the work by writing something like this[3]:

```
typedef int elementType;

void swap ( elementType *x, elementType *y)
{
    elementType temp = *x;
    *x = *y;
    *y = temp;
}
```

and you would call this with a call such as

```
int a, b;
a = ... ; b = ... ;
swap(&a, &b);
```

[3]There are other methods as well, but these are the two principal approaches.

In *C*, the parameters need to be pointers to the variables to be swapped, and their address must be passed. If you wanted to swap doubles, you would change the `typedef` by replacing the word "`int`" by "`double`."

In *C++*, you could do the same thing using reference parameters:

```
typedef int elementType;

void swap ( elementType &x, elementType &y)
{
    elementType temp = x;
    x = y;
    y = temp;
}
```

and you could call this with code such as

```
int a, b;
a = ... ; b = ... ;
swap(a, b);
```

Although you do not have to write a separate swap function for each different element type, it is inconvenient. The *C++* language introduced function templates as a way to avoid this.

1.4.1 Function Templates

A ***function template*** is a template for a function. It is not an actual function, but a template from which the compiler can create a function if and when it sees the need to create one. This will be clarified shortly. A template for the swap function would look like this:

```
template <class elementType>
void swap ( elementType &x, elementType &y)
{
    elementType temp = x;
    x = y;
    y = temp;
}
```

The word *class* in the template syntax has nothing to do with classes in the usual sense. It is just a synonym for the word *type*. All types in *C++* are classes. The syntax of a (single-parameter) function template definition is

```
template <class type_parameter> function-definition
```

where *`function-definition`* is replaced by the body of the function, as `swap()` above demonstrates. The syntax for a (single-parameter) function template declaration (i.e., prototype) is

```
template <class type_parameter > function-declaration
```

You need to repeat the line

```
template <class type_parameter>
```

before both the declaration and the definition. For example:

```
// Declare the function template prototype
template <class T>
void swap( T & x, T & y );

int main()
{
     int n= 5;
     int m= 8;
     char ch1 = 'a', ch2 = 'b';
     // more stuff here
     swap(n,m);
     swap(ch1, ch2);
     // ...
}

// Define the function template declared above
template <class T>
void swap( T & x, T & y )
{
    T temp = x;
    x = y;
    y = temp;
}
```

You will often see just the letter "T" used as the type parameter in the template.

When the compiler compiles the main program above, it sees the first call to a function named `swap`. It is at that point that it has to create an instance of a function from the template. It infers from the types of its parameters that the type of the template's parameter is `int`, and it creates a function from the template, replacing the type parameter by `int`. When it sees the next call, it creates a second instance whose type is `char`.

Because function templates are not functions, but just templates from which the compiler can create functions, there is a bit of a problem with projects that are in multiple files. If you want to put the function prototype in a header file and the function definition in a separate `.cpp` file, the compiler will not be able to compile code for it in the usual way if you use that function in a program. To demonstrate, suppose that we create a header file with our swap function prototype, an implementation file with the definition, and a main program that calls the function.

This is `swap.h`:

```
#ifndef SWAP_H
#define SWAP_H

template <class T>
void swap( T &x, T &y);
#endif
```

and `swap.cpp`:

```
template <class T>
void swap( T &x, T &y)
{
    T temp = x;
    x = y;
    y = temp;
}
```

and `main.cpp`:

```
#include "swap.h"
int main ()
{
    int a = 10, b = 5;
    swap(a,b);
    return 0;
}
```

When we run the command

```
g++ -o demo swap.cpp main.cpp
```

we will see the error message

```
/tmp/ccriQBJX.o: In function ‘main’:
main.cpp:(.text+0x29): undefined reference to ‘void swap<int>(int&, int&)’
collect2: ld returned 1 exit status
```

This is because the function named `swap` does not really exist when `main` is compiled. It has a reference only to a function template. The solution is to put the function template implementation into the header file, as unsatisfying as that is because it breaks the wall that separates interface and implementation. This can be accomplished with an `#include` directive:

```
#ifndef SWAP_H
#define SWAP_H
```

```
template <class T>
void swap( T &x, T &y);

#include "swap.cpp"
#endif
```

The general rule then, is to put the function template prototypes into the header file, and at the bottom, include the implementation files using an `#include` directive. There will be no problem with multiply-defined symbols in this case when you compile the code.

Note. Function templates, and templates in general, can have multiple parameters, and they do not have to be classes, but that is a topic beyond the scope of this introduction. You may also see the word `typename` used in place of the word `class` as the type of the template parameter. For the most part, these are interchangeable, but it is better to use `class` until you know the subtle difference. The interested reader can refer to a good C++ book for the details.

1.4.2 Class Templates

Imagine that you want to implement a list class. There is nothing in the description of a list that is specific to any particular type of data object, other than the ability to copy objects. For a sorted list, the objects do have to be comparable to each other in some linear ordering, but that is about the only limitation in terms of the data's specific properties. It stands to reason that you should be able to create a generic kind of list, one whose definition does not depend on the underlying element type. This is one reason that *C++* allows you to create templates for classes as well. A class template is like a generic description of that class that can be instantiated with different underlying data types.

1.4.2.1 Defining Class Templates

As with function templates, a *C++* class template is not a class, but a *template for a class*. An example of a simple class template interface is

```
template <class T>
class Container
{
public:
    Container();
    Container( T initial_data);
    void set( T new_data);
    T get() const;
private:
    T mydata;
};
```

Notice that a class template begins with the `template` keyword and template parameter list, after which the class definition looks the same as an ordinary class definition. The only difference is that it uses the type parameter `T` from the template's parameter list. The syntax for the implementations

of the class template member functions when they are outside of the interface is a bit more complex. The above functions would have to be defined as follows:

```
template <class T>
void Container<T>::set ( T initial_data )
{
    mydata = new_data;
}

template <class T>
T Container<T>::get() const
{
    return mydata);
}
```

Notice the following:

1. Each member function is actually a function template definition.
2. All references to the class are to `Container<T>` and not just `Container`. Thus, the name of each member function must be preceded by `Container<T>::`.

In general the syntax for creating a class template is

```
template <class T>  class class_name {  class_definition  };
```

and a member function named `foo` would have a definition of the form

```
template <class T>
return_type class_name<T>::foo ( parameter_list ) { function definition }
```

1.4.2.2 Declaring Objects

To declare an object of a class that has been defined by a template requires, in the simplest case, using a syntax of the form

```
class_name<typename> object_name;
```

as in

```
Container<int>    int_container;
Container<double> double_container;
```

If the `Container` class template had a constructor with a single parameters, the declarations would instead be something like

```
    Container<int>    int_container(1);
    Container<double> double_container(1.0);
```

The following is a complete listing of a very simple program that uses a class template.

Listing 1.1: A program using a simple class template.

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

template <class T>
class MyClass
{
  public:
      MyClass(  T initial_value);
      void set( T x) ;
      T get( )    ;

  private:
      T val;
};

template < class T >
MyClass < T >:: MyClass (T initial_value)
{
    val = initial_value;
}

template < class T >
void MyClass < T >:: set (T x)
{
    val = x;
}

template < class T >
T MyClass < T >::get ()
{
    return val;
}

int main ()
{
    MyClass<int>     intobj(0);
    MyClass<double>  floatobj(1.2);

    cout << "intobj value = " << intobj.get()
         << " and floatobj value = "  << floatobj.get() << endl;
    intobj.set(1000);
    floatobj.set (0.12345);
    cout << "intobj value = " << intobj.get()
         << " and floatobj value = "  << floatobj.get() << endl;

    return 0;
}
```

Again, remember that *a class template is not an actual definition of a class, but of a template for a class.* Therefore, if you put the implementation of the class member functions in a separate implementation file, which you should, then you must put an `#include` directive at the bottom of the header file of the class template, including that implementation file. In addition, make sure that you do not add the implementation file to the project or compile it together with the main program. For example, if `myclass.h`, `myclass.cpp`, and `main.cpp` comprise the program code, with `myclass.h` being of the form

```
#ifndef MYCLASS_H
#define MYCLASS_H

// stuff here

#include "myclass.cpp"
#endif // MYCLASS_H
```

and if `main.cpp` includes `myclass.h`, then the command to compile the program must be

```
g++ -o myprogram main.cpp
```

not

```
g++ -o myclass.cpp main.cpp
```

because the former will cause errors like

```
myclass.cpp:4:6: error: redefinition of 'void MyClass<T>::set(T)'
myclass.cc :4:6: error: 'void MyClass<T>::set(T)' previously declared here
```

This is because the compiler will compile the `.cpp` file twice! This is not a problem with function templates, but it is with classes, because classes are turned into objects.

1.4.3 Object and Comparable

The textbook makes use of `Object` and `Comparable` as generic types. The `Object` class represents any class with a default constructor, an `operator=`, and a copy constructor. The `Comparable` class is intended to represent any class that, in addition, has an `operator<`. When a class has an `operator<`, the collection of all of its instances is a totally ordered set; i.e., given any two `Comparables A` and `B`, either `A < B` or `B < A`.

1.4.4 Function Objects

A ***function object*** is a special type of object in *C++*. It is a class that contains a public overload of the ***function call operator***, `operator()`. A function object may be used instead of an ordinary function . In *C++* this is also called a ***class type functor***. An example or two will illustrate.

A simple function object can look like this:

```
class islessthan
{
public:
    bool operator()(const int &a, const int &b) const
    {
        return a < b;
    }
};
```

This is a class with nothing but the overloaded `operator()`. It has two parameters and compares their values and returns true if the first is less than the second, and false otherwise. It can be called like this:

```
int x = 10, y = 7;
if ( islessthan(x,y) )
    /* more stuff */
```

The call is indistinguishable from a call to a function named `islessthan`. We can pass a function object, not an instance of the function, but the class name itself, to a routine that expects a function pointer. Listing 1.2 demonstrates.

Listing 1.2: Function object example 1.

```
#include <iostream>
#include <iterator>
#include <vector>
#include <algorithm>

/* The function object is named islessthan */
class islessthan
{
public:
    bool operator()(const int &a, const int &b) const
    {
        return a < b;
    }
};


int main()
{
    std::vector<int> items;

    /* put numbers in reverse order into the vector */
    for ( int i = 10; i > 0; i--)
        items.push_back(i);
    /* The sort() algorithm from the standard algorithm library has the
       syntax
        void sort (RandomAccessIterator first,
                   RandomAccessIterator last,
                   Compare comp);
```

```
        where the first two parameters are iterators to the start and one
           past the end of the range of the container to be sorted, and
           the third parameter is a binary function that accepts two
           elements in the range as arguments, and returns a value
           convertible to bool. The third argument can be a function
           pointer or a function object.
    */

    std::sort(items.begin(), items.end(), islessthan());

    /* prove it is sorted by printing it out */
    for ( int i = 0; i < 10; i++)
        std::cout << items[i] << " ";
    std::cout << std::endl;

    return 0;
}
```

Function objects can also have data members to maintain their state. There is no restriction about this. They can also have a constructor to initialize the data members. This is a function object that retains its state and can be used like the ticker in a store that makes you take a number to be served:

```
class Ticker {
private:
    int &count;
public:
    Ticker(int &n) : count(n) {}
    int operator()()
    {
        return count++;
    }
};
```

The following listing shows how it could be used.

Listing 1.3: Function object with state.

```
#include <iostream>
#include <iterator>
#include <algorithm>

class Ticker {
private:
    int &count;
public:
    Ticker(int &n) : count(n) {}
    int operator()()
    {
        return count++;
    }
};
```

```
int main ()
{
    int numbers[20];
    int startvalue(10);

    std::generate_n (numbers, 20, Ticker(startvalue));

    std::cout << "Numbers given out today are:";
    for (int i=0; i<20; ++i)
        std::cout << ' ' << numbers[i];
    std::cout << '\n';

    return 0;
}
```

Chapter 2 Mathematical Concepts and Performance Measures

The main point of this chapter is to develop a system for evaluating and comparing ***algorithms***, not programs. For this reason, we do not measure performance differences between two implementations of the same algorithm, nor are we concerned with precise calculations of running times, since these depend on the implementation, the compiler, the operating system, and the machine architecture. Instead, we devise a means of evaluating the actual algorithm, independent of its implementation. We must work at a higher level of abstraction.

2.1 Mathematical Background

First, we define a method of measuring how "fast" functions of a single variable can grow. Use your intuition here. Consider the three non-decreasing functions

$f(x) = x^2$

$g(x) = x^3$

$h(x) = 5x^2$

Your intuition should tell you that as x gets larger and larger, the function $g(x)$ grows faster and faster than the others. Furthermore, for any value of x, $h(x)$ will always be exactly $5f(x)$. So whatever the rate of growth of $f(x)$, $h(x)$ is growing at the same rate as $f(x)$. They stay in a kind of lock step, with h and f proportionally the same as x marches towards infinity. On the other hand, as x increases, clearly $g(x)$ gets larger and larger than both $f(x)$ and $h(x)$. To see this, look at the first six integer cubes: 1, 8, 27, 64, 125, 216; whereas the first six integer squares are 1, 4, 9, 16, 25, 36. Even the first six values of $h(x)$ are overtaken by the faster growing $g(x)$: 5, 20, 45, 80, 125, 180.

The point is that whatever means we use to measure the relative rates of growth of functions, it ought to ignore constant factors such as the 5 above, and must rank functions like cubics as being faster of quadratics. The following definitions do just that.

2.2 Definitions of Asymptotic Rates of Growth

The word ***asymptotic*** is an adjective that means, "approaching a limit." In computer science, the limit is usually infinity, and the adjective is applied to the behavior of a function. ***Asymptotic analysis*** refers to the study of the limiting behavior of algorithms as their inputs approach infinity.

The three operators, "big O", "big omega Ω", and "theta Θ", define sets of functions. In other words, $O(f(n))$ is a set of functions (of one variable) that are related to f in some precise way, $\Omega(f(n))$ is a different set of function related to f. It is good to think of "big O", "big omega Ω", and "theta Θ" as operators that define sets of functions. In fact we will use the membership symbol $\in$ to indicate this.

The formal definitions of these operators are as follows.

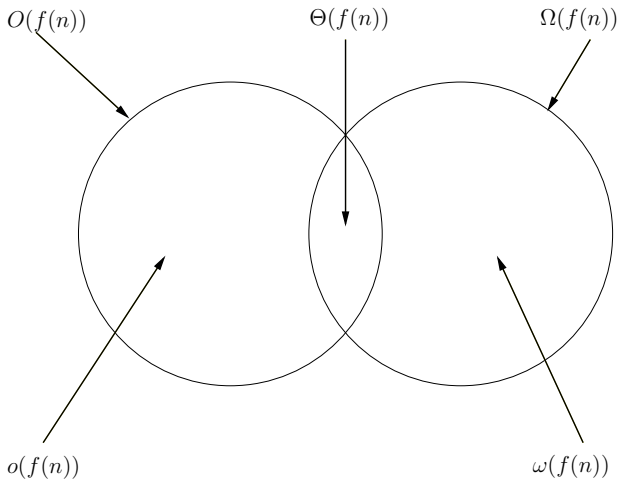


Figure 2.1: Relationships among the different classes of rates of growth.

Big O

$t(n) \in O(f(n))$ if there exist two numbers m and c such that $m \geq 0$ and $c \geq 0$ and $t(n) \leq cf(n)$ for all $n \geq m$.

In other words, $O(f(n))$ is the set of all functions $t(n)$ such that there is a constant c such that $t(n) \leq cf(n)$ for sufficiently large values of n. You could say that $O(f(n))$ is the set of functions that grow no faster than f.

Big Omega Ω

$t(n) \in \Omega(f(n))$ if there exist two numbers m and c such that $m \geq 0$ and $c > 0$ and $t(n) \geq cf(n)$ for all $n \geq m$.

In other words, Ω defines something like the opposite of big-O. The functions $t(n)$ in $\Omega(f(n))$ have the property that for each such $t(n)$ there is a c such that for sufficiently large n, $t(n) \geq cf(n)$. You could say that $\Omega(f(n))$ is the set of functions that grow no slower than f.

Theta Θ

$t(n) \in \Theta(f(n)$ iff $t(n) \in O(f(n))$ and $t(n) \in \Omega(f(n))$.

The set $\Theta(f(n))$ is actually the intersection of the first two sets. It consists of those functions that grow no faster than f and grow no slower than f. You could say that it is the set of functions that grow at the same rate as f. Remember to be careful with this. The functions $1000f(n)$ and $f(n)/10000$ are each in $\Theta(f(n))$.

Little o

$t(n) \in o(f(n)$ iff $t(n) \in O(f(n))$ and $t(n) \notin \Theta(f(n))$.

The set $o(f(n))$ is the intersection of $O(f(n))$ and the complement of $\Theta(f(n)$. It consists of functions that grow strictly slower than f. For example, $n \in o(n^2)$ but $2n^2 \notin o(n^2)$.

Little Omega ω

$t(n) \in \omega(f(n))$ iff $t(n) \in \Omega(f(n))$ and $t(n) \notin \Theta(f(n)$.

The set $\omega(f(n))$ is the intersection of $\Omega(f(n))$ and the complement of $\Theta(f(n)$. It consists of functions that grow strictly faster than f. For example, $n^3 \in \omega(n^2)$ but $2n^2 \notin \omega(n^2)$.

See Figure 2.1 for a visual depiction of the relationships among these different functions.

2.2.1 Some Implications Of The Definitions

Lemma 1. *If $t(n) \in O(f(n))$ and $s(n) \in O(g(n))$ then*

1. $s(n) + t(n) \in O(f(n) + g(n))$
2. $s(n) \cdot t(n) \in O(f(n) \cdot g(n))$

Proof. Because $t(n) \in O(f(n))$, there is an integer m and a constant c such that for all $n > m$, $t(n) \leq cf(n)$. Similarly, there are constants k and d such that $s(n) \leq gd(n)$ for all $n > k$. Let $C = \max(c, d)$ and let $M = \max(m, k)$. Then $s(n) + t(n) \leq Cf(n) + Cg(n) = C(f(n) + g(n))$ for all $n > M$, proving that $s(n) + t(n) \in O(f(n) + g(n))$.

Similarly, $s(n) \cdot t(n) \leq Cf(n) \cdot Cg(n) = C^2 f(n) \cdot g(n)$ for all $n > M$. In this case we use C^2 as the constant, and this shows that $s(n) \cdot t(n) \in O(f(n) \cdot g(n))$. □

Define

$$\max(f(n), g(n)) = \begin{cases} f(n) & if\ g(n) \in O(f(n)) \\ g(n) & if\ f(n) \in O(g(n)) \\ undefined & otherwise \end{cases}$$

In other words, $\max(f(n), g(n))$ is the faster growing function of $f(n)$ and $g(n)$. It is not hard to show that, if max is defined for f and g then $O(f(n) + g(n)) = O(\max(f(n), g(n)))$. The reason that it may not be defined is that one or the other of $f(n)$ and $g(n)$ may be ***oscillating***. See below for an example.

Calculus and the theory of limits can be used to determine the relative growth rates of functions. If necessary, ***L'Hopital's rule*** can be used for solving the limit:

if $\lim_{n\to\infty}(f(n)/g(n))=$	then all of these are true statements:
0	$f(n) \in O(g(n))$ and $f(n) \in o(g(n))$
$c \neq 0$	$f(n) \in \Theta(g(n))$
∞	$f(n) \in \Omega(g(n))$ and $f(n) \in \omega(g(n))$
oscillating	there is no limit and there is no relationship.

For an example of an oscillating fraction, let

$$f(n) = \begin{cases} n & \text{if } n \text{ is even} \\ 1 & \text{if } n \text{ is odd} \end{cases}$$

and let $g(n) = n$. Then the fraction $f(n)/g(n)$ alternates between 1 and $1/n$ as n approaches infinity, so there is no limit.

2.2.2 Some Growth Rate Relationships

In the following, the symbol < means that the function to the left is little-o of the function to the right.

$$c < \log N < \log^k N < N < N \log N < N^2 < N^3 < N^{3+k} < 2^N < 3^N < \ldots < N!$$

where the base of the log does not matter, and k is any integer greater than 1.

2.3 Model of Computation

When analyzing running time, we assume that the algorithm runs on some theoretical, abstract computer. It is usually a computer based on the ***Von Neumann architecture***. The Von Neumann architecture, named after John von Neumann[1], the mathematician who invented it, is one in which both the program and its data are stored in memory, and instructions are executed one after the other in sequence, fetching operands from memory. This is exactly the kind of computer you use today. Before Von Neumann, computers were single-purpose machines whose programs were hardware controlled. Other abstract models yield different running times.

Further assumptions about program execution are that

- all instructions take exactly the same amount of time
- memory is infinite
- all instructions are "simple" instructions that act on scalars, not vectors.

These assumptions are designed to simplify the analysis process without any loss of correctness. While it is true that some instructions take longer than others, for example, the difference is a constant factor that gets ignored anyway in the rate of growth analysis.

2.3.1 Input Size

Every input is assumed to have a positive integer size that depends on the particular problem to be solved. For example, the sorting problem sorts lists of things. The number of elements in the list is the size of the input, not the lengths of the items to be sorted, or their combined lengths. If the problem is searching for the occurrence of one string in a second string, the number of characters in the first string and the number of characters in the second string are the two input sizes. Different algorithms may depend on their sum, or product, or some other function of the two sizes. If the problem is to search for a keyword in a dictionary, the input size is not the length of the keyword, but the number of words in the dictionary.

[1] John von Neumann, "First Draft of a Report on the EDVAC", United States Army Ordinance Department and the University of Pennsylvania, 1945.

2.3.2 What to Analyze

The running time of an algorithm depends on the input it is given. For some inputs it might be fast and for others, it might be slow. We are usually interested in knowing the worst it can possibly do, because then we can plan conservatively. Sometimes though, we want to know the *average* running time. The idea of *average* is really useless because it treats all inputs as equally likely to occur, which is never true. What is more useful is the ***expected*** running time, which is the weighted average, i.e., each input is weighted by its probability of occurring. This is also a little silly since we rarely are able to weight the inputs realistically. Therefore, average running time is used, but everyone knows it is only an approximation. I will use the following notation the running times for an input of size N:

$T_{avg}(N)$ average (not expected value)

$T_{expected}(N)$ probabilistic analysis taking into account distribution of inputs

$T_{worst}(N)$ worst case

2.3.3 Running Time Calculations

These are rules for analyzing the running time of programs as representations of algorithms – the rules are designed to ignore details of implementation.

Remember that we usually ignore constant multiples and constant terms.

2.3.3.1 For Loops

The running time of a ***for loop*** is at most the running time of the statements inside the loop multiplied by the number of iterations of the loop.

2.3.3.2 Nested Loops

From the preceding statement, it follows that the running time of a statement inside ***nested loops*** is the running time of the statement multiplied by the product of the number of iterations of the loops. For example, the triply nested pseudo-code loop

```
for i = 1 to n
    for j = 1 to 2n
        for k = 1 to 3n
            S
```

runs in time proportional to the product of the running time of S and $6n^3$.

2.3.3.3 Consecutive Statements

The total running time of a sequence of statements is the sum of the running times of the statements. For example, the sequence

```
S1 ; S2 ; S3 ; ... ; Sn
```

has a running time equal to the sum of the running times of `S1`, `S2`, `S3`, ..., `Sn`. If order notation is being used to represent running time, then it is usually just the maximum of the running times of the individual statements that is reported.

2.3.3.4 If/else

The running time of an ***if/else*** statement

```
if (condition)
    S1
else
    S2
```

is at most the running time of the evaluation of the condition plus the maximum of the running times of `S1` and `S2`.

If we are interested in worst case analysis, then we have to assume maximums unless we can prove that certain statements are never reached.

2.3.3.5 Function Calls

If a sequence of statements contains function calls, the running time of the calls must be determined first. For recursive functions, the analysis can get difficult.

Example

```
long fib(int n)
{
    if (n <= 1)
        return 1;
    else
        return fib(n-1) + fib(n-2);
}
```

Let $T(n)$ be the running time of this function given input n. Then when the argument to the function is 0 or 1, the function returns without a recursive call, so the running time is a constant, which we choose to be 1:

$$T(0) = T(1) = 1$$

If the input is greater than 1, then two recursive calls are made and some constant work is done as well.

$$T(n) = T(n-1) + T(n-2) + 1$$

The constant 1 is for the evaluation of the condition and the execution of the `return` statement. This is a recurrence relation.

To solve this recurrence we make a few observations:

1. $T(n) \geq \text{Fib}(n)$ because it adds 1 each time.
2. $\text{Fib}(n) \geq (3/2)^n$ for all $n > 4$, which we now prove.

Proof. We prove this by induction on n.

Base Case: $\text{Fib}(5) = 8 > (3/2)^n = 7.59375$ proving it is true when $n = 5$.

Inductive Hypothesis: Assume it is true for arbitrary n. Then we have

$$\begin{aligned}
\text{Fib(n+1} &= \text{Fib (n)+Fib (n-1)} \\
&> (3/2)^n + (3/2)^{n-1} \\
&= (3/2)^{n-1}((3/2)+1) \\
&= (3/2)^{n-1} \cdot 5/2 \\
&> (3/2)^{n-1} \cdot 2.25 \\
&= (3/2)^{n-1} \cdot (3/2)^2 \\
&= (3/2)^{n+1}
\end{aligned}$$

which proves it is true for $n+1$. □

This shows that $T(n)$ grows exponentially. It doesn't matter how bad it is since that is bad enough. Recursion is not a very good way to compute Fibonacci numbers.

2.4 Example: The Maximum Subsequence Problem

This is a very nice problem to study for several reasons. One is that it shows how some cleverness can be used to replace a poorly performing solution by an extremely efficient one. Second, it gives us a chance to analyze three very different types of algorithms. Third, it illustrates an important concept in algorithm design, that whenever an algorithm computes a piece of information, it should try to reuse that information as much as possible. You shall see what this means soon.

Roughly stated, the maximal subsequence problem asks you to find a subsequence of a sequence of positive and negative numbers whose sum is the largest among all possible subsequences of the sequence. For example, given this sequence of numbers:

```
1, 2, -4, 1, 5, -10, 4, 1
```

you can verify that the subsequence 1, 5 has a sum of 6, and no other subsequence has a sum greater than 6.

Formally, given a sequence of possibly negative integers A_1, A_2, ..., A_N , find

$$\max_{i \le j} \left\{ 0, \sum_{k=i}^{j} A_k \right\}$$

For those unfamiliar with the notation, it means, find the maximum sum of each possible sequence of numbers from A_i to A_j for all pairs of indices i and j such that $i \le j$, and choose zero if all are negative.

Example. The sequence -2, 11, -4, 13, -5, -2 has a maximal subsequence whose sum is 20. The following table proves this. There is a row for each possible starting value, and a column for each possible length of sequence that starts at that value. The largest sum is 20, for the sequence of length 3 starting at 11.

	Length of Sequence					
Starting value of sequence	**1**	**2**	**3**	**4**	**5**	**6**
-2	-2	9	5	18	13	11
11	11	7	20	15	13	0
-4	-4	9	4	2	0	0
13	13	8	6	0	0	0
-5	-5	-7	0	0	0	0
-2	-2	0	0	0	0	0

The brute force method that is described later was used to make the table above.

2.4.1 Rules for Evaluating Running Time

1. Do not count the time to read input, since this is always $O(n)$, and it will hide the real running time of efficient algorithms.
2. Ignore constant multiples – use only order notation for rate of growth problems.

2.4.2 Solutions to Maximal Subsequence Problem

2.4.2.1 Brute Force

The following is a C++ function that solves this problem using brute force. It just checks all possible starting positions and all sequence lengths at that position, and adds the numbers in the subsequence:

```
int maxSubSeqSum( const  vector<int> & a)
{
    int max = 0;
    for (int i = 0; i < a.size(); i++)
        for (int j = i; j < a.size(); j++) {
            int sum = 0;
            for (int k =i ; k <= j; k++)
                sum += a[k];
            if ( max < sum)
            max = sum;
        }
    return max;
}
```

2.4.2.2 Analysis

The innermost loop executes $(j-i+1)$ times. This ranges over all j from i to N. Thus, the body of the loop is executed

$$\sum_{j=i}^{N-1}(j-i+1) = \sum_{j=0}^{N-1-i}(j+1) = \sum_{j=1}^{N-i} j = \frac{(N-i+1)(N-i)}{2}$$

times. The value of i ranges from 0 to $N-1$. Thus, the total number of executions is

$$\begin{aligned}
\sum_{i=0}^{N-1}\frac{(N-i+1)(N-i)}{2} &= \sum_{i=1}^{N}\frac{(N-(i-1)+1)(N-(i-1))}{2} \\
&= \sum_{i=1}^{N}\frac{(N-i+2)(N-i+1)}{2} \\
&= \sum_{i=1}^{N}\frac{N^2+3N+2-(2Ni+3i)+i^2}{2} \\
&= \frac{1}{2}\left(\sum_{i=1}^{N} N^2+3N+2\right) - \frac{2N+3}{2}\left(\sum_{i=1}^{N} i\right) + \frac{1}{2}\left(\sum_{i=1}^{N} i^2\right) \\
&= \frac{N(N^2+3N+2)}{2} - \frac{2N+3}{2}\left(\frac{N(N+1)}{2}\right) + \frac{N(N+1)(2N+1)}{12} \\
&= \frac{6N(N^2+3N+2)}{12} - \left(\frac{3N(N+1)(2N+3)}{12}\right) + \frac{N(N+1)(2N+1)}{12} \\
&= \frac{6N(N+1)(N+2)}{12} - \left(\frac{3N(N+1)(2N+3)}{12}\right) + \frac{N(N+1)(2N+1)}{12} \\
&= N(N+1)\left(\frac{6(N+2)-3(2N+3)+(2N+1)}{12}\right) \\
&= N(N+1)\left(\frac{2N+4}{12}\right) \\
&= \frac{N(N+1)(N+2)}{6} = \frac{N^3+2N^2+2N}{6}
\end{aligned}$$

which is $O(N^3)$. Why didn't I just use the nested loop rule? Because it was not clear that the loops were each $O(n)$, was it?

2.4.2.3 A Divide-and-Conquer, Recursive Solution

A more efficient solution can be obtained from the following observation:

A maximal subsequence is either entirely within the first half, or entirely within the second half, or it straddles the two halves. If it straddles the two halves, it can be broken into two pieces:

1. the sequence with the largest sum entirely within the first half that contains the last element of the first half.
2. the sequence with the largest sum entirely within the last half that contains the first element of the last half.

Thus, we can find the sequences in each half that satisfy these conditions, add them up and compare to the sequences found recursively in each half. We just take the max of all of them.

```
/**
Recursive maximum contiguous subsequence sum algorithm. Finds maximum sum
   in subarray spanning a[left..right]. Does not attempt to maintain
   actual best sequence.
*/
int maxSumRec( const vector<int> & a, int left, int right )
{
    if( left == right )  // Base cases
        if( a[ left ] > 0 )
            return a[ left ];
        else
            return 0;

    int center      = ( left + right ) / 2;
    /* Recursion here */
    int maxLeftSum  = maxSumRec( a, left, center );
    int maxRightSum = maxSumRec( a, center + 1, right );

    int maxLeftBorderSum = 0;
    int leftBorderSum    = 0;

    /* find the sum of every sequence ending at a[center]
       and starting at i, where i = center, center-1, center-2,...
       and save the maximum sum in MaxLeftBorderSum */
    for ( int i = center; i >= left; i-- ) {
        leftBorderSum += a[ i ];
        if ( leftBorderSum > maxLeftBorderSum )
            maxLeftBorderSum = leftBorderSum;
    }

    /* Do the analogous thing to the right-hand side of the center */
    int maxRightBorderSum = 0, rightBorderSum = 0;
    for ( int j = center + 1; j <= right; j++ ) {
        rightBorderSum += a[ j ];
```

```
        if ( rightBorderSum > maxRightBorderSum )
            maxRightBorderSum = rightBorderSum;
    }
        return max3( maxLeftSum, maxRightSum,
              maxLeftBorderSum + maxRightBorderSum );
}

/* Driver for divide-and-conquer maximum contiguous subsequence sum
   algorithm. */
int maxSubSum3( const vector<int> & a )
{
    return maxSumRec( a, 0, a.size( ) - 1 );
}
```

Analysis The algorithm makes two recursive calls with half size arrays. It also processes each element once in a pair of loops. The recurrence is

$T(1) = 1$

$T(N) = 2T(N/2) + cN$

If we "telescope" this recurrence relation, we get

$$\begin{aligned} T(N) &= 2(2T(N/4) + cN/2) + cN \\ &= 4T(N/4) + cN + cN \\ &= 4(2T(N/8) + cN/4) + 2cN \\ &= 8T(N/8) + 3cN \\ &= \ldots \\ &= 2^k T(N/2^k) + kcN \end{aligned} \tag{2.1}$$

The telescoping stops when $N/2^k = 1$ which occurs when $N = 2^k$ or when $k = \log_2 N$. Substituting $\log_2 N$ for k in Eq. 2.1, we get

$$\begin{aligned} T(N) &= N \cdot T(1) + \log_2 N \cdot cN \\ &= N + cN \cdot \log_2 N \\ &\in O(N + N \log N) \end{aligned}$$

This solution has a running time that is $O(N \log N)$ which beats $O(N^3)$ significantly! Can we do even better?

2.4.2.4 Linear Time Solution

A linear time solution to this problem is not hard to find. The previous solutions did not use information they discovered while examining the sequence. This is what I was talking about earlier. As the algorithm scans the string, it can learn so much more than it was doing, and avoid having to recompute or even re-examine previous partial sums.

Concept We take the liberty of letting the notation $a_i \ldots a_{j-1}$ mean both the sequence and its sum when the meaning is clear. The following assertions form the basis for the algorithm.

1. If a_i is negative, it cannot be the start of a maximal subsequence. (The sequence starting at a_{i+1}would be greater than the one starting at a_i if a_i is a negative number.)

2. More generally than that, any negative subsequence cannot be the start of a maximal subsequence. (By the same reasoning as in step 1.)

3. If $a_i \ldots a_{j-1}$ is positive but $a_i \ldots a_j$ is negative, then $a_i \ldots a_j$ cannot be the start of a maximal subsequence. Of course, by step 2, if $a_i \ldots a_j$ is negative, then $a_i \ldots a_j$ cannot be the start of a maximal subsequence, so what does this statement add to that? The next step is the key.

4. Suppose that j is the smallest index greater than i such that $a_i \ldots a_{j-1} \geq 0$ but $a_i \ldots a_j < 0$. In other words, for each index k, $i \leq k < j$, $a_i ... a_k \geq 0$. Consider any p such that $i < p < j$. The sum of the sequence $a_i \ldots a_{j-1}$ can be written as the sum of the numbers from a_i to $a_{p-1}$and the sum of the numbers from a_p through a_{j-1}:

   $$a_i \ldots a_{j-1} = a_i ... a_{p-1} + a_p \ldots a_{j-1} \tag{2.2}$$

   Since we said that for any k, $i \leq k < j$, $a_i ... a_k \geq 0$, it is true for $k = p - 1$,so $a_i \ldots a_{p-1} \geq 0$. Eq. 2.2 is of the form

   $$a_i \ldots a_{j-1} = X + a_p \ldots a_{j-1}$$

   where X a non-negative number, so it follows that if we subtract it from the right-hand side, the right-hand side stays the same or gets smaller:

   $$a_i \ldots a_{j-1} \geq a_p \ldots a_{j-1}$$

   In other words, $a_i \ldots a_{j-1}$ is greater than any of its suffix subsequences. In particular, when we append a_j to both sides of the inequality, it still holds:

   $$a_i \ldots a_j \geq a_p \ldots a_j$$

   and since the left hand side is negative, so is the right hand side.

5. This implies that if we have some initial subsequence $a_i \ldots a_{j-1}$ that is positive or zero, and we encounter an a_j that makes the sum negative, we can advance the start index i to the next position after j, i.e., $j + 1$ and start looking for a new maximum subsequence there, because $a_i \ldots a_j$ cannot be the start of a maximum subsequence.

This leads to the following very simple algorithm.

```
int maxSubSum4( const vector<int> & a )
{
    int maxSum = 0, thisSum = 0;
    for ( int j = 0; j < a.size( ); j++ ) {
        thisSum += a[ j ];
        if ( thisSum > maxSum )
            maxSum = thisSum;
        else if ( thisSum < 0 )
            // this a[j] made the initial sequence negative - start over
            thisSum = 0;
    }
    return maxSum;
}
```

Analysis It is clearly a linear algorithm since it is a single loop that iterates over every element of the sequence.

You may be tempted to ask whether there exists an even faster solution. If there is not, then we would say that this solution is ***optimal***, meaning the best it can be, because it runs as fast as is possible. But if there does exist a faster solution, this one would not be considered optimal. Can there be a solution that runs in $o(N)$, meaning in less than linear time? Intuitively it is impossible, because any solution must look at each number at least once, and so any solution must run in at least $\Theta(N)$ time.

Chapter 4 Trees

Preliminaries

Tree definitions

If you already know what a binary tree is, but not a general tree, then pay close attention, because binary trees are not just the special case of general trees with degree two. I use the definition of a tree from the textbook, but bear in mind that other definitions are possible.

Definition. A **tree** T consists of a (possible empty) set of nodes. If it is not empty, it consists of a distinguished node *r* called the root of T and zero or more non-empty subtrees T_1, T_2, …, T_k such that there is a directed edge from *r* to each of the roots of T_1, T_2, …, T_k.

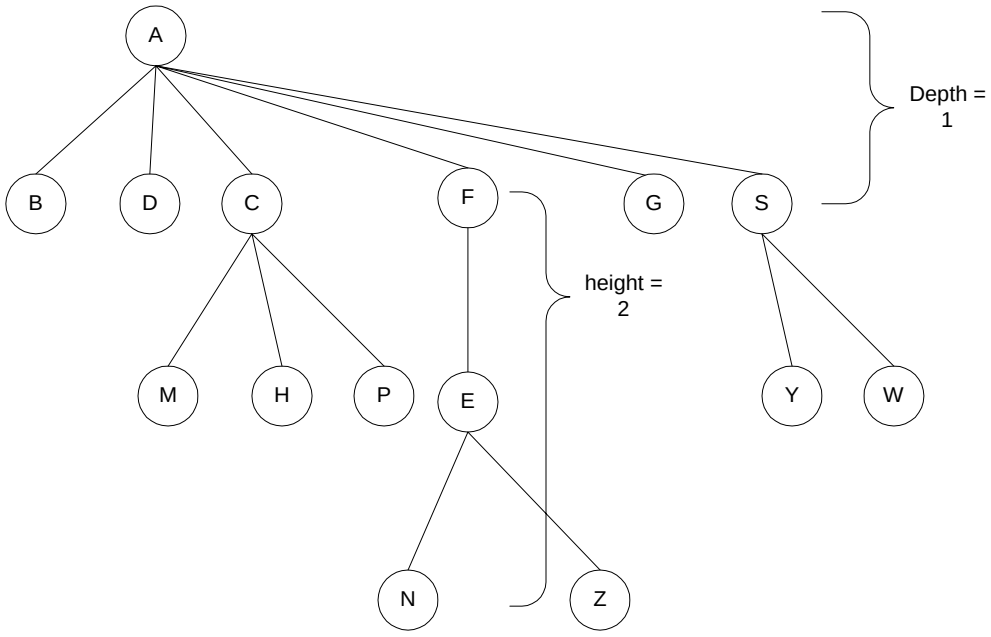
Definition. A **forest** is a collection of non-empty trees. You can always create a tree from a forest by creating a new root node and making it the parent of the roots of all of the trees in the forest. Conversely, if you lop off the root of a tree, what is left is a forest.

I assume that you are familiar with the terminology of binary trees, e.g., *parents*, *children*, *siblings*, *ancestors*, *descendants*, *grandparents*, *leaf nodes, internal nodes*, *external nodes*, and so on, so I will not repeat their definitions here. Because the definitions of height and depth may vary from one book to another, I do include their definitions here, using the ones from the textbook.

Definition. A **path** from node n_1 to node n_k is a sequence of nodes n_1, n_2, …, n_k such that n_i is the parent of n_{i+1} for $1 \leq i < k$. The **length** of a path is the *number of edges* in the path, *not* the number of nodes.

Because the edges in a tree are directed, all paths are "downward", i.e., towards leaves and away from the root. The **height** of a node is the length of the longest path from the node to any of its descendants. Naturally the longest path must be to a leaf node. The **depth** of a node is the length of the path from the root to the node. The root has depth 0. All leaf nodes have height 0.

The **height** of a tree is the height of its root. The **degree of a node** is the number of children of the node. The **degree of a tree** is the maximum degree of the degrees of its nodes. The tree on the next page has height 3 and degree 6.



It is not hard to see that a tree with N nodes must have N-1 edges because every node except the root has exactly one incoming edge. *How many edges are in a forest with N nodes and K trees?*

Applications of General Trees

A general tree is useful for representing hierarchies in which the number of children varies.

- In a file system, a node represents each file, and if the file is a directory, then it is an internal node whose children are the files contained in the directory. Some file systems do not restrict the number of files per folder, implying that the number of children per node is varying and unbounded.
- In computational linguistics, as sentences are parsed, the parser creates a representation of the sentence as a tree whose nodes represent grammatical elements such as predicates, subjects, prepositional phrases, and so on. Some elements such as subject elements are always internal nodes because they are made up of simpler elements such as nouns and articles. Others are always leaf nodes, such as nouns. The number of children of the internal nodes is unbounded and varying.
- In genealogical software, the tree of descendents of a given person is a general tree because the number of children of a given person is not fixed.

Tree implementations

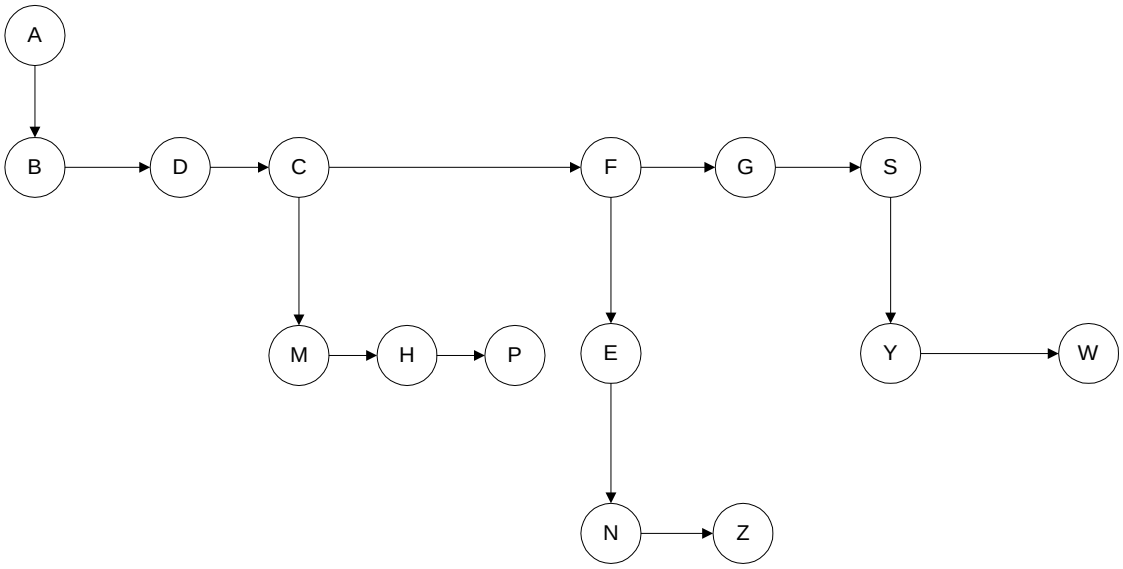
Because one does not know the maximum degree that a tree may have, and because it is inefficient to create a structure with a very large fixed number of child entries, the best implementation of a general tree uses a linked list of siblings and a single child, as follows.

General tree of unknown degree

A tree node is an element and a left-child and next-sibling pointer:

```
struct  TreeNode
{
      Object      element;
      TreeNode *  firstChild;
      TreeNode * nextSibling;
};
```

The following sibling tree would represent the above tree.



General Tree Traversal: An Example.
An algorithm to print the files and folders in a directory hierarchy:

```
void  listAll( int depth = 0) const
{
      printname(depth);
      if (isDirectory() )
      for each child c in this directory
                  c.listAll(depth + 1);
}
```

The above algorithm is a pre-order traversal because it visits the root of every subtree prior to visiting any of the children, and when it visits the children, it does so in a "left-to-right" order. A general tree has no notion of left and right, but the algorithm creates one. If `printname()` prints a word with depth tabs preceding it, then this will print an indented listing of the directory tree, with files at depth d having d tabs to their left.

One can also define post-order traversals of general trees, but there is no single notion of in-order traversal because of the fact that the number of subtrees varies from one node to the next and the root may be visited in many positions relative to its children.

Binary Trees

A binary tree is not a general tree because in binary trees the left and right subtrees are distinguished from each other. A binary tree is either empty or has a root node and a left and right subtree, each of which are binary trees, and whose roots are children of the root of the tree. The book uses the following implementation of a binary tree node.

```
struct BinaryNode
{
            Object  element;
            BinaryNode *left;
            BinaryNode *right;
};
```

The most important applications of binary trees are in compiler design and in search structures. The binary search tree is an important data structure. I assume that you are familiar with them from the previous course. The goal here is to look at how they can be implemented using C++.

The Search Tree ADT-Binary Search Trees.

Binary search trees are binary trees that store comparable elements in such a way that insertions, deletions, and find operations never require more than O(h) operations, where h is the height of the tree. Formally,

Definition. (S, R) is an ordered set if S is a set and R is a binary relation on S.

Definition. If x is a node, element(x) is the element contained in node x.

Definition. A binary search tree for the ordered set (S,"<") is a binary tree in which for each node x, element(x) $\in$ S, and for every node z in the left subtree of x, element(z) < element(x), and for every node z in the right subtree of x, element(x) < element(z).

It is important to realize that binary search trees use an implicit ordering relation to organize their data, and that a comparison operation must always be defined for this data.

A binary search tree must support insertion, deletion, find, a test for emptiness, and a find-minimum operation. It would also be useful to support a list-all operation that lists all elements in sorted order. Since it is a container (an object that contains other objects), it needs to provide methods to create an empty instance of a binary search tree, or to make a new tree as a copy of an existing tree, and to destroy instances of trees.

Algorithms

Most tree algorithms are expressed easily using recursion. On the other hand, to do so requires that the parameter of the algorithm is a pointer to the root of the current node, but this is a problem, because one does not want to expose to the object's clients the node pointers. Nodes should be hidden from the object's clients. Put another way, the clients should not know and not care how the data is structured inside the tree; in fact, they should even know that it *is* a tree!

The binary search tree should be a black box with hooks to the methods it makes available to its clients, and no more.

Therefore, the sensible solution is to create a "wrapper" method that the client calls that wraps a call to a recursive function. This is exactly how the find, insert, and remove functions work. Below are their implementations. I skip the implementations of everything else. You should not skip them though; you have to make sure you understand how this class implementation works!

find

The find algorithm is a recursive algorithm. If the current node is empty, return an indication that it is not in the tree, otherwise if the element is smaller than the current node's element, look in the left subtree otherwise if it is larger look in the right subtree, otherwise it is equal.

insert

Insertion is similar. If the current node is empty, we reached the point where the element is supposed to be but it is not there, so insert it there. Otherwise, if the element to be inserted is smaller than the current one, insert in the left, otherwise, if it is larger, insert it in the right, otherwise it is equal and thus should not be inserted.

remove

The deletion algorithm is the most complex. It depends on how many children the node to be deleted has. If no children, it is easy: just delete the node. If it has one child, delete the node and make the only child the child of the node's parent. If it has two children, then find the smallest node in the node's right subtree and copy it into the node, effectively deleting that element. Then delete the node that it came from. That node cannot possibly have two children because if it did, one would have to be smaller than the node, contradicting the assumption that it was the smallest node in the right subtree.

Implementation of a Binary Search Tree Node

The book implements binary search trees using some fairly clever coding techniques. Some of the techniques are hard to follow. In these notes I clarify the more difficult or subtle aspects of the implementation and omit discussion of the rest. The class interface for a binary search tree node follows. Notice that its constructor is private but that the `BinarySearchTree` class is a friend. The only objects that can create nodes are tree objects.

```
template <class Comparable>
class BinarySearchTree;

template <class Comparable>
class BinaryNode
{
        Comparable element;
        BinaryNode *left;
        BinaryNode *right;
        BinaryNode( const Comparable & theElement, BinaryNode *lt, BinaryNode *rt )
          : element( theElement ), left( lt ), right( rt ) { }
        friend class BinarySearchTree<Comparable>;
};
```

The Class Interface

Part of the BinarySearchTree interface follows.

```
template <class Comparable>
class BinarySearchTree
{
  public:
    explicit BinarySearchTree( const Comparable & notFound );
    BinarySearchTree( const BinarySearchTree & rhs );
    ~BinarySearchTree( );
    const Comparable & findMin( ) const;
    const Comparable & find( const Comparable & x ) const;
    bool isEmpty( ) const;
    void insert( const Comparable & x );
    void remove( const Comparable & x );
    const BinarySearchTree & operator=( const BinarySearchTree & rhs );

  private:
    BinaryNode<Comparable> *root;
    const Comparable ITEM_NOT_FOUND;
    const Comparable & elementAt( BinaryNode<Comparable> *t ) const;
    void insert( const Comparable & x, BinaryNode<Comparable> * & t ) const;
    void remove( const Comparable & x, BinaryNode<Comparable> * & t ) const;
    BinaryNode<Comparable> * findMin( BinaryNode<Comparable> *t ) const;
    BinaryNode<Comparable> *
              find( const Comparable & x, BinaryNode<Comparable> *t ) const;
    BinaryNode<Comparable> * clone( BinaryNode<Comparable> *t ) const;
};
```

Algorithm Implementations

For the algorithms to work as prescribed, the private member functions return a pointer to a binary node, but the corresponding public member functions return the element contained in the node. To make this happen, the author uses the *elementAt()* function, defined below.

```
template <class Comparable>
const Comparable & BinarySearchTree<Comparable>::
elementAt( BinaryNode<Comparable> *t ) const
{
   return t == NULL ? ITEM_NOT_FOUND : t->element;
}
```

Notice that it returns the constant Comparable ITEM_NOT_FOUND if the pointer is NULL, and the element in the node if the pointer is not NULL.

Here is the public wrapper for the find method.

```
template <class Comparable>
const Comparable & BinarySearchTree<Comparable>::find( const Comparable & x ) const
{
   return elementAt( find( x, root ) );
}
```

Here is the private find member function that it calls:

```
template <class Comparable>
BinaryNode<Comparable> *BinarySearchTree<Comparable>:: find( const Comparable &  x,
                                    BinaryNode<Comparable> *t ) const
 {
   if( t == NULL )
       return NULL;
   else if( x < t->element )
        return find( x, t->left );
   else if( t->element < x )
        return find( x, t->right );
   else
        return t;    // Match
}
```

Insertion is similar:

```
template <class Comparable>
void BinarySearchTree<Comparable>::insert( const Comparable & x )
{
   insert( x, root );
}
```

```
void BinarySearchTree<Comparable>::
insert( const Comparable & x, BinaryNode<Comparable> * & t ) const
{
   if( t == NULL )
      t = new BinaryNode<Comparable>( x, NULL, NULL );
   else if( x < t->element )
      insert( x, t->left );
   else if( t->element < x )
      insert( x, t->right );
   else
      ;  // Duplicate; do nothing
}
```

Deletion is the same idea, but the algorithm is different, as I described above. The deletion public member calls the private remove member, copied below. The remove function calls a *findMin()* function to find the minimum element in a given subtree. *findMin()* has a public and a private recursive version.

```
template <class Comparable>
const Comparable & BinarySearchTree<Comparable>::findMin( ) const
{
   return elementAt( findMin( root ) );
}
template <class Comparable>
BinaryNode<Comparable> *
BinarySearchTree<Comparable>::findMin( BinaryNode<Comparable> *t ) const
{
   if( t == NULL )
      return NULL;
   if( t->left == NULL )
      return t;
   return findMin( t->left );
}

template <class Comparable>
void BinarySearchTree<Comparable>::
remove( const Comparable & x, BinaryNode<Comparable> * & t ) const
{
   if( t == NULL )
      return;   // Item not found; do nothing
   if( x < t->element )
      remove( x, t->left );
   else if( t->element < x )
      remove( x, t->right );
```

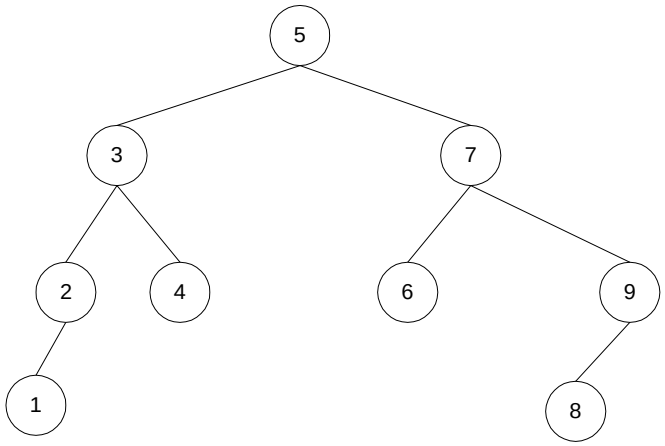
```
    else if( t->left != NULL && t->right != NULL ) // Two children
    {
       t->element = findMin( t->right )->element;
       remove( t->element, t->right );
    }
    else
    {
       BinaryNode<Comparable> *oldNode = t;
       t = ( t->left != NULL ) ? t->left : t->right;
       delete oldNode;
    }
}
```

Examples of search trees.

This sequence of insertions:

5 3 7 2 9 8 4 6 1

results in this tree of height 3.



In contrast,

9 8 7 6 5 4 3 2 1

produces a tree of height 8. The order in which the keys are inserted determines the height of the tree.

Performance Analysis

The insertion, deletion, and search algorithms all take a number of steps that is proportional to the height of the tree in the worst case. Deletion may involve a larger constant than insertion and search because of the extra steps involved, but it still does not visit more nodes than the length of the longest path in the tree. The running time of these algorithms is therefore dependent on the depth of the tree. Hence the question, *what is the expected height of a binary search tree?*

The preceding example shows that the order of insertions affects tree height and that in the worst case the height is O(N). But what about the average height? To clarify the picture, we can assume that keys are positive integers. Since the actual numeric difference between successive keys does not affect the shape of the tree, we might as well assume that a tree with N nodes consists of the integers 1 through N. In other words, we get the same tree with the sequence 1,2,3 as we do with 1, 20, 400 or 2, 23, 800, so we assume that the sequence is 1,2,3,…,N.

If the first key to be inserted is the number i, with 1 <= i <= N, then this determines how many nodes are in the left and right subtrees: there must be (i-1) nodes in the left subtree and (N-i) nodes in the right subtree. This is true of every subtree of the tree.

Definition. The **internal path length** of a tree is the sum of the depths of all nodes in the tree.

Let D(N) denote the expected internal path length of some arbitrary tree T with N nodes. Suppose that the root is the number i. Then the left subtree has (i-1) nodes and it has expected internal path D(i-1). To get from the root to any node in the left subtree requires traversing one extra edge, so the total expected path length of the left subtree starting at the root is D(i-1) + (i-1). Similarly, the right subtree has an expected path length of D(N-i) + (N-i). This leads to the recurrence:

$$D(1) = 0$$
$$D(N) = D(i-1) + D(N-i) + (i-1) + (N-i) = D(i-1) + D(N-i) + N-1$$

If all sequences 1,2,…,N are equally likely then there is a uniform 1/N probability that the first number will be i . In other words, the root may be any of 1,2,3,…,N with equal probability, and hence, the average height is

$$\frac{1}{N}\left[(D(0)+D(N-1))+(D(1)+D(N-2))+\cdots+(D(N-2)+D(1))+(D(0)+D(N-1))\right]+N-1$$
$$=\frac{2}{N}\left[\sum_{i=0}^{N-1} D(i)\right]+N-1$$

It will be shown later that D(N) is O(N log N). Therefore the average binary search tree has a total path length of O(N log N), implying that the average path is length O(log N).

This shows that on average, insertions, deletions, and finds are O(log N) operations if the trees are constructed randomly from N keys. But if trees are subjected to deletions, then their shapes will change because the deletion algorithm favors one side or the other. To avoid this, the deletion algorithm could alternate between choosing the smallest element in the right subtree and the largest in the left subtree. If this is done, it has been shown, then the expected depth will be $\Theta\left(\sqrt{N}\right)$.

A more interesting solution to the problem was invented by Tarjan in the early 1980s and the technique was subsequently generalized to other problems. He invented self-adjusting trees, which after each operation restructured the tree to make future operations more efficient.

Balanced Binary Search Trees

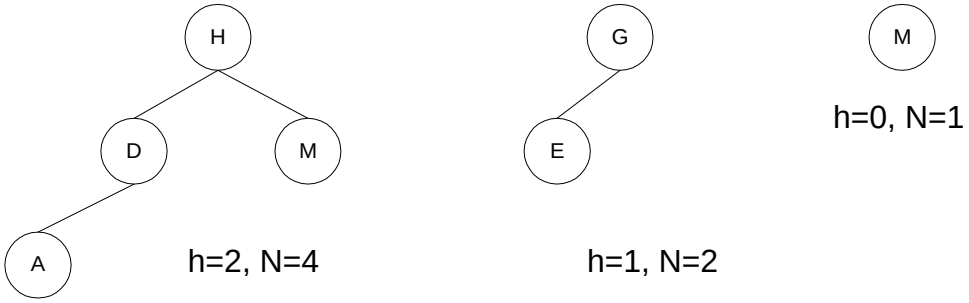
We can define a perfectly balanced binary search tree with N nodes to be a complete binary search tree, one in which every level except the last is completely full. A perfectly balanced BST would have guaranteed O(log N) search time. Unfortunately it is too costly to rebalance such a tree after each insertion and deletion. We need a tree with a weaker balancing condition.

AVL Trees

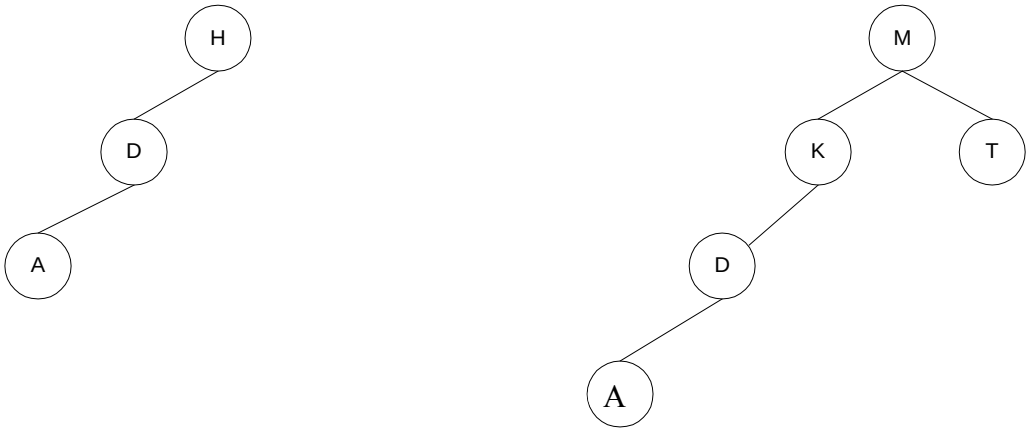
An AVL tree (AVL for the names of its inventors, Adelson-Velskii and Landis) is a binary search tree with a balancing condition that is relatively inexpensive to maintain.

Definition. An AVL tree is a binary search tree in which, for every node in the tree, the heights of the left and right subtrees of the node differ by at most 1.

Examples



Each of these trees satisfies the balancing condition. The following trees are not AVL trees



AVL Tree Search

A search takes O(h) time. Later we will see that the height of a tree is at worst O(log n), so that searching takes O(log n) time.

AVL Tree Insertion

An insertion increases the height of some subtree by 1. Consider traveling up the tree from the point of insertion, inspecting the balance condition at each node. The first node found that is out of balance can be out of balance in one of 4 ways. An analysis shows that of the 4 different cases, two are symmetric. There are therefore 2 theoretical cases and their symmetric counterparts. Let A be the node of greatest depth that is out of balance.

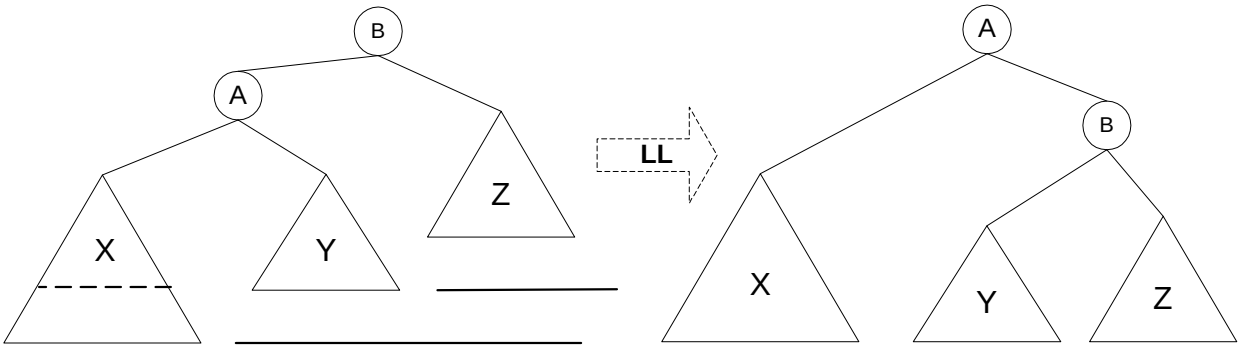
1. The insertion was in the left subtree of the left child of A
2. The insertion was in the right subtree of the left child of A
3. The insertion was in the left subtree of the right child of A
4. The insertion was in the right subtree of the right child of A

Cases 1 and 4 are symmetric and are resolved with an LL or RR rebalancing.

Cases 2 and 3 are symmetric and are resolved with an LR or RL rebalancing.

Single Rotation

This is the LL rotation. The RR is symmetric.



LL and RR are single rotations. RL and LR are double rotations. Single rotations are easier to understand and implement. Double rotations require slightly more work.
The single rotation does not increase the height of the tree rooted at the unbalanced node. To see this you have to observe the following facts. Let h be the height of tree X after the insertion. The tree rooted at B has height h+2.

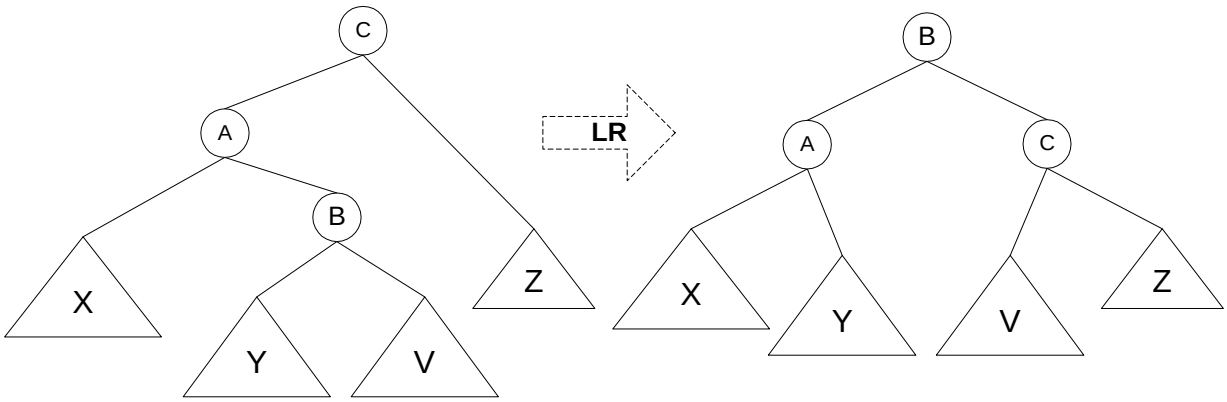
1. Since all nodes below B are in balance, the tree rooted at A is in balance,
   height(X) – height(Y) < 2

If height(Y) = height(X) after the insertion, then before the insertion, the tree rooted at B would not have been an AVL tree, since the left subtree of B would have had the same height that it has after the insertion, and it is not an AVL tree now. Therefore, height(X) = height(Y) +1.

2. Since the tree is not an AVL tree now, height(X) - height(Z) >= 2. If height(X) - height(Z) > 2, then it would not have been an AVL tree before the insertion, so height(X) - height(Z) = 2.
3. It follows that height(Y) = h-1, height(Z) = h-1.
4. After the rotation, the left subtree of A has height (h+1) and the right subtree has height (h-1+2) = (h+1). Thus the tree has height h+1, which is the height that the tree was prior to the insertion.

This shows that the subtree rooted at B is restored to the height it had prior to the insertion, which implies that no further rebalancing is necessary.

Double Rotation



The double rotation is required because the single rotation would not rebalance in this case. Try it. To do the double rotation, travel up the tree from the point of insertion until you reach a node that is no longer in balance. Call this node C. The right subtree of the left child of C has grown in height by 1. We don't know whether it is the left or right subtree of this tree, but it does not matter. The diagram shows the rotation. Let the root of the right child of the left child of C be B. To do the double LR rotation, make B the new root, make its parent A its left child and make the old root, C, the right child of B. Because B is now the root and it has two children, it must give up its old subtrees for adoption. It gives Y to A and V to C, as illustrated.

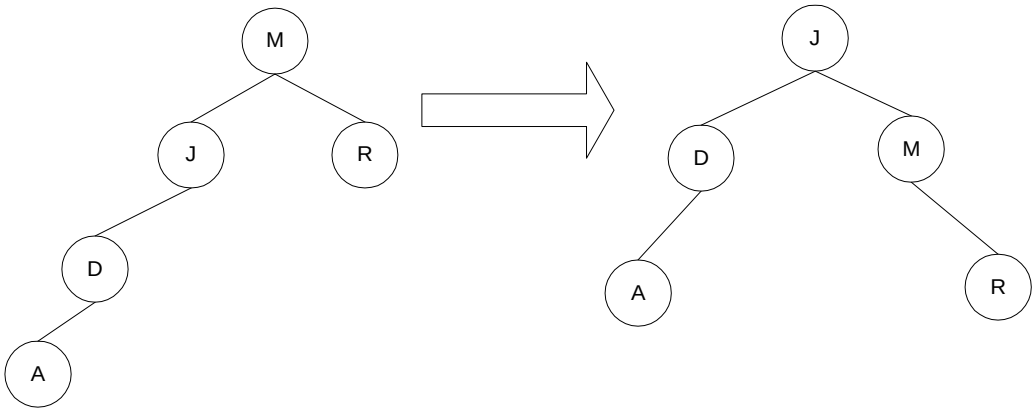
The rotation reduces the height of the tree by 1 for similar reasons as above. If h is the height of Y (or V) after the insertion, then height(X) must be h also. Height(Z) must be h+1. It cannot be less. Why? After the rebalancing, the tree rooted at B has height h+2, which is the height it has prior to insertion.

Analysis of Insertion Time

If no rotation is performed, the insertion takes O(h) steps. If a rotation is performed, regardless of which it is, the height of the rebalanced subtree is reduced to what it was prior to the insertion. (So then how can a tree grow in height?)

This implies that the height of the tree after rebalancing is the height that it was before the insertion. Therefore, since the tree was an AVL tree before the insertion, all ancestor nodes were in balance and they remain in balance after the insertion and rebalance. This implies that once the rebalancing has been done, the insertion is complete. Rebalance takes constant time; therefore insertion takes O(h) steps for the search plus a constant for the rebalancing.

Example



AVL Tree Deletion

Deletion from an AVL tree is more complicated than insertion. First of all, the basic deletion algorithm is the same as that of an unbalanced BST: if a node is a leaf, delete it; if it has one subtree, make the subtree the child of the node's parent, and if it has two subtrees, replace it with the in-order successor or predecessor and delete that node.

Having deleted the node, the height of some subtree has been reduced by 1. This reduction in height might have caused some node on a path from that node up to the root to lose its balance condition. Let B be the first node on the path back to the root whose balance condition is now violated.

It doesn't matter that the condition was caused by a deletion – one of the four rebalancing algorithms will rebalance the tree that is out of balance. The question is, how do we know which to do? The wrong choice will leave the tree unbalanced. I will come back to this soon.

The other problem is that rebalancing *may* make the height of the tree smaller. (After an insertion, rebalancing *always* restores the tree to its height before the insertion, but after a deletion, it *may not change the height at all*!) Unlike the insertion problem, where the height had increased and is now restored to what it was before, in deletion, the height was diminished, and now rebalancing may diminish it again. This implies that the parent of the node whose tree was

just rebalanced may now have an imbalance condition, requiring that its tree be rebalanced, and then its parent might need it, and so on until we reach the root. When we finally get to the root, we rebalance and the process stops there. The time for a deletion is therefore O(h) steps to find the node and delete it, and O(h) rebalancing steps in the worst case.

The textbook does not provide a deletion algorithm, but I do. The proof that it is correct is tedious, but I include it because, if for no other reason, I need to prove to myself that it is correct. So here we go.

Suppose that we delete a node from the right subtree of a node, C. Assume that the tree was balanced prior to the deletion but that after the deletion it is not balanced. Let h be the height of the right subtree. Then the left subtree is of height h+2. Unlike the unbalanced tree that results from an insertion, the resulting unbalanced tree may have more than one leaf node in the left subtree at height h+2. I claim the following:

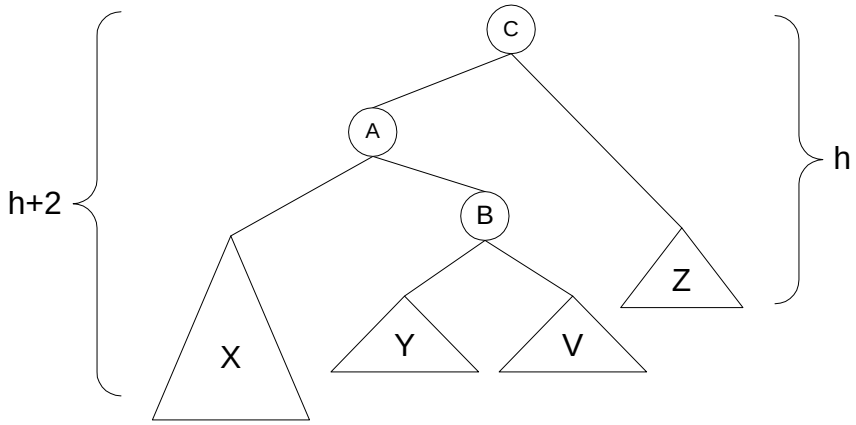
Lemma. Suppose that C is the root of an AVL tree in whose right subtree a deletion has occurred, and all subtrees of C are balanced. Suppose that t is a pointer to C and that height(t->left) – height(t->right) = 2. Then an LL rotation at t will rebalance the tree if and only if height((t->left)->left) >= height((t->left)->right) and an LR rotation will rebalance the tree otherwise.

Proof.

First I prove that if height((t->left)->left) >= height((t->left)->right) then an LL rotation will rebalance the tree. I do this in two steps, separating the case of unequal heights from equal heights. the Assume that the tree is rooted at C. Assume also that t is a pointer to C, that the height of the right subtree is h, and consequently that the height of the left is h+2.

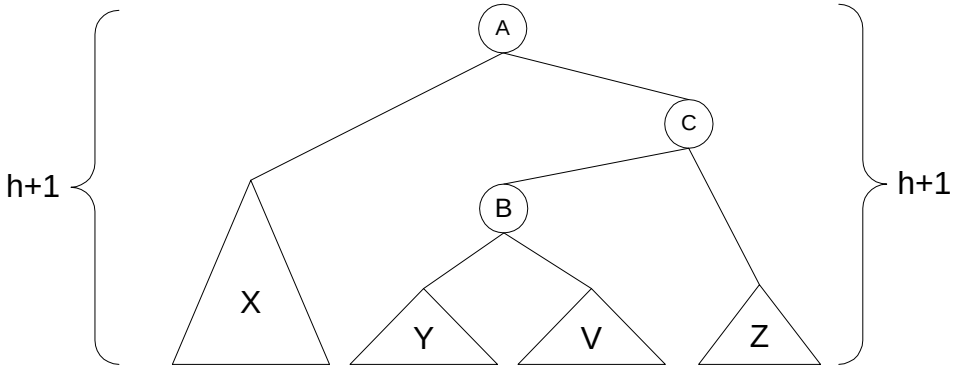
Case 1: height((t->left)->left) > height((t->left)->right)

In this case, the left subtree of A is deeper than the right subtree. Since A is an AVL tree, the height to the bottom of one or both of Y or V is exactly h+1. It is possible that one is one level taller than the other. Before the LL rotation, the tree will be as is depicted in the next figure.



Case 1

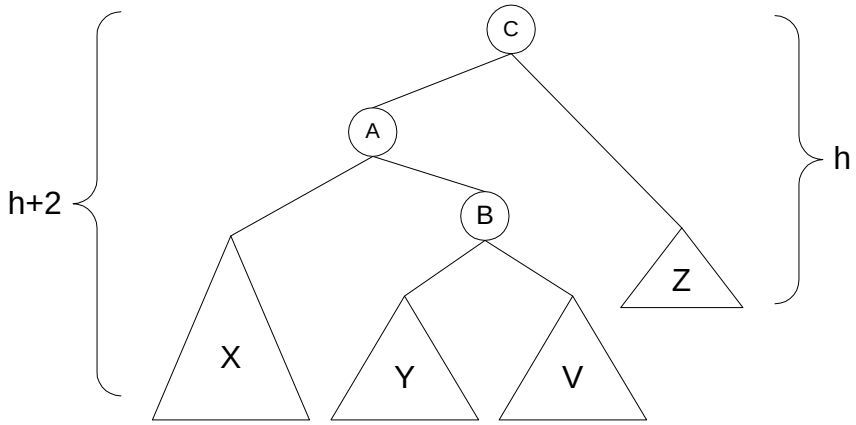
The figure below shows the tree after the LL rotation at the root. The tree now has height h+1, and the bottoms of X and Z are at the same height, and one or both of Y and V have their bottom nodes at this height as well. This proves that the LL rotation rebalances the tree.



Case 1, after the LL rotation

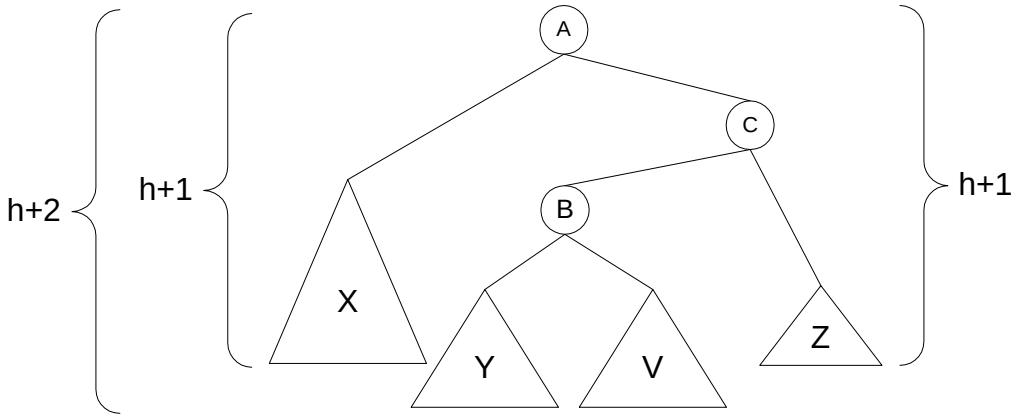
Case 2: height((t->left)->left) = height((t->left)->right)

If the heights of the left and right subtrees of A are equal, then it means that either Y or V has nodes at depth h+2 in the tree, or possibly both.



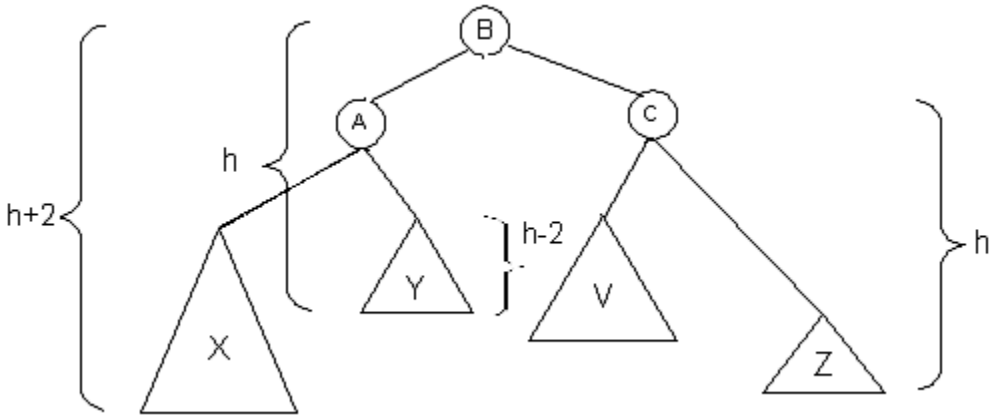
Case 2

After the LL rotation, the tree will be as shown below. The height of the right subtree of A will still be h+2 relative to the pointer t, and the tree remains an AVL tree.



Case 2, after the LL rotation

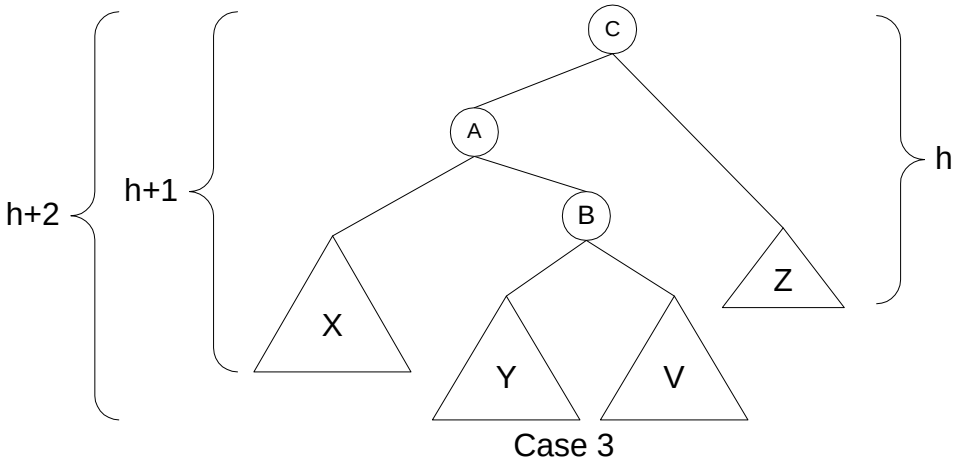
This proves that the LL rotation will rebalance the tree if the height of the right subtree of the left child is not greater than that of the left subtree of the left child. To prove that the LR rotation will fail if this condition is true, consider the tree in Case 2, but with height(Y) = height(V) - 1. In this case, height(Y) = h-2 and height(X) = h. After the LR rotation at C, the resulting tree will be unbalanced at node A:



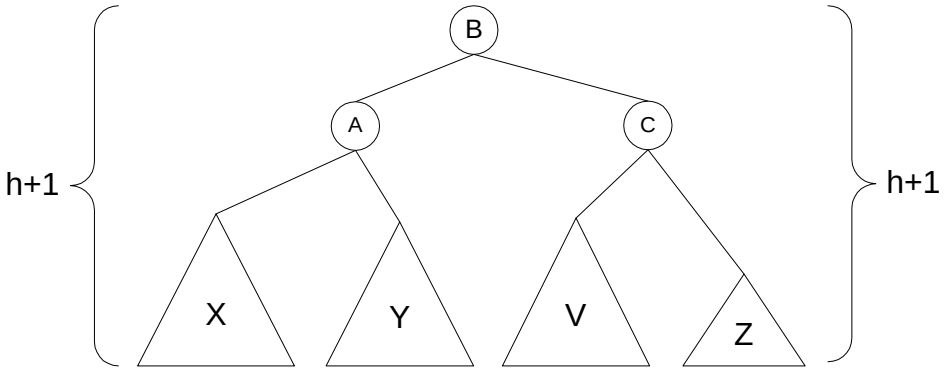
You can see that the height difference between the left and right subtrees, X and Y, of A is now 2. This shows that an LL rotation must be applied in this case. An LL must also be applied in Case 1 for the same reason, because the Case 1 tree will have both Y and V of height h-2.

Case 3: height((t->left)->left) < height((t->left)->right)

In this case the depth of the bottom of X is h+1, and the depth of the bottommost nodes of one or both of Y or V is h+2, as shown below. After an LR rotation, the tree will be as shown in the next figure. The bottoms of X and V are at h+1, and one or both of Y and V is at h+1, and if one of them is not, then it is at height h, since node B was balanced prior to the rotation, implying that the shorter one can be at most one level shorter than the other. This proves that the last case is true.



Case 3



Case 3, after an LR rotation

I leave it as an exercise for you to show that if an LL rotation were tried in this case, the tree would be unbalanced afterwards.

Worst case AVL Trees (Fibonacci Trees)

Do AVL trees have O(log N) depth in the worst case? To answer this question, we can try to determine the tallest possible AVL trees with a fixed number of nodes. It turns out to be easier to do the converse, namely given an AVL tree of height h, what is the least number of nodes in it? These *minimal* AVL trees are worst case trees, because they pack the least nodes possible for their height.

Let S(h) be the number of nodes in a minimal AVL tree of height h. If T is a minimal AVL tree of height h, then its left and right subtrees must also be minimal AVL trees. Furthermore, their heights differ by at most 1. This means that one is of height h-1 and the other is of height h-2, because the root adds 1 to the height of each. This leads to the recurrence relation:

S(0) = 1
S(1) = 2
S(h) = S(h-1) + S(h-2) + 1

Notice that the recurrence looks similar to the Fibonacci number recurrence relation. If you write out the first few terms you get 1, 2, 4, 7, 12, 20, 33, 54. It is easy to prove by induction that S(h) = F(h+2) – 1, where F(k) is the k^{th} Fibonacci number. If we write instead f_k as the kth Fibonacci number, then you can see that the Fibonacci recurrence is of the form

(1) $f_n = a_1 f_{n-1} + a_2 f_{n-2} + \cdots + a_{k-2} f_{k-2}$

in which k=2 and $a_1=a_2=1$:

(2) $f_n = f_{n-1} + f_{n-2}$

which can be rewritten as

(3) $f_n - f_{n-1} - f_{n-2} = 0$

A recurrence in this form is called a *homogeneous linear recurrence* with constant coefficients. All such recurrences have solutions of the form

$$F_n = cr^n$$

Therefore, the Fibonacci equation is

(4) $cr^n - cr^{n-1} - cr^{n-2} = 0$

which is equivalent to

(5) $cr^{n-2}(r^2 - r - 1) = 0$

This has two non-zero solutions, which we can denote ϕ_1 and ϕ_2. I.e., $\phi_1(n)$ and $\phi_2(n)$ are functions that satisfy equation (5).

Since $\phi_1(n)$ and $\phi_2(n)$ are both solutions to (5), so is their sum $\phi_1 + \phi_2$, since if

$\phi_1(n) = \phi_1(n-1) + \phi_1(n-2)$

and

$\phi_2(n) = \phi_2((n-1) + \phi_2(n-2)$

then

$\phi_1(n) + \phi_2(n) = \phi_1(n-1) + \phi_2(n-1) + \phi_1(n-2) + \phi_2(n-2)$.

and so is any linear combination of them $a\phi_1 + b\phi_2$ as well. The initial conditions of the Fibonacci sequence are used to find a unique solution. In other words, we have two equations with two unknowns, a and b:

$$a\phi_1^0 + b\phi_2^0 = 0$$

$$a\phi_1^1 + b\phi_2^1 = 1$$

Solving these, a = -b = $1/(\phi_1 - \phi_2)$. Since

$$\phi_1 = \frac{1+\sqrt{5}}{2} \quad \text{and} \quad \phi_2 = \frac{1-\sqrt{5}}{2}$$

we have that

$$\text{F(k)} = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^h - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^h$$

is the unique solution satisfying the initial conditions. This is the non-recursive formula for the kth Fibonacci number. It follows that

$$\text{S(k)} = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{h+2} - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^{h+2} - 1$$

Since the second term $\left(\frac{1-\sqrt{5}}{2}\right)^{h+2}$ vanishes as k increases, and the third term is negligible,

$$S(k) \approx \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{h+2}$$

If we let

$$n = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{h+2} \approx 0.44 \cdot 2.618 \cdot 1.618^{k} \approx 1.1708 \cdot 1.618^{k}$$

and solve for k, we get

$$\log_2 n \approx 0.6942 \cdot k$$

or k = 1.44 log n. This means that the height of an AVL tree with n nodes is at most roughly 1.44 log n. In other words, the height of an AVL tree is about 45% taller than the height of a perfectly balanced full tree with n nodes, in the worst case.

If h is the least h for which N <= S(h) then the height of an AVL tree with N nodes is at most h, because if it were h' > h, then S(h') <= S(h) for h < h' and it would follow that we could remove some nodes from the tree of height h' to make it height h and have an AVL tree of height h with fewer than S(h) nodes.

Therefore, the AVL trees that are minimal trees of height h act as worst case trees: any tree with fewer than S(h) nodes has height less than h.
It follows that AVL trees have O(log n) height in the worst case.

An AVL tree interface

The author's AVL tree interface is included below, with the deletion implementation added by me.

```
template <class Comparable>
  class AvlTree;

  template <class Comparable>
  class AvlNode
  {
     Comparable element;
     AvlNode   *left;
     AvlNode   *right;
     int       height;

     AvlNode( const Comparable & theElement, AvlNode *lt, AvlNode *rt, int h = 0 )
       : element( theElement ), left( lt ), right( rt ), height( h ) { }
     friend class AvlTree<Comparable>;
  };
```

```
template <class Comparable>
   class AvlTree
   {
    public:
      explicit AvlTree( const Comparable & notFound );
      AvlTree( const AvlTree & rhs );
      ~AvlTree( );

      const Comparable & findMin( ) const;
      const Comparable & findMax( ) const;
      const Comparable & find( const Comparable & x ) const;
      bool isEmpty( ) const;
      void printTree( ) const;

      void makeEmpty( );
      void insert( const Comparable & x );
      void remove( const Comparable & x );

      const AvlTree & operator=( const AvlTree & rhs );

     private:
      AvlNode<Comparable> *root;
      const Comparable ITEM_NOT_FOUND;

      const Comparable & elementAt( AvlNode<Comparable> *t ) const;

      void insert( const Comparable & x, AvlNode<Comparable> * & t ) const;
      AvlNode<Comparable> * findMin( AvlNode<Comparable> *t ) const;
      AvlNode<Comparable> * findMax( AvlNode<Comparable> *t ) const;
      AvlNode<Comparable> * find( const Comparable & x, AvlNode<Comparable> *t )
const;
      void makeEmpty( AvlNode<Comparable> * & t ) const;
      void printTree( AvlNode<Comparable> *t ) const;
      AvlNode<Comparable> * clone( AvlNode<Comparable> *t ) const;

         // Avl manipulations
      int height( AvlNode<Comparable> *t ) const;
      int max( int lhs, int rhs ) const;
      void rotateWithLeftChild( AvlNode<Comparable> * & k2 ) const;
      void rotateWithRightChild( AvlNode<Comparable> * & k1 ) const;
      void doubleWithLeftChild( AvlNode<Comparable> * & k3 ) const;
      void doubleWithRightChild( AvlNode<Comparable> * & k1 ) const;
   };
```

I include the implementations of the insertion methods and nothing else. These are the only items of interest.

Insertion Implementation

```
template <class Comparable>
void AvlTree<Comparable>::insert( const Comparable & x, AvlNode<Comparable> * & t )
const
      {
        if( t == NULL )
           t = new AvlNode<Comparable>( x, NULL, NULL );
        else if( x < t->element )
        {
           insert( x, t->left );
           if( height( t->left ) - height( t->right ) == 2 )
              if( x < t->left->element )
                 rotateWithLeftChild( t );
              else
                 doubleWithLeftChild( t );
        }
        else if( t->element < x )
        {
           insert( x, t->right );
           if( height( t->right ) - height( t->left ) == 2 )
              if( t->right->element < x )
                 rotateWithRightChild( t );
              else
                 doubleWithRightChild( t );
        }
        else
           ;  // Duplicate; do nothing
        t->height = max( height( t->left ), height( t->right ) ) + 1;
      }
```

Deletion Implementation

```
template <class Comparable>
void AvlTree<Comparable>::remove( const Comparable & x, AvlNode<Comparable> * & t )
const
{
      if ( t == NULL )
            // can't delete from an empty tree
            return;

      if ( x < t->element )
```

```
{
        // delete from the left subtree
        remove( x, t->left );

        // check if the heights of the subtrees are now too different
        if ( height( t->right ) - height( t->left ) == 2 ) // unbalanced
                // right subtree too tall relative to left
                // Which rotation to use depends on whether the left subtree of the
                // right subtree is larger, or the right of the right is larger.
                // If the left is larger we MUST use
                if ( height((t->right)->right) >=  height((t->right)->left) )
                        rotateWithRightChild( t );
                else
                        doubleWithRightChild( t );
}
else if( t->element < x )
{
        // delete from the right subtree
        remove( x, t->right );
        if( height( t->left ) - height( t->right ) == 2 ) // unbalanced
                // left subtree too tall
                if( height((t->left)->left) >= height((t->left)->right) )
                        rotateWithLeftChild( t );
                else
                        doubleWithLeftChild( t );
}
else { // delete this node
        if ((t->left != NULL) && (t->right != NULL) )  // two non-empty subtrees
        {
                t->element = findMin(t->right)->element;
                remove(t->element, t->right);
                if ( height( t->left ) - height( t->right ) == 2 ) // unbalanced
                        // left subtree too tall
                        if ( height((t->left)->left) >= height((t->left)->right) )
                                rotateWithLeftChild( t );
                        else
                                doubleWithLeftChild( t );
        }
        else {
                AVLNode<Comparable>* OldNode = t;
                t = (t->left != NULL)? t->left : t->right;
                delete OldNode;
        }
}
if ( NULL != t )
```

```
            t->height = max( height( t->left ), height( t->right ) ) + 1;
}

template <class Comparable>
void AvlTree<Comparable>::rotateWithLeftChild( AvlNode<Comparable> * & k2 ) const
    {
      AvlNode<Comparable> *k1 = k2->left;
      k2->left = k1->right;
      k1->right = k2;
      k2->height = max( height( k2->left ), height( k2->right ) ) + 1;
      k1->height = max( height( k1->left ), k2->height ) + 1;
      k2 = k1;
    }

    /**
     * Rotate binary tree node with right child.
     * For AVL trees, this is a single rotation for case 4.
     * Update heights, then set new root.
     */
template <class Comparable>
void AvlTree<Comparable>::rotateWithRightChild( AvlNode<Comparable> * & k1 ) const
    {
      AvlNode<Comparable> *k2 = k1->right;
      k1->right = k2->left;
      k2->left = k1;
      k1->height = max( height( k1->left ), height( k1->right ) ) + 1;
      k2->height = max( height( k2->right ), k1->height ) + 1;
      k1 = k2;
    }

    /**
     * Double rotate binary tree node: first left child.
     * with its right child; then node k3 with new left child.
     * For AVL trees, this is a double rotation for case 2.
     * Update heights, then set new root.
     */
template <class Comparable>
void AvlTree<Comparable>::doubleWithLeftChild( AvlNode<Comparable> * & k3 ) const
    {
      rotateWithRightChild( k3->left );
      rotateWithLeftChild( k3 );
    }

    /**
     * Double rotate binary tree node: first right child.
```

```
     * with its left child; then node k1 with new right child.
     * For AVL trees, this is a double rotation for case 3.
     * Update heights, then set new root.
     */
    template <class Comparable>
    void AvlTree<Comparable>::doubleWithRightChild( AvlNode<Comparable> * & k1 )
const
    {
      rotateWithLeftChild( k1->right );
      rotateWithRightChild( k1 );
    }
```

B-Trees

AVL trees and other binary search trees are suitable for organizing data that is entirely contained within computer memory. When the amount of data is too large to fit entirely in memory, i.e., when it is *external* data, these structures perform poorly. This is because of the enormous disparity in access times between memory accesses and disk accesses. The big-O analysis may show that a search is (O(log n)) in the worst case, but it hides the fact that the constants may be extremely large.

The problem is that disk accesses are typically 1000 times slower than in-memory accesses. For example, consider a disk revolving at 10,800 RPM, current state of the art. One revolution takes 1/180th second. Since on average, we need to seek one half a rotation, we need $1/360^{th}$ of a second to begin reading the data, or about 3 msec. In 3 msec., a typical CPU can execute about 75,000 instructions. If we use a binary search tree as a disk's file index, then every node access requires time equivalent to 75,000 disk accesses!

One solution is to increase the degree of the tree, so that the tree is flatter and fewer disk accesses are needed. More time is spent in the CPU to find the right node, but that is small compared to disk I/O. In 1970, R. Bayer and E. McCreight discovered (or invented, depending on your point of view) a multi-way tree called a B-tree. The strategy underlying the B-tree idea is to flatten the tree, making it a high degree, and pay the bill by doing more work inside each internal node, possibly a search through a list of children, to find the correct child. Even resorting to a linear search of an array of children's keys is worth the effort, since it saves the equivalent of hundreds of thousands of instructions.

The textbook defines a slight variation on the original B-tree idea.

Definition. A ***B-tree of order m*** and ***L*** is an m-ary tree with the following properties:

1. The data items are stored in the leaves.
2. The interior nodes store up to m-1 keys, K_1, K_2, ..., K_{m-1}. Key K_i is the smallest key in subtree T_{i+1} of the node.
3. For all keys in a given node, $K_1 < K_2 < \ldots < K_{m-1}$.

4. All interior nodes except the root have between ceil(m/2) and m children.
5. If the root is not a leaf node, then it has between 2 and m children.
6. All leaf nodes are at the same depth.
7. If the tree has at least ceil(L/2) data items then all leaf nodes have between ceil(L/2) and L children.

The figure below shows an internal node of a B-tree of order m. It has m subtree pointers, P_0 through P_{m-1}.

T_1	K_1	T_2	K_2	T_3	.....	T_{m-1}	K_{m-1}	T_m

Note about terminology.

The term "leaf node" in the above definition refers to the nodes that hold data items exclusively. These leaf nodes are a different type of nodes from the interior nodes, which store pointer and keys. The definition, part 5, states that the root of the tree may be a leaf node or an interior node. When the first few items are inserted into the tree, there are not enough of them to require an interior node. Therefore, the tree consists of a single leaf node. As soon as there are enough data items to warrant two leaf nodes, an interior node is created with pointer to the two leaf nodes.

Determining Values for m and L

The design of a B-tree should take into consideration the size of a disk block, the size of a disk block address, the size of a data item, and the size of the key. For example, suppose a disk block is 4096 bytes, disk addresses are 4 bytes long, a data item is a 512 byte record, and a key is 32 bytes long. Since a node has m-1 keys and m children, we need 32(m-1) + 4m bytes per interior node. Solving for the largest m such that 36m-32 <= 4096, we get m = 114. 8 data items fit in a disk block. It follows that the way to minimize disk reads is to let m = 114 and L = 8. Each time a block is accessed, all keys for a given node are loaded, and the appropriate subtree can be determined. A more accurate analysis should really be done, to account for actual search time, if m starts getting very large (because keys are small compared to block sizes).

B-Tree Search

To search using a B-tree requires either linear searching or binary searching through the keys in each node. If binary search is used, each node requires O(log m) comparisons. When the leaf node s finally reached, the data items can be searched linearly or using binary search. The number of disk accesses is equal to the height of the tree, which is $O(\log_m n)$.

B-Tree Insertion

Inserting into a B-tree is much more complex than into an AVL tree. This is because the requirements are more stringent. The number of children must be controlled in both leaf nodes and interior nodes. An insertion can cause a leaf node to exceed its child limit. When this

happens, the node must be split into two nodes each with fewer children. This split in turn might cause the parent node's child limit to be exceeded, and so on. The algorithm is roughly:

Search the tree for the place to insert x. If x is already in the tree, return.

If the leaf node has fewer than L children, insert x into the appropriate position and return. Otherwise, split the leaf into two leaves, one having *ceil*((L+1)/2) and the other having *floor*((L+1)/2) items.

Recursively do the following:
If the parent has fewer than m children,
  insert the new leaf into the parent node and update the keys of the parent so that
  there is a new key for the new leaf.
Otherwise,
  split the parent into two nodes, a left and a right, making the first ceil(m/2) nodes
  children of the left parent node and the remaining children into the
  right parent node.
  If the parent node has a parent, then insert the newly created node into the parent
  of the parent, and let the parent be the parent of the parent, and repeat these steps.
  If not, this is a root. Create a new root and make the two nodes children of the new root.

Notice that the tree grows in height only as a result of the root node being split. This is an extremely rare event. The tree reaches a height of h (measured as the number of edges from the root to the leaf nodes) as a result of h-1 splitting operations. A tree of height h of order m and L has at least $2 \cdot (\lceil m/2 \rceil)^{h-1} \cdot \lceil L/2 \rceil$ data items. For example, if m = 20, L = 40 and h = 4, the tree has at least 40,000 data items and the root split just 3 times! What is the most data items in this tree? If a tree has 40,000 items, what is the least height it would have, given m and L?

B-Tree Deletion

Deleting a node from a B-tree is as complex as B-tree insertion. A worst-case deletion may require deleting nodes all the way up to the root. If it makes it that far, the root is deleted and the height of the tree is diminished by 1.

The problem with deletion is that a node may have too few children after a deletion. In that case, nodes are merged. Merging is the opposite of splitting. The node is merged with the node to its left, unless it is the leftmost node, in which case it is merged with the node to its right. If the total number of nodes in the combined node is greater than L, then merging cannot be done, in which case data is borrowed from the adjacent node to replace the item that was deleted.

If merging takes place, then the gap in the sequence of child trees is filled in by moving all sibling trees to the left to fill in the place of the node that was deleted. If the parent node now has too few children, i.e., less than ceil(m/2), then the parent node is merged with a sibling of the parent, combining their children into a single node. The method of doing this is very straightforward. If this merging cannot be done, then borrowing is used instead, taking a child tree from the neighbor to replace the subtree that was merged.

This process can repeat all the back to the root. If the root has fewer than ceil(m/2) children, it is okay as long as it has at least two children. If it has only one child node, the root is deleted and this child node becomes the new root.

Chapter 5 Hashing and Hash Tables

5.1 Introduction

A hash table is a look-up table that, when designed well, has nearly $O(1)$ average running time for a find or insert operation. More precisely, a ***hash table*** is an array of fixed size containing data items with unique keys, together with a function called a ***hash function*** that maps keys to indices in the table. For example, if the keys are integers and the hash table is an array of size 127, then the function $hash(x)$, defined by

$\text{hash}(x) = x\%127$

maps numbers to their modulus in the finite field of size 127.

Conceptually, a hash table is a much more general structure. It can be thought of as a table H containing a collection of *(key, value)* pairs with the property that H may be indexed by the key itself. In other words, whereas you usually reference an element of an array `A` by writing something like `A[i]`, using an integer index value `i`, with a hash table, you replace the index value `"i"` by the key contained in location `i`. For example, if H contains the set of pairs

```
("Italy", "Rome"), ("Japan", "Nagano"), ("Canada", "Banff")
```

then you could write a statement such as

```
print H["Italy"]
```

and `Rome` would be printed, or

```
print H["Japan"]
```

and `Nagano` would be printed.

A table that can be addressed in this way, where the index is content rather than a subscript, is called a ***content-addressable-table*** (***CAT***). Hash tables are content-addressable tables. Obviously, if you can look up a value in $O(1)$ running time, you've got a good thing going, especially if inserting the value also takes $O(1)$ time. This would be a much better search table than a binary search tree of any kind, balanced or not.

5.2 Properties of a Good Hash Function

To *hash* means to chop up or make a mess of things, liked hashed potatoes. A hash function is supposed to chop up its argument and reconstruct a value out of the chopped up little pieces. Good hash functions make the original value hard to reconstruct from the computed hash value. To be good, a hash function should

- be easy to compute (for speed) and
- randomly disperse keys evenly throughout the table, making sure that no two keys map to the same index.

Easy to compute generally means that the function is an $O(1)$ operation, practically independent of the input size and hash table size. For example, if the function tried to find all of the prime factors of a given number in order to compute the hash function, this would not be easy to compute. Being easy to compute is a fuzzy concept. Dispersing the keys evenly means that there is as much distance between successive pairs of keys as possible. For example, if the hash table is of size 1000 and there are 200 keys in it, they should each be about five addresses apart from their neighbors.

In principle, if the set of keys is finite and known in advance, we can construct a perfect hash function, one that maps each key to a unique index. Much research has been done on how to find perfect hash functions efficiently. For example, if we have the integer keys

112, 46, 75, 515

we would want a function that maps them to the numbers0,1,2, and 3 uniquely. Coincidentally, although I picked these numbers randomly, my first guess at a perfect hash function for them was a good guess. Suppose that $g(x)$ is a function that returns the sum of the decimal digits in x. $g(x)$ could be defined recursively by

$$g(x) = \begin{cases} x & \text{if } x \leq 9 \\ x\%10 + g(x/10) & \text{otherwise} \end{cases}$$

For example, $g(122) = 5$ and $g(75) = 12$. Let $h(x)$ be defined by

$$h(x) = \begin{cases} g(x) & \text{if } g(x) \leq 9 \\ h(g(x) & \text{otherwise} \end{cases}$$

The function $h(x)$ is the sum of the digits in x, but added recursively until it falls into the range $[0, 9]$. For example, $h(112) = 1{+}1{+}2 = 4$, $h(46) = h(4{+}6) = h(10) = 1$, and $h(75) = h(7{+}5) = h(12) = 3$. It was just dumb luck that these numbers mapped to unique indices; this particular hash function is, in fact, a very poor hash function. It is poor because it does not use much of the information content in the key such as the order of the digits; to wit, $h(112) = h(121) = h(211) = 4$. There are tools that construct perfect hash functions. One such tool is GNU's *gperf*, which can be downloaded from ftp://ftp.gnu.org/pub/gnu/.

5.3 Collisions

Since almost all practical hash functions are not perfect, they will map one or more keys to the same indices, the way that $h()$ defined above mapped 112, 121, and 211 to the same index value 4. When two or more keys are mapped to the same location by the hash function, it is called a ***collision***. When a collision occurs, a new location must be found for the key that caused it, i.e., the second key to be hashed. The strategy for relocating keys for which a collision occurred is called the ***collision resolution strategy*** or algorithm. Since the relocation itself may cause further collisions, the goal of a collision resolution algorithm is to minimize the total number of collisions in a hash table.

Notation. Throughout the remainder of these notes on hash tables, M will denote the length of the hash table, and the table itself will be denoted by H, so that $H[0]$ through $H[M-1]$ are the table entries. Hash functions will be denoted by function symbols such as $h(x)$, $h_1(x)$, $h_2(x)$ and so on, and x will always be a key.

5.4 Hash Functions

While there are many different classes of integer functions to server as candidate has functions, I will focus on the three most common types:

division-based hash functions: primary operation is division of the key

bit-shift-based hash functions: bit-shifting is the primary operation

multiplicative hash functions: multiplication is the primary operation

5.4.1 Division Based Hash Functions

The division method uses the modulus operation (%), which is actually a form of division both conceptually and operationally[1]. In the division method, the hash function is of the form

$$h(x) = x \,\%\, M$$

Certain choices of M are obviously worse than others. For example, if x is an n-bit number and $M = 2^m$, then $x \,\%\, M$ is nothing more than the m least significant bits of x, which is not particularly good because it fails to use the remaining $m - n$ bits, throwing away a lot of information. If M is any even number, $h(x)$ will be even for even x and odd for odd x, introducing bias into the table. It is a bad idea for M to be a multiple of 3 also, because then two numbers that differ only by a permutation of their digits will be hashed to locations that differ by a multiple of 3. For example, if we let $M = 6$, then $52 \,\%\, 6 = 4$ and $25 \,\%\, 6 = 1$, a distance of 3 apart. Similarly, $1157 \,\%\, 6 = 5$ and if we permute 1157 to 7511, $7511 \,\%\, 6 = 5$. (This is a result of the fact that $10x \,\%\, 3 = 1$ and $4x \,\%\, 3 = 1$.)

There are other subtle problems that arise for various values of M. Making the table size M a prime number tends to avoid these problems. In fact, there are even certain types of primes that work better than others. (See *The Art of Computer Programming, Vol. 3, Sorting and Searching* by Knuth for more details.) The hash functions

$$h_1(x) = x \,\%\, 127$$
$$h_2(x) = x \,\%\, 511$$
$$h_3(x) = x \,\%\, 2311$$

are examples of division-based hash functions with M being prime. Compared to multiplication, addition, and subtraction, division is a slow operation; this is one reason to investigate other types of hash functions.

[1] $k = x \,\%\, M$ stores into k the remainder of x divided by M for positive x and M. Unless M is a power of 2, it will most likely require a hardware division.

5.4.2 Bit-Shifting Hash Functions Using the Middle Square Method

Bit-shifting refers to the machine operation in which data is shifted to the left or right by some fixed number of bits. A bit shift of k bits to the right, filling the upper k bits with zeros, is equivalent to integer division by 2^k. For example, shifting the bitstring $10100_2 = 20_{10}$ to the right two bits results in $00101_2 = 5_{10}$, which is 20/4. Let w be the word size in bits of a processor and let $W = 2^w$. In today's machines, w is usually either 32 or 64. W is the total number of integers that can be represented in a machine word. In ***twos-complement*** arithmetic, W is `MAXINT+1`, where `MAXINT` is the largest representable positive integer. The 32-bit number below is `MAXINT`; W is the number that could be represented with a 1 in the 33^{rd} bit and 0's in the remaining 32 bits.

```
MAXINT = 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

It is not hard to see that the expression $(b \,\%\, W)$ where b is any integer, is nothing more than the low-order 32 bits of b. If the machine word is 32 bits, this is just a way to ignore the overflow if $b \geq W$ and b is stored in a machine word. If you tell the processor to ignore integer overflow during integer computations, you can compute $(b \,\%\, W)$. Now consider the expression:

$$h(x) = \frac{M}{W}(x^2 \,\%\, W)$$

For this function to be efficient, M must be a power of 2. Assume that $M = 2^m$ for some positive integer m. Then $M/W = 2^{m-w}$. This is equivalent to

$$h(x) = 2^{m-w}(x^2 \,\%\, 2^w)$$

Notice that, because W is larger than M, $0 < M/W < 1$ and $(m - w) < 0$. Thus, multiplying by M/W is the same as shifting right $(w - m)$ bits. This hash function basically ignores the overflow caused by squaring x, and then shifts x^2 to the right by $(w - m)$ bits. Since x^2 is shifted $(w - m)$ bits to the right in a word with w bits, the leading $(w - m)$ bits are zero filled and the final value lies in the low-order m bits. This proves that $0 <= h(x) < 2^m = M$. Hence this hash function generates numbers between 0 and $M - 1$, as it should.

Notice also that this function does not require a division operation, since it is equivalent to the following *C/C++* expression:

```
h(x) = ( x * x ) >> ( w - m)
```

where `>>` is the shift-right operator in *C/C++*. This function uses one multiplication and one bit shift and is therefore much faster than the division method.

5.4.3 Multiplicative Hash Functions

A multiplicative hash function is of the form

$$h(x) = \frac{M}{W}(Ax \,\%\, W)$$

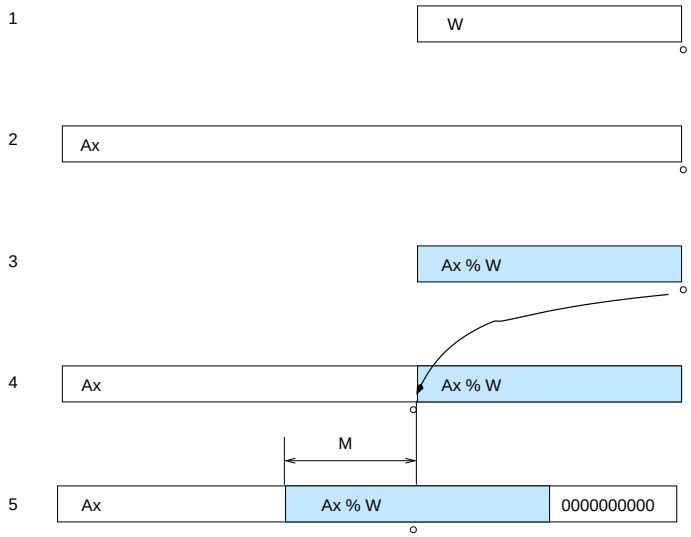


Figure 5.1: Multiplicative hash function.

where A is a carefully chosen constant. Although it is not necessary for M to be a power of 2, assume as well that $M = 2^m$. Below, I explain more about how to choose the constant A and why its value is so critical to the behavior of this hash function.

If $Ax < W$, this does nothing more than bit-shift the product Ax to the right by $(w - m)$ bits. If Ax is larger than W, it removes all but the low-order 32 bits of Ax and then shifts this value to the right $(w - m)$ bits. The result is a value between 0 and $M - 1$, for the same reason as the *Middle Squares* method result lies between 0 and $M - 1$.

Another way to think about this hash function is to pretend that the binary number has a decimal point (a binary point?) to the right of the least significant bit, and that the effect of the division by W is to move that binary "decimal point" to the left side of the most significant bit. Suppose that the rectangle labeled W in line 1 in Figure 5.1 is the length of a machine word with w bits. Suppose that the product Ax is much larger than W, as depicted in line 2. Then Ax % W is the portion of Ax shown in line 3, shaded lightly. Multiplying by M/W is the same as dividing by W and then multiplying by M. Dividing by W is the same as shifting the "binary decimal point" from the extreme right of the word to the point just left of W, as shown in line 4. Multiplication by M is the same as shifting the quantity Ax % W to the left by m bits. If the line labeled M represents the length of an m-bit word, then line 5 shows the result of this multiplication.

You have always assumed that the "binary decimal point" is to the right of the 32-bit integer. Suppose instead that all numbers z stored in machine words are actually the numerators of fractions of the form (z/W) and therefore that the binary decimal point is actually to the left of the word, as it is in line 4. The picture in line 4 shows you that we can think of Ax % W as just the fractional part of Ax. Let us denote the fractional part of any real number z by $\{z\}$ i.e.,

$$\{z\} = z - \lfloor z \rfloor$$

Then Ax % W is just $\{Ax\}$, and

$$h(x) = \lfloor M \cdot \{Ax\} \rfloor$$

Note too that $0 \leq M \cdot \{Ax\} < M$ since $0 \leq \{Ax\} < 1$.

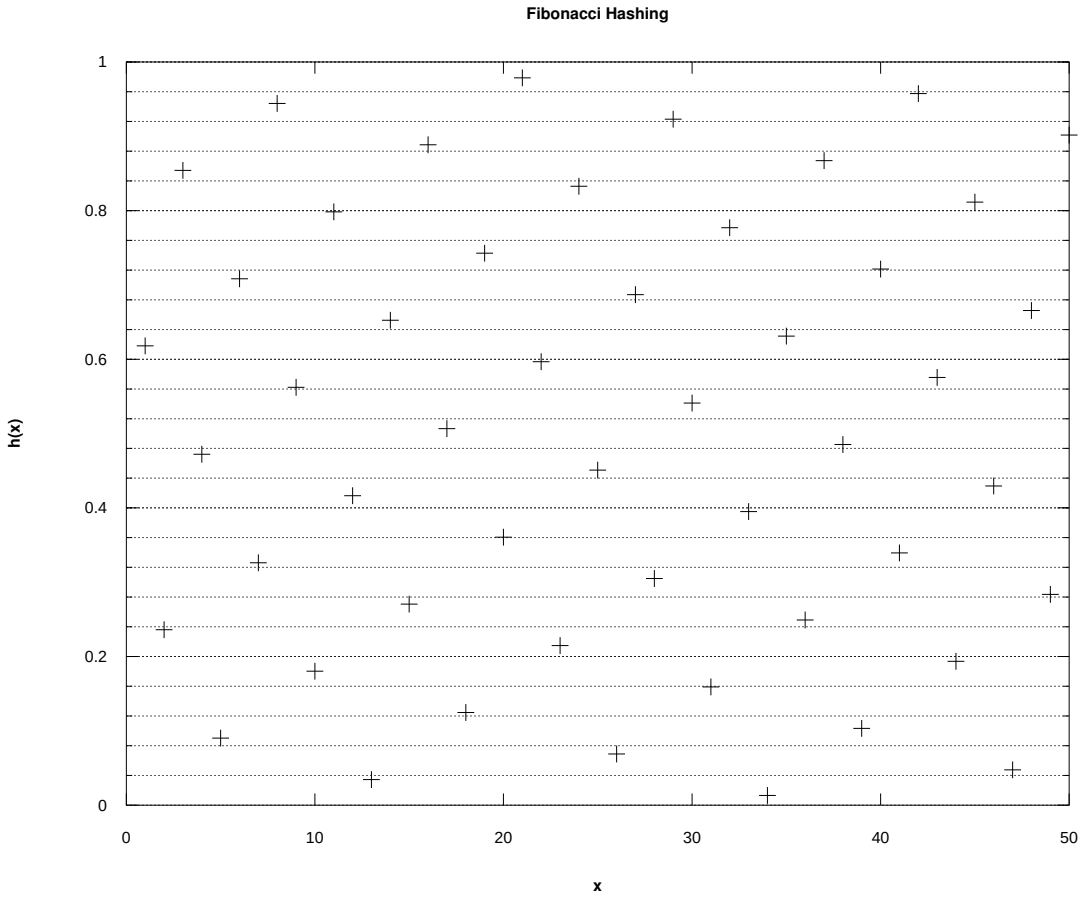


Figure 5.2: Spread of hashed values using Fibonacci hashing.

In the multiplicative method, the multiplier A must be carefully chosen to scatter the numbers. A very good choice for the constant A is ϕ^{-1}, where ϕ, known as the ***golden ratio***, is the positive root of the polynomial

$$x^2 - x - 1$$

This is because, x is a solution to the polynomial if and only if

$$\begin{aligned} & x^2 - x - 1 = 0 \\ iff\ & x^2 - x = 1 \\ iff\ & x - 1 = \frac{1}{x} \end{aligned}$$

In other words, a solution x has the property that its inverse is $x - 1$. The solution, ϕ, is called the golden ratio because it arises in nature in an astonishing number of places and because the ancient Greeks used this ratio in the design of many of their buildings, considering it a divine proportion. Thus, ϕ and $\phi^{-1} = \phi - 1$ are both roots of x^2–x–1. ϕ is also the value to which f_n/f_{n-1} converges as $n \to \infty$, where f_n is the n^{th} Fibonacci number. Since $\phi^{-1} = \phi - 1$, it is approximately 0.61803398874989484820458683436563. (Well *approximately* depends on your notion of approximation, doesn't it?)

When we let $M = \phi^{-1}W$ as the multiplicative constant, the multiplicative method is called ***Fibonacci hashing***. The constant has many remarkable mathematical properties, but the property

that makes it a good factor in the above hash function is the following.

The sequence of values $\{\phi^{-1}\}, \{2\phi^{-1}\}, \{3\phi^{-1}\}, ...$ lies entirely in the interval $(0, 1)$. Remember that the curly braces mean, "the fractional part of", so for example, $2\phi^{-1} \approx 1.236067977$ and $\{2\phi^{-1}\} \approx 0.236067977$. The first value divides the interval into two segments whose lengths are in the golden ratio. In fact, every value divides the segment into which it is placed into two segments whose lengths are in the golden ratio. Moreover, each successive value is placed into the largest segment in the interval $(0, 1)$. This implies that the successive values are spread out across the interval uniformly, no matter how many points there are. See Figure 5.2.

When used in the hash function above, they spread successive keys into the hash table in the same way. This tends to reduce the possibility of collisions. A table showing the values of the multiplier for typical machine word sizes is shown in Table 5.1.

w	$\phi^{-1}W$
16	40,503
32	2,654,435,769
64	11,400,714,819,323,198,485

Table 5.1: Fibonacci hashing multipliers.

5.5 String Encodings

Keys are often strings. A hash function is usually a function from integers to integers, so how do we map strings to integers? Let an arbitrary string be denoted s, consisting of the $k + 1$ symbols over some fixed alphabet:

$s = s_0 s_1 s_2 ... s_k$

Let S denote the set of all possible strings over this alphabet. A ***string encoding*** is a function from the set of strings S to non-negative integers

encode : $S \rightarrow Z$

The `encode` function does not have to be invertible, i.e., one-to-one and onto; it may map different strings to the same number. Once we have an encoding function, it can be composed with a hash function h as follows:

```
table_index = h(encode(s));
```

Encodings need to be chosen carefully. There are good encodings and bad ones. Here is a bad one:

`encode1(`s`)` = `int(`s_0`)` + `int(`s_1`) + ... + int(`s_k`)`

This is bad because it ignores letter order, so that `stop`, `spot`, and `tops` hash to the same location. It is better to find an encoding that maps words that are permutations of each other to unique numbers. One way to do that is to reinterpret a text string as a number over a very large alphabet.

In our customary decimal number system, a numeral such as 3276 is a representation of the number that we express in English as "three thousand two hundred seventy six," which is equal to

$$3 \cdot 10^3 + 2 \cdot 10^2 + 7 \cdot 10^1 + 6 \cdot 10^0$$

When you see a hexadecimal numeral such as $BADCAB2$, you know that this is a representation of the number

$$B \cdot 16^6 + A \cdot 16^5 + D \cdot 16^4 + c \cdot 16^3 + A \cdot 16^2 + b \cdot 16^1 + 2 \cdot 16^0$$

where A represents the number 10, B represents 11, C represents 12, and so on. This numeration system is surjective (onto) in that, for any integer n, there is a unique hexadecimal string that represents it. We can generalize this idea. Suppose all strings are over an alphabet of, say 26 characters, such as the symbols, `a,b,c,d,...,z`. Suppose that `a` represents the value 0, `b`, the value 1, `c`, 2, up to `z` representing 25. Then the string `hashing` represents the number

$$\begin{aligned}
&h \cdot 26^6 + a \cdot 26^5 + s \cdot 26^4 + h \cdot 26^3 + i \cdot 26^2 + n \cdot 26^1 + g \cdot 26^0 \\
=&7 \cdot 26^6 + 0 \cdot 26^5 + 18 \cdot 26^4 + 7 \cdot 26^3 + 8 \cdot 26^2 + 13 \cdot 26^1 + 6 \cdot 26^0 \\
=&2162410432 + 8225568 + 123032 + 5408 + 338 + 156 \\
=&2170764934
\end{aligned}$$

This method of encoding strings maps any string s to a unique integer. The problem is that the integer is usually larger than can be represented in even a 64-bit number. In the example above, the value of `hashing` will not fit into a signed 32-bit integer. Nonetheless, this encoding can be used by selecting some of the letters from a given word instead of all of them. The encoding can use the even numbered letters or the first m letters, or every third letter, and so on.

Summarizing, let $s = s_0 s_1 s_2 ... s_k$ be a string over an alphabet containing B distinct symbols. As it does not matter which side of s we start from, we can define an encoding of s to a unique integer by

$$\text{encode } (s) = \sum_{j=0}^{k} s_j B^j = s_0 B^0 + s_1 B^1 + s^2 B^2 + \cdots + s_k B^k$$

As mentioned, unless the string is sampled, the encoded value for most strings will be too large. Another problem with this encoding is that it does not satisfy the condition that it is easy to compute. The naive way to compute it would require $k + (k-1) + (k-2) + ... + 2 + 1 = k(k+1)/2$ multiplications. A much more efficient way to compute this is to use ***Horner's Rule***, turning off integer overflow. Horner's Rule is a way to compute polynomials efficiently. It is based on applying the following definition of a polynomial recursively:

$$\begin{aligned}
a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n &= a_0 + x(a_1 + a_2 x + \cdots + a_n x^{n-1}) \\
&= a_0 + x(a_1 + x(a_2 + a_3 x + \cdots + a_n x^{n-2})
\end{aligned}$$

and so on. In other words, a polynomial $p(x)$ of degree n can be written $p(x) = a_0 + x(q(x))$ where $q(x)$ is of degree $n-1$. This definition uses the customary notation for polynomials, with the highest index coefficient multiplying the highest power of x. The following function uses Horner's Rule for computing a code value for a string over an alphabet with `RADIX` many symbols:

```
long long encode ( const int RADIX, const string & s)
{
    long long hashval = 0;
    for (int i = 0; i < s.length(); i++)
hashval = s[i] + RADIX * hashval; // p(x) = s_i + x(q(x))
   return hashval;
}
```

Notice that this function performs one multiplication and one addition per iteration, for a total of n multiplications and n additions for a string of length n.

5.6 Collision Resolution

There are two basic methods of collision resolution:

- separate chaining
- open addressing

Open addressing itself has many different versions, as you will see. ***Separate chaining*** is the easiest to understand, and perhaps implement, but its performance is not as good as open addressing.

5.6.1 Separate Chaining

In separate chaining, the hash table is an array of linked lists, with all keys that hash to the same location in the same list. New keys are inserted in the front of the list. In other words, each hash table entry is a pointer to a list of keys and their associated data items. See Figure 5.3.

To insert a key into the table, the hash table index is computed, and then the list is searched to see if the key is already in the table. If it is not, it is inserted at the head of the list. In the worst case, this requires a search of the entire list. On average, half of the list is searched on each insertion. It is not worth keeping the list in sorted order if it is short.

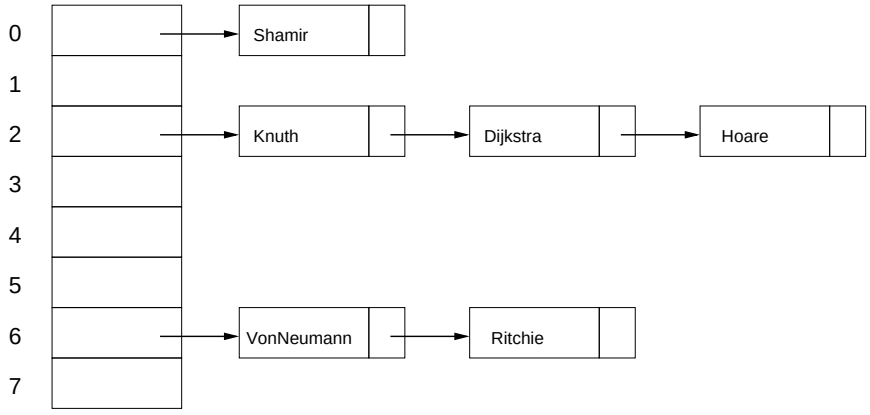


Figure 5.3: Separate chaining method of collision resolution.

The ***load factor***,λ, is defined as the ratio of the number of items in the table to the table size. If M is the table size and N is the number of items in the table, then $\lambda = N/M$. The table in Figure 5.3 has a load factor of $5/8 = 0.625$. In separate chaining, the load factor is the average length of the linked lists, if we count the empty ones as well as the non-empty ones. It is possible for the load factor to exceed 1.0 when separate chaining is used.

The following listing shows an example of a template class interface for a hash table. From the private part we can surmise that this table uses separate chaining. This interface provides functions to insert, remove, and find keys, as well as a copy constructor and an ordinary constructor. Once

again the weakness of C++ surfaces here because the interface, which should hide the fact that the hash table uses separate chaining, does not. We could overcome this by declaring an abstract base class and deriving the specific hash table from it.

```
template <class HashedObj>
class HashTable
{
    public:
        explicit HashTable     ( const HashedObj & notFound, int size =
           101 );
        HashTable              ( const HashTable & rhs ) :
                                 ITEM_NOT_FOUND(rhs.ITEM_NOT_FOUND ),
                                    Lists(rhs.theLists ) { }
        const HashedObj & find ( const HashedObj & x ) const;
        void makeEmpty         ( );
        void insert            ( const HashedObj & x );
        void remove            ( const HashedObj & x );

    private:
        vector<List<HashedObj> > Lists;   // array of linked lists
        const HashedObj ITEM_NOT_FOUND;
};

int hash( const string & key, int tableSize );
int hash( int key, int tableSize );
```

Let us now analyze the performance of this method of implementation. The most expensive operation in using a hash table is the operation of accessing a key in a hash table entry and inspecting its value. We call such an operation a ***probe***. Consider the find operation. If a search is successful, the key is found in exactly one of the nodes in some list. If that node is the k^{th} node in the list, then k probes are required. If all nodes are equally likely to be the one searched for, then there are, on average,

$$\frac{1}{\lambda}\sum_{k=1}^{\lambda} k = \frac{\lambda+1}{2}$$

probes in a list of length λ. As λ is the average list length, $(\lambda+1)/2$ is the expected time to find a key that is in the table. An unsuccessful search requires traversing the entire list, which is λ links. Inserting into a separately chained hash table takes the same amount of time as an unsuccessful search, roughly. An insertion has to search to see if the key is in the table, which requires traversing the entire list in which it is supposed to be located. Only after the list is checked can it be inserted at the front of that list.

Separate chaining makes deletion from a hash table quite easy because it is amounts to nothing more than a linked list deletion. On the other hand, as we will soon see, it is slower than deletion in open addressing because of the time it takes to traverse the linked lists.

5.6.2 Open Addressing

In open addressing, there are no separate lists attached to the table. All values are in the table itself. When a collision occurs, the cells of the hash table itself are searched until an empty one is found.

Which cells are searched depends upon the specific method of open addressing. All variations can be generically described by the functions

$$\begin{aligned} h_0(x) &= h(x) + f(0,x) \ \% \ M \\ h_1(x) &= h(x) + f(1,x) \ \% \ M \\ &\ldots \\ h_k(x) &= h(x) + f(k,x) \ \% \ M \end{aligned}$$

The idea is that the hash function $h(x)$ is first used to find a location in the hash table for x. If we are trying to insert x into the table, and the index $h(x)$ is empty, we insert it there. Otherwise we need to search for another place in the table into which we can store x. The function $f(i,x)$, called the ***collision resolution function***, serves that purpose. We search the locations

$$\begin{aligned} &h(x) + f(0,x) \ \% \ M \\ &h(x) + f(1,x) \ \% \ M \\ &h(x) + f(2,x) \ \% \ M \\ &\ldots \\ &h(x) + f(k,x) \% \ M \end{aligned}$$

until either an empty cell is found or the search returns to a cell previously visited in the sequence. The function $f(i,x)$ need not depend on both i and x. Soon, we will look at a few different collision resolution functions.

To search for an item, the same collision resolution function is used. The hash function is applied to find the first index. If the key is there, the search stops. Otherwise, the table is searched until either the item is found or an empty cell is reached. If an empty cell is reached, it implies that the item is not in the table. This raises a question about how to delete items. If an item is deleted, then there will be no way to tell whether the search should stop when it reaches an empty cell, or just "jump" over the hole. The way around this problem is to ***lazy deletion***. In lazy deletion, the cell is marked DELETED. Only when it is needed again is it re-used. Every cell is marked as either

ACTIVE: it has an item in it

EMPTY: it has no item in it

DELETED: it had an item that was deleted – it can be re-used

These three constants are supposed to be defined for any hash table implementation in the ANSI standard.

5.6.3 Linear Probing

In ***linear probing***, the collision resolution function, $f(i,x)$, is a linear function that ignores the value of x, i.e.., $f(i) = a \cdot i + b$. In the simplest case, $a = 1$ and $b = 0$. In other words, consecutive locations in the hash table are probed, treating the table like a circular list.

Example 1. Consider a hash table of size 10 with the simple division hash function $h(x) = x \ \% \ 10$ and suppose we insert the sequence of keys,

$5, 15, 6, 3, 27, 8$

In principle, only one collision should occur: 5 and 15 because they both map to the location 5. But linear probing causes many more collisions. After inserting 5, 15 causes a collision. It is placed in `H[6]`. Then 6 has a collision at `H[6]` and is placed in `H[7]`. 3 gets placed without a collision, but 27 collides with 6 and is placed in `H[8]`. This causes 8 to collide, and it is placed in $H[9]$. Figure 5.4 shows the state of the hash table after each insertion.

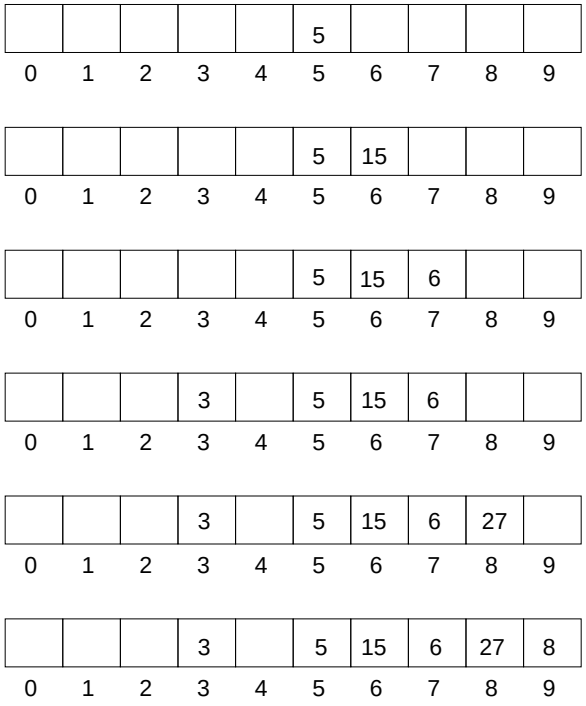


Figure 5.4: Linear probing example. This shows the successive states of the table when the keys $5, 15, 6, 3, 27, 8$ are inserted and the hash function is $h(x) = x \ \% \ 10$.

The problem with linear probing is that it tends to form ***clusters***. This phenomenon is called is ***primary clustering***. As more cells that are adjacent to the cluster are filled, a snowball effect takes place, and the cluster grows ever more faster. It can be proved that the expected number of probes for insertions and unsuccessful searches is approximately

$$\frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$$

and that the expected number of probes for successful searches is

$$\frac{1}{2}\left(1 + \frac{1}{(1-\lambda)}\right)$$

Linear probing performs very poorly as load factor increases.

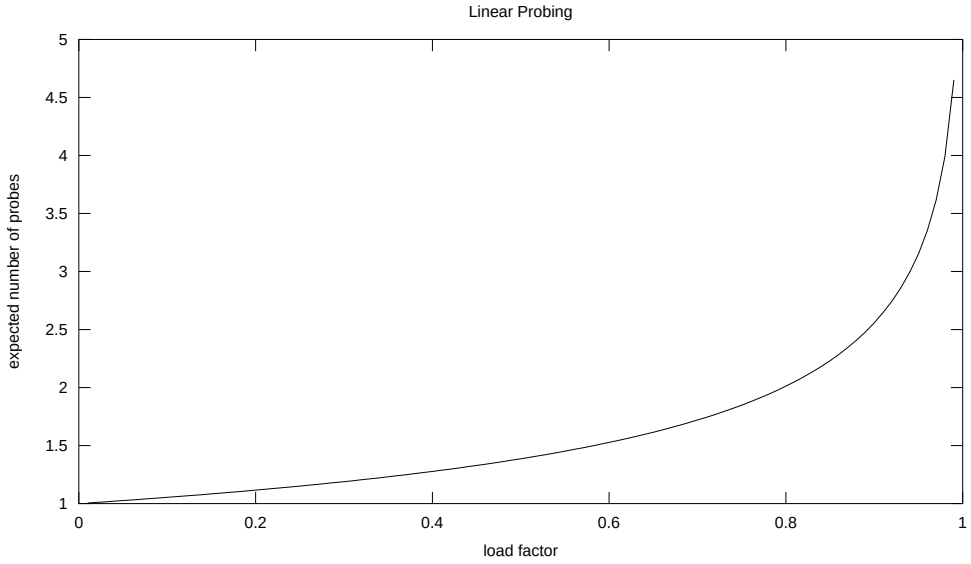


Figure 5.5: Graph of number of expected probes as a function of load factor in linear probing.

5.6.4 Random Probing

We can compare the performance of linear probing with a theoretical strategy in which clusters do not form and the probability that the next probe will succeed is independent of the probability that the previous probes succeeded. This is a kind of random collision resolution strategy. In this strategy, the number of probes in an unsuccessful search when the table has a load factor of λ is $1/(1–\lambda)$. This is because, if the collision resolution strategy is random, keys are placed uniformly throughout the table. Since λ is the fraction of the table that is occupied, the fraction of the table that is free is $1–\lambda$. This means that there is a λ probability that a random probe will hit an unoccupied cell, and a $(1–\lambda)$ probability that it hits an empty cell. The expected number of probes in an unsuccessful search is the expected number of cells that we visit before we find an empty cell, given that the item is not in the table. This is the mean of a geometric distribution with parameter $(1 - \lambda)$, which is $1/(1–\lambda)$.

As the table is filled, the load factor increases, from 0 to the current load factor,λ. The average number of probes is therefore approximated by the definite integral

$$I(\lambda) = \frac{1}{\lambda}\int_0^{\lambda} \frac{1}{1-x}dx = \frac{1}{\lambda}\ln\left(\frac{1}{1-\lambda}\right)$$

5.6.5 Quadratic Probing

Quadratic probing eliminates clustering. In quadratic probing the collision resolution function is a quadratic function of i and does not depend on x, namely $f(i, x) = i^2$. In other words, when a collision occurs, the successive locations to be probed are at a distance (modulo table size) of $1, 4, 9, 16, 25, 26, 49$, and so on. The sequence of successive locations is defined by the equations

$$\begin{aligned} h_0 &= h(x) \\ h_i &= (h_0 + i^2) \ \% \ M \end{aligned}$$

You should wonder why this is efficient, because it adds another multiplication for each probe. If it were implemented this way, it would not be efficient. Instead, it can be computed without multiplications by taking advantage of the recurrence relation

$$\begin{aligned} d_0 &= 1 \\ h_0 &= 0 \\ h_{i+1} &= h_i + d_i \\ d_{i+1} &= d_i + 2 \end{aligned}$$

The values of h_i are the successive squares. This relies on the observation that the odd numbers separate successive squares:

```
0     1     4     9     16     25     36 ...
   1     3     5     7      9     11  ...
      2     2     2     2      2  ...
```

To find an item x in the hash table when quadratic probing is used, the table must be probed until either an empty cell is found or until x is found. But this raises the question, is it possible that neither case will arise? The answer is yes, if M is not a prime number.

Example 2. Let $M = 24$, and suppose that we use the simple hash function $h(x) = x \ \% \ 24$ and quadratic probing. Further, suppose that the current state of the table is that $H[k] = k$ for $k = 0, 1, 4, 9, 12$, and 16. Only 6 out of 24 cells are in use, less than half. Suppose that we try to insert , into the table. The probe sequence will be

h_0	h_1	h_2	h_3	h_4	h_5
$(24+0)\%24$	$(24+1)\%24$	$(24+4)\%24$	$(24+9)\%24$	$(24+16)\%24$	$(24+25)\%24$
0	1	4	9	16	1

h_6	h_7	h_8	h_9	h_{10}	h_{11}
$(24+36)\%24$	$(24+49)\%24$	$(24+64)\%24$	$(24+81)\%24$	$(24+100)\%24$	$(24+121)\%24$
12	1	16	9	4	1

h_{12}	h_{13}	h_{14}	h_{15}	h_{16}	h_{17}
$(24+144)\%24$	$(24+169)\%24$	$(24+196)\%24$	$(24+225)\%24$	$(24+256)\%24$	$(24+289)\%24$
0	1	4	9	16	1

h_{18}	h_{19}	h_{20}	h_{21}	h_{22}	h_{23}
$(24+324)\%24$	$(24+361)\%24$	$(24+400)\%24$	$(24+441)\%24$	$(24+484)\%24$	$(24+529)\%24$
12	1	16	9	4	1

and the sequence $(0, 1, 4, 9, 16, 1, 12, 1, 16, 9, 4, 1, ...)$ will repeat *ad infinitum*. M must be prime or else the search will go on forever, but this is a necessary but not sufficient condition to prevent the infinite search. The table must also be less than half full.

Example 3. Suppose $M = 24$ and the table currently has the following contents:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0	1			4					9			12				16							

Suppose we try to insert 24. $h(24) = 0$. The locations searched are the successive squares, modulo 24, which are: $1, 4, 9, 16, 25\%24 = 1$, $36\%24 = 12$, $49\%24 = 1$, $64\%24 = 16$, $125\%24 = 1$, and so on. In fact, the only locations that will be probed are the ones that are currently filled, and no others. But if M is a prime number and the table is less than half full we are guaranteed to find a cell. This is stated as a theorem:

Theorem 4. *If quadratic probing is used, and table size is prime, a new element can always be inserted if the table is at least half empty.*

Proof. Suppose that M is prime and the table is at least half empty. Assume that M is larger than 2, because if $M = 2$, it is trivial to see that it is true: there is 1 empty cell and adding 1 to the first location finds it.

Since $M > 2$ and prime, it is an odd number. Let $M = 2m + 1$ for some m. Since the table is at least half empty and M is odd, at least $m+1$ cells are empty. (If only m cells were empty, it would be less than half of $2m + 1$.) Therefore at most m cells are full. Now suppose that the theorem is false. To say it is false means that indefinite, repeated probing fails to find an empty cell. Consider the first $m+1$ locations probed by quadratic probing, including the first location where the collision occurred. Let these $m + 1$ locations be labeled h_0, $h_1, h_2, \ldots, h_m$, where

$h_k = (h(x) + k^2)\ \%\ M$

Since probing did not find an empty cell in these first $m+1$ locations, all of these $m+1$ locations are full. But there are at most m full cells. This implies that two of these $m+1$ probes must have been at the same location. (Pigeon-hole principle: if there are $m + 1$ pigeons and only m pigeon holes, then two pigeons share a hole.) Suppose that h_i and h_j are the two probes at the same location, where $0 \le j < i \le m$. Then

$$\begin{aligned} h_i &= h_j && iff \\ (h(x) + i^2)\ \%\ M &= (h(x) + j^2)\ \%\ M && iff \\ i^2\ \%\ M &= j^2\ \%\ M && iff \\ (i^2 - j^2)\ \%\ M &= 0\ \%\ M && iff \\ (i + j)(i - j)\ \%\ M &= 0\ \%\ M \end{aligned}$$

Since M is a prime number it has no factors other than 1 and itself. This implies that either $(i - j)$ is zero or a multiple of M, or $(i + j)$ is a nonzero multiple of M. But since $i > j$, $(i - j) > 0$. Also, since $(i - j) < M$, their difference cannot be a multiple of M greater than 0. Therefore, the only possibility is that $(i + j)$ is a multiple of M. But $0 \le j < i \le m$, which implies that $j < m$ and $i \le m$, so the sum of i and j cannot be equal to or greater than $2m$. Since $2m < 2m + 1 = M$, this implies that $(i + j) < M$, so it certainly is not a positive multiple of M. This is a contradiction, which implies that the hypothesis that probing failed to find an empty location is false. The theorem must be true. □

5.7 Algorithms

The probing algorithm can be used for both insertion and searching. The quadratic probing function is called `findPos`, and is below. It returns a hash table location where the key is located, or the first empty cell found. The hash table must have a tag with each cell to indicate whether it is empty or not. This member is called `info`. The calling function can check if the return value is an empty cell using code such as

```
if ( H[findPos(x)].info == EMPTY)
```

where `findPos` can be defined as follows:

```
template <class HashedObj>
int HashTable<HashedObj>::findPos( const HashedObj & x ) const
{
int collisionNum = 0;
int currentPos = hash( x, array.size( ) );
    while( array[ currentPos ].info != EMPTY
        &&  array[ currentPos ].element != x ) {
        currentPos += 2 * ++collisionNum - 1;  // Compute ith probe
        if( currentPos >= array.size( ) )
            currentPos -= array.size( );
    }
    return currentPos;
}
```

The insertion algorithm:

```
template <class HashedObj>
void HashTable<HashedObj>::insert( const HashedObj & x )
{
    // Insert x as active
    int currentPos = findPos( x );
    if ( isActive( currentPos ) )
        return;
    array[ currentPos ] = HashEntry( x, ACTIVE );
    // Rehash
    // if the insertion made the table get half full,
    // increase table size
    if( ++currentSize > array.size( ) / 2 )
        rehash( );
}
```

The algorithm to find an item is

```
template <class HashedObj>
const HashedObj & HashTable<HashedObj>::find( const HashedObj & x ) const
{
   int currentPos = findPos( x );
   return
    isActive( currentPos ) ?
       array[ currentPos ].element : ITEM_NOT_FOUND;
}
```

5.8 Double Hashing

In ***double hashing***, the sequence of probes is a linear sequence with an increment obtained by applying a second hash function to the key:

```
f(i) = i * hash2(x);
```

We search locations `hash(x) + i*hash2(x)` for `i` = 1,2,3,...

The choice of the second hash function can be disastrous – it should never evaluate to a factor of the table size, obviously. It should be relatively prime to table size. It should never evaluate to 0 either. Choosing

```
hash2(x) = R- (x % R)
```

will work well if `R` is a small prime number.

5.9 Rehashing

If the hash table gets too full it should be resized. The best way to resize it is to create a new hash table about twice as large and hash all of the elements of the hash table into the new table using its hash function.

Rehashing is expensive, so it should only be done when necessary:

1. When an insertion fails, or
2. When the table gets half full, or
3. When the table load factor reaches some predefined value.

Chapter 6 Priority Queues

Many applications require a special type of queuing in which items are pushed onto the queue by order of arrival, but removed from the queue based on some other priority scheme. A priority scheme might be based on the item's level of importance for example. The goal in this chapter is to investigate efficient data structures for storing data in this fashion.

Examples:

- **Process or job queue**: Operating systems enqueue processes as they are created or woken up from some state of inactivity, but they usually pick the highest priority process in the queue to run next.
- **Task lists in general**: Task lists, such as project schedules or shopping lists, are often constructed by adding items to the list as they are thought of, in their order of arrival, but tasks are removed from the list and completed in some specific priority ordering, such as due date or order of importance.

A queue that supports an ordinary insertion operation at its rear, and a deletion operation that deletes according to some priority ordering is called a ***priority queue***. Since the highest priority element can always be thought of as the minimum element in some appropriate numeric ordering, this delete operation is called ***deleteMin***. For example, priority numbers can be assigned to the items so that priority 0 is the highest priority; priority 1 is second highest; 2, third highest, and so on. Therefore, a ***priority queue*** is a queue that supports a ***deleteMin*** operation and an ***insert*** operation. No other operations are required.

The question is, what is a good data structure for this purpose. The performance measure of interest is not the cost of a single operation, but of a sequence of N insertions and deleteMin operations, in some arbitrary, unspecified order. Remember that a queue is only a buffer, in other words, a temporary storage place to overcome short term variations in the rate of insertion and deletion. In the long run, the size of a queue must remain constant otherwise it implies that the two processes inserting and deleting are mismatched and that sooner or later the queue will get too large. Therefore the number of insertions must be equal to the number of deleteMins in the long run.

Naïve Implementations

The naïve approach, i.e., the first one to come to mind, is to use an ordinary queue as a priority queue. Insertion is an O(1) operation and deletion is O(n), where n is the current size of the queue, since the queue must be searched each time for the min element. An alternative is to use a balanced binary search tree instead of a queue. Insertions and deletions in a balanced binary search tree are both O(log n), but the overhead is high anyway.

The first suggestion, using an ordinary queue, will require $O(N^2)$ operations in the worst case. This is because it will require roughly N/2 insertions, which is O(N), since each is constant time, and N/2 deletions. But each deletion takes time proportional to the queue size at the time. In the

worst case, all insertions take place first and the deletions take N/2 + (N/2 - 1) + (N/2 - 2) + ... + 1 = $O(N^2)$ steps. On average, the insertions and deletions will intermingle and the average queue size for a deletion will be N/4. This still leads to $O(N^2)$ steps on average.

The use of a binary tree requires a slightly more complicated analysis.

These solutions are not efficient. They do not use the information available at the time of the insertion. When an item is inserted, we know its value. Knowing its value is a piece of information that can be used to position it to make future deletions more efficient.

Binary Heaps: An Efficient Implementation

A ***binary heap*** is a special kind of binary tree. Usually the "binary" is dropped from the term and it is just called a "***heap***".

To review our terminology, a binary tree of height h is ***full,*** or ***perfect,*** if there are 2^h nodes at depth h[1]. (This implies that all levels are full). A binary tree of height h is ***complete*** if the tree of height h-1 rooted at its root is full, and the bottom-most level is filled from left to right.

A binary tree has the ***heap order property*** if every node is smaller than or equal to its two children. Since each node has this property, it is not hard to prove by induction on the height of a node in the tree that every node is smaller than all of its descendants. A ***heap*** is a complete binary tree with the **heap-order property**. It follows from the definition that the root of a heap is the minimum element in the tree.

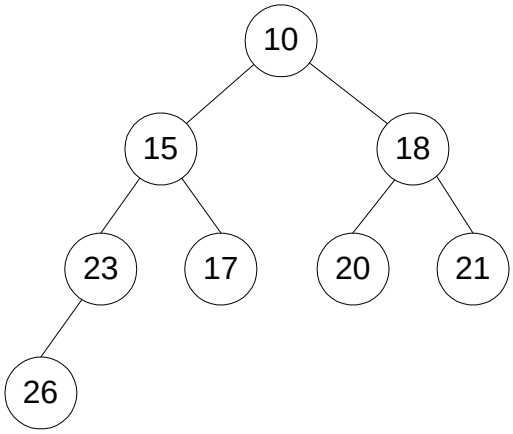
Heaps and Arrays

A heap can be implemented easily in an ordinary array if the root is placed in position 1 instead of position 0. Consider the array below, with keys, 10, 15, 18, 23, 17, 20 , 21, 26.

	10	15	18	23	17	20	21	26
0	1	2	3	4	5	6	7	8

which represents the binary tree

[1] The textbook calls this complete. The NIST allows either word. Some authors use full for what I call complete, and vice versa.



In this tree:

- left child(i) = 2i if i <= n/2
- right child(i) = 2i + 1 if i < n/2
- parent(i) = floor(i/2) if i > 1
- Number of nodes in a complete binary tree of height h is at least 2^h and at most $2^{h+1} - 1$.
- Height of a complete binary tree with n nodes is floor(log n).

The book does not prove this, but I include a proof here. Let left(k) denote the index of the left child of the node whose index is k, let right(k) be the index of the right child of the node with index k, and let parent(k) denote the index of the parent of the node with index k.

Theorem. If the nodes in a complete binary tree are numbered in breadth-first order, from left to right in each level, with the root having number 1, then

left(k) = 2k if k <= N/2
right(k) = 2k+1 if k < N/2
parent(k) = k/2 if k > 1

Proof

Assume that k is the index of an arbitrary node such that k <= N/2. Suppose node k has a left child. If it has a left child, then by the definition of a complete binary tree, the entire level to the right of node k must be filled, and the entire level to the left of the left child must also be filled.

Let d be the depth of node k in the tree. There are $2^d - 1$ nodes in the tree above level d because a full binary tree of height d–1 has $2^d - 1$ nodes. So how many nodes are to the left of node k in its level? It must be

$$k - (2^d - 1) - 1$$
$$= k - 2^d$$

How many nodes are to the right of k in level d? It must be 2^d less the number of nodes up to and including k:

$$2^d - (\ k - 2^d\ ) - 1 \ = \ 2^{d+1} - k - 1$$

Each child to the left of node k contributes 2 children in level d+1 to the left of node k's left child, so there are $2(k - 2^d)$ nodes to the left of left(k) in level d+1. Since there are $2^{d+1} - k - 1$ nodes to the right of node k in level d, the index of the left child of node k is

$$\begin{aligned} \text{left}(k) &= k + (2^{d+1} - k - 1) + 2(k - 2^d) + 1 \\ &= 2^{d+1} - 1 + 2k - 2^{d+1} + 1 \\ &= 2k \end{aligned}$$

The right child, if it exists, must have index 2k+1. Finally, since left(k) = 2k and right(k) = 2k+1, it follows that parent(k) = floor(k/2).

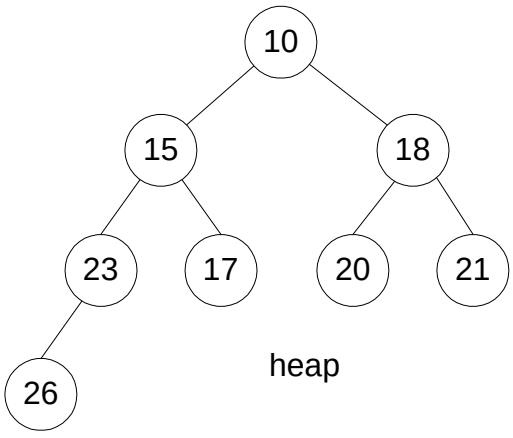
Corollary 1. In a complete binary tree with N nodes, the highest index non-leaf node is the node with index floor(N/2).

Proof.

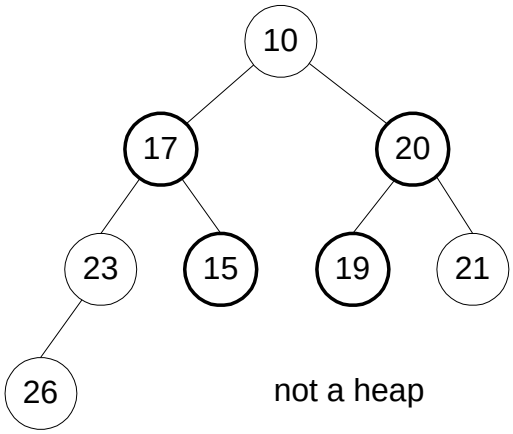
The highest index internal node has a left child and possibly a right child, and no node to its right has any children. If N is even, the last node is a left child of its parent, and the parent is node floor(N/2). If the last node is a right child of its parent, the parent has index floor(N/2) also.

Corollary 2. In a heap with N elements, the last N/2 elements are leaf nodes.

The fact that the index alone tells which node is the left child, right child, or parent implies that no links are required to represent the tree: children are found from the node's index and the parent is found from the node's index.



heap



not a heap

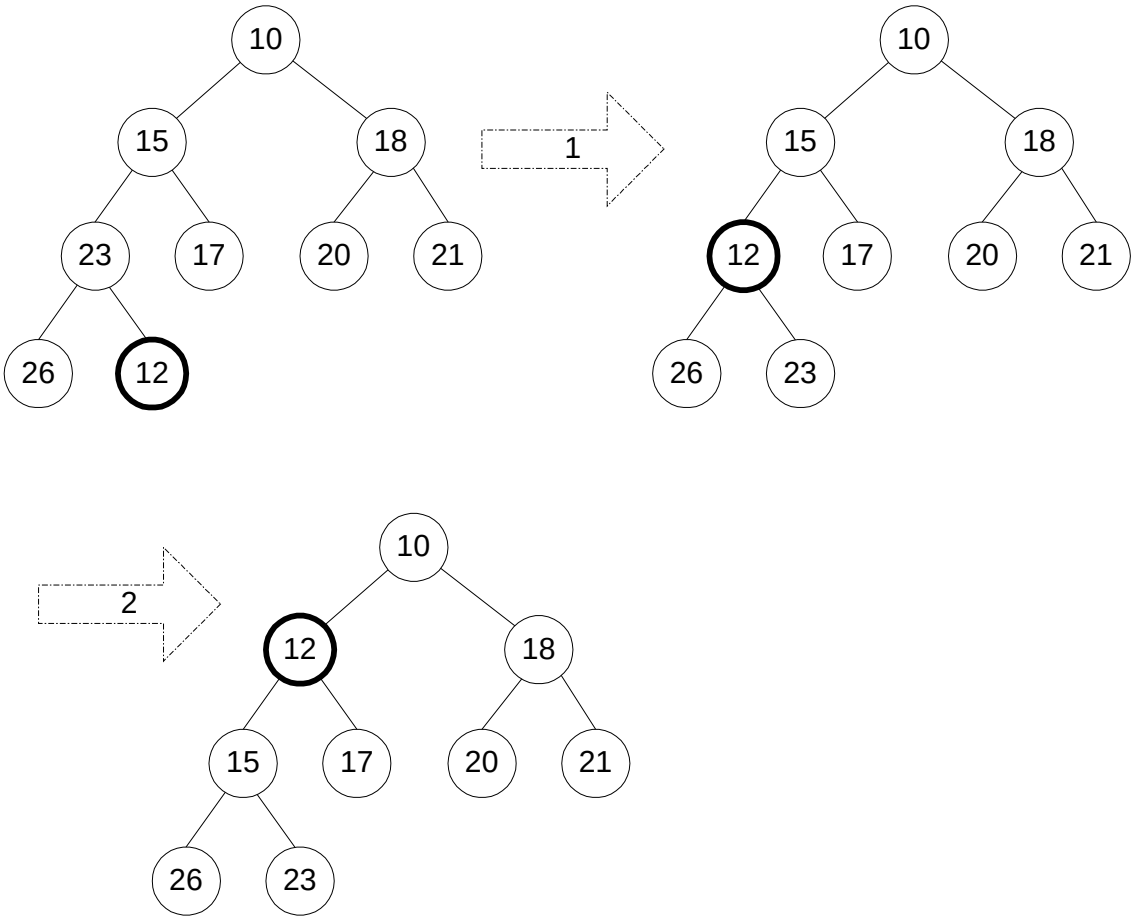
Insertion

Inserting into a heap is simple: put the element at the end of the array – in the bottom level of the tree to the right of the rightmost key. Then percolate it up to the top as far as necessary to restore the heap-order.

Code.

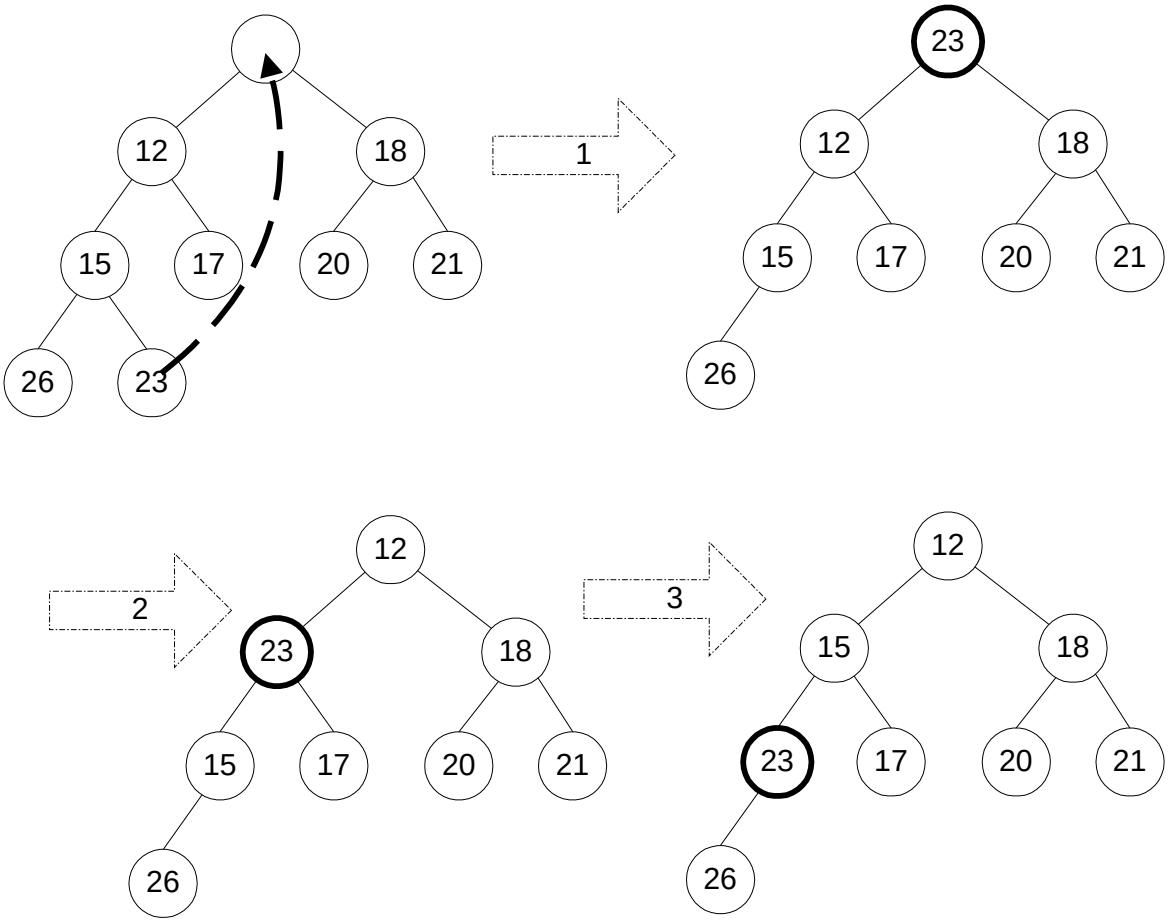
```
template <class Comparable>
 void BinaryHeap<Comparable>::insert( const Comparable & x )
 {
     if( isFull( ) )
         throw Overflow( );

         // Percolate up
     int hole = ++currentSize;
     for( ; hole > 1 && x < array[ hole / 2 ]; hole /= 2 )
         array[ hole ] = array[ hole / 2 ];
     array[ hole ] = x;
 }
```

Example of Insertion Algorithm



Deleting the Minimum Element

deleteMin is also very easy to implement. To delete the smallest element, remove the root. Now there is a hole. Take the last element of the heap (the rightmost leaf in the bottom level) and put it where the root was. It might be bigger than its children. Pick the smaller of the two children and swap it with the root. Repeat this if necessary after the swap with the subtree from which the swap was made. Do this until the children are leaf nodes.



Code.

```
template <class Comparable>
void BinaryHeap<Comparable>::deleteMin( )
{
   if( isEmpty( ) )
      throw Underflow( );

   array[ 1 ] = array[ currentSize-- ];
   percolateDown( 1 );
}
```

Building the Heap

A heap with n keys is built by inserting the n keys into an initially empty heap in any order without maintaining the heap property. After all items are in the tree, it is "heapified". A binary tree can be converted into a heap by starting at the level just above the lowest level and percolating down all keys that are too big. When this level is finished, the next level up is processed, and so on, until we reach the root. The algorithm is summarized below:

```
void BinaryHeap<Comparable>::heapify()
{
      for ( int i = n/2; i > 0; i--)
            percolateDown(i);
}
```

where `percolateDown` is defined by

```
void BinaryHeap<Comparable>::percolateDown( int hole )
{
    int child;
    Comparable tmp = array[ hole ];
    for( ; hole * 2 <= currentSize; hole = child )
    {
        child = hole * 2;
        if( child != currentSize && array[child + 1] < array[child])
            child++;
        if( array[ child ] < tmp )
            array[ hole ] = array[ child ];
       else
            break;
    }
    array[ hole ] = tmp;
}
```

Theorem. Heapifying a binary tree with n nodes takes O(n) time.

Proof

Observe that n steps are needed to insert into the heap if an array is used, since each insert is O(1). Then heapifying requires that each of the nodes from the root to the level above the bottom be percolated. In the worst case, each of these is moved to the lowest level. Thus, the total number of steps is equal to the sum of the heights of these nodes. First I prove the lemma below.

Lemma. The sum of the heights of the nodes in a full (perfect) binary tree of height h is $2^{h+1} - 1 - (h+1)$.

Proof.

There are 2^h nodes at height 0. There are 2^{h-1} nodes at height 1, 2^{h-2} nodes at height 2, and so on, until there is a single node at height h. In general, there are 2^k nodes at height (h-k). Since the nodes at height 0 add nothing to the sum of the heights, the sum can be expressed as

$$S = \sum_{k=1}^{h} k2^{h-k}$$

While it is possible to solve for S in a straightforward approach, the following "trick" of the trade works well. Double S and you get

$$2S = \sum_{k=1}^{h} 2k2^{h-k} = \sum_{k=1}^{h} k2^{h-k+1}$$

Separate the low order k=1 term in 2S:

$$2S = 2^h + \sum_{k=2}^{h} k2^{h-k+1}$$

and the high-order k=h term in S:

$$S = \sum_{k=1}^{h-1} k2^{h-k} + h$$

Now subtract S from 2S:

$$S = 2S - S = 2^h + \sum_{k=2}^{h} k2^{h-k+1} - \sum_{k=1}^{h-1} k2^{h-k} - h$$

Since

$$\sum_{k=2}^{h} k2^{h-k+1}$$

is the same as

$$\sum_{k=1}^{h-1} (k+1)2^{h-(k+1)+1}$$

we can rewrite the previous equation as follows:

$$\begin{aligned} S &= 2^h + \sum_{k=1}^{h-1} (k+1)2^{h-(k+1)+1} - \sum_{k=1}^{h-1} k2^{h-k} - h \\ &= 2^h + \sum_{k=1}^{h-1} (k+1)2^{h-k} - \sum_{k=1}^{h-1} k2^{h-k} - h \\ &= 2^h + \sum_{k=1}^{h-1} ((k+1)2^{h-k} - k2^{h-k}) - h \end{aligned}$$

This last sum is easily reduced to

$$S = 2^h - h + \sum_{k=1}^{h-1}(k+1-k)2^{h-k}$$

$$= 2^h - h + \sum_{k=1}^{h-1}2^{h-k}$$

$$= 2^h - h + \sum_{k=1}^{h-1}2^{k}$$

$$= 2^h - h + (2^h - 1) - 1$$

$$= 2^{h+1} - 1 - (h+1)$$

Since n, the number of nodes in the tree in a complete binary tree of height h, is between 2^h and $2^{h+1} - 1$, in the worst case, n would be as little as 2^h and $2n = 2^{h+1}$ would be strictly greater than the number of moves in the worst case. Therefore, the running time for building a heap lies between n and 2n, clearly O(n).

Summary

A heap is a good data structure for priority queues. The time to build it is linear in the number of elements and the cost of inserting and deleteMin operations is O(log n), with very low overhead.

Chapter 7 Sorting

7.1 Introduction

Insertion sort is the sorting algorithm that splits an array into a sorted and an unsorted region, and repeatedly picks the lowest index element of the unsorted region and inserts it into the proper position in the sorted region, as shown in Figure 7.1.

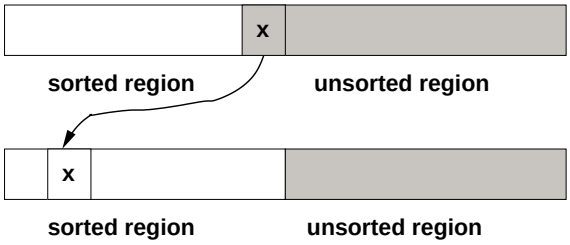


Figure 7.1: Insertion sort modeled as an array with a sorted and unsorted region. Each iteration moves the lowest-index value in the unsorted region into its position in the sorted region, which is initially of size 1.

The process starts at the second position and stops when the rightmost element has been inserted, thereby forcing the size of the unsorted region to zero.

Insertion sort belongs to a class of sorting algorithms that sort by comparing keys to adjacent keys and swapping the items until they end up in the correct position. Another sort like this is bubble sort. Both of these sorts move items very slowly through the array, forcing them to move one position at a time until they reach their final destination. It stands to reason that the number of data moves would be excessive for the work accomplished. After all, there must be smarter ways to put elements into the correct position.

The insertion sort algorithm is below.

```
for ( int i = 1; i < a.size(); i++) {
    tmp = a[i];
    for ( j = i; j >= 1 && tmp < a[j-1]; j = j-1)
        a[j] = a[j-1];
    a[j] = tmp;
}
```

You can verify that it does what I described. The step of assigning to tmp and then copying the value into its final position is a form of efficient swap. An ordinary swap takes three data moves; this reduces the swap to just one per item compared, plus the moves at the beginning and end of the loop.

The worst case number of comparisons and data moves is $O(N^2)$ for an array of N elements. The best case is $\Omega(N)$ data moves and$\Omega(N)$ comparisons. We can prove that the average number of comparisons and data moves is $\Omega(N^2)$.

7.2 A Lower Bound for Simple Sorting Algorithms

An inversion in an array is an ordered pair (i, j) such that $i < j$ but $a[i] > a[j]$. An exchange of adjacent elements removes one inversion from an array. Insertion sort needs to remove all inversions by swapping, so if there are m inversions, m swaps will be necessary.

Theorem 1. *The average number of inversions in an array of n distinct elements is n(n-1)/4.*

Proof. Let L be any list and L_R be its reverse. Pick any pair of elements $(i, j) \in L$ with $i < j$. There are $n(n-1)/2$ ways to pick such pairs. This pair is an inversion in exactly one of L or L_R. This implies that L and L_R combined have exactly $n(n-1)/2$ inversions. The set of all $n!$ permutations of n distinct elements can be divided into two disjoint sets containing lists and their reverses. In other words, half of all of these lists are the reverses of the others. This implies that the average number of inversions per list over the entire set of permutations is $n(n-1)/4$. □

Theorem 2. *Any algorithm that sorts by exchanging adjacent elements requires* $\Omega(n^2)$ *time on average.*

Proof. The average number of inversions is initially $n(n-1)/4$. Each swap reduces the number of inversions by 1, and an array is sorted if and only if it has 0 inversions, so $n(n-1)/4$ swaps are required. □

7.3 Shell Sort

Shell sort was invented by Donald Shell. It is like a sequence of insertion sorts carried out over varying distances in the array and has the advantage that in the early passes, it moves data items close to where they belong by swapping distant elements with each other. Consider the original insertion sort modified so that the gap between adjacent elements can be a number besides 1:

Listing 7.1: A Generalized Insertion Sort Algorithm

```
int gap = 1;
for ( int i = gap; i < a.size(); i++) {
    tmp = a[i];
    for ( j = i; j >= gap && tmp < a[j-gap]; j = j-gap)
        a[j] = a[j-gap];
    a[j] = tmp;
}
```

Now suppose we let the variable `gap` start with a large value and get smaller with successive passes. Each pass is a modified form of insertion sort on each of a set of interleaved sequences in the array. When the `gap` is h, it is an insertion sort of the h sequences

$0, h, 2h, 3h, \ldots, k_0 h,$

$1, 1+h, 1+2h, 1+3h, \ldots, 1+k_1 h$

$2, 2+h, 2+2h, 2+3h, \ldots, 2+k_2 h$

$\ldots$

$h-1, h-1+h, h-1+2h, \ldots, h-1+k_{h-1} h$

where k_i is the largest number such that $i + k_i h < n$. For example, if the array size is 15, and the gap is $h = 5$, then each of the following subsequences of indices in the array, which we will call ***slices***, will be insertion-sorted independently:

```
slice 0:  0    5    10
slice 1:  1    6    11
slice 2:  2    7    12
slice 3:  3    8    13
slice 4:  4    9    14
```

For each fixed value of the gap, h, the sort starts at array element $a[h]$ and inserts it in the lower sorted region of the slice, then it picks $a[h+1]$ and inserts it in the sorted region of its slice, and then $a[h+2]$ is inserted, and so on, until $a[n-1]$ is inserted into its slice's sorted region. For each element $a[i]$, the sorted region of the slice is the set of array elements at indices i, $i-h$, $i-2h$, ... and so on. When the gap is h, we say that the array has been ***h-sorted***. It is obviously not sorted, because the different slices can have different values, but each slice is sorted. Of course when the gap $h = 1$ the array is fully sorted. The following table shows an array of 15 elements before and after a 5-sort of the array.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Original Sequence:	81	94	11	96	12	35	17	95	28	58	41	75	15	65	7
After 5-sort:	35	17	11	28	7	41	75	15	65	12	81	94	95	96	58

The intuition is that the large initial gap sorts move items much closer to where they have to be, and then the successively smaller gaps move them closer to their final positions.

In Shell's original algorithm the sequence of gaps was $n/2$, $n/4$, $n/8$,..., 1. This proved to be a poor choice of gaps because the worst case running time was no better than ordinary insertion sort, as is proved below.

Listing 7.2: Shell's Original Algorithm

```
for ( int gap = a.size()/2; gap > 0; gap /=2)
    for ( int i = gap; i < a.size(); i++) {
        tmp = a[i];
        for ( j = i; j >= gap && tmp < a[j-gap]; j = j-gap)
            a[j] = a[j-gap];
        a[j] = tmp;
}
```

7.3.1 Analysis of Shell Sort Using Shell's Original Increments

Lemma 3. *The running time of Shell Sort when the increment is h_k is $O(n^2/h_k)$.*

Proof. When the increment is h_k, there are h_k insertion sorts of n/h_k keys. An insertion sort of m elements requires in the worst case $O(m^2)$ steps. Therefore, when the increment is h_k the total number of steps is

$$h_k \cdot O\left(\frac{n^2}{h_k^2}\right) = O\left(\frac{n^2}{h_k}\right)$$

□

Theorem 4. *The worst case for Shell Sort, using Shell's original increment sequence, is* $\Theta(n^2)$.

Proof. The proof has two parts, one that the running time has a lower bound that is asymptotic to n^2, and one that it has an upper bound of n^2.

Proof of Lower Bound.

Consider any array of size $n = 2^m$, where m may be any positive integer. There are infinitely many of these, so this will establish asymptotic behavior. Let the $n/2$ largest numbers be in the odd-numbered positions of the array and let the $n/2$ smallest number be in the even numbered positions. When n is a power of 2, halving the gap, which is initially n, in each pass results in an even number. Therefore, in all passes of the algorithm until the very last pass, the gaps are even numbers, which implies that all of the smallest numbers must remain in even-numbered positions and all of the largest numbers will remain in the odd-numbered positions. The j^{th} smallest number is in position $2j-1$ and must be moved to position j when $h = 1$. Therefore, this number must be moved$(2j-1) - j = j - 1$ positions. Moving all $n/2$ smallest numbers to their correct positions when $h = 1$ requires

$$\sum_{j=1}^{n/2}(j-1) = \sum_{j=0}^{(n/2)-1} j = \frac{(n/2-1)(n/2)}{2} = \Theta(n^2)$$

steps. This proves that the running time is at least $\Theta(n^2)$.

Proof of Upper Bound.

By Lemma 3, the running time of the pass when the increment is h_k is $O(n^2/h_k)$. Suppose there are t passes of the sort. If we sum over all passes, we have for the total running time,

$$O\left(\sum_{k=1}^{t}\frac{n^2}{h_k}\right) = O\left(n^2\sum_{k=1}^{t}\frac{1}{h_k}\right) = O\left(n^2\sum_{k=1}^{t}\frac{1}{2^k}\right) = O(n^2)$$

because, as we proved in Chapter 1,

$$\sum_{k=1}^{t}\frac{1}{2^k} = 2 - \frac{1}{2^{t-1}} - \frac{t}{2^t} < 2$$

and this shows that n^2 is an asymptotic upper bound as well. It follows that the worst case is $\Theta(n^2)$. □

7.3.2 Other Increment Schemes

Better choices of gap sequences were found afterwards. Before looking at some of the good ones, consider an example using a simple sequence such as 5, 3, 1.

Example

This uses the three gaps, 5, 3, 1 on an arbitrary array whose initial state is on the first row in the table below.

	0	1	2	3	4	5	6	7	8	9	10	11	12
Original Sequence:	81	94	11	96	12	35	17	95	28	58	41	75	15
After 5-sort:	35	17	11	28	12	41	75	15	96	58	81	94	95
After 3-sort:	28	12	11	35	15	41	58	17	94	75	81	96	95
After 1-sort:	11	12	15	17	28	35	41	58	75	81	94	95	96

Hibbard proposed using the sequence

$$h_1 = 1,\ h_2 = 3,\ h_3 = 7,\ h_4 = 15, \ldots,\ h_t = 2^t - 1$$

which is simply the numbers one les than powers of 2. Such a sequence, 1, 3, 7, 15, 31, 63, 127,..., has the property that no two adjacent terms have a common factor: if they had a common factor p, then their difference would also be divisible by p. But the difference between successive terms is $(2^{k+1}–1)–(2^k–1) = (2^{k+1}–2^k) = 2^k$ which is divisible by powers of 2 alone. Hence the only number that can divide successive terms is a power of 2. Since all of the terms are odd numbers, none are divisible by 2, so they have no common factor.

Successive terms are related by the recurrence

$$h_{k+1} = 2h_k + 1$$

Sedgewick proposed a few different sequences, such as this one, defined recursively:

$$h_1 = 1,\ h_{k+1} = 3h_k + 1$$

which generates the sequence

$$1, 4, 13, 40, 121, 364, \ldots$$

Hibbard's increment sequence turns out to be an improvement over Shell's original increment sequence.

Theorem 5. *The worst case running time of Shell Sort using Hibbard's sequence,* $h_1 = 1$, $h_2 = 3$, $h_3 = 7$, $h_4 = 15$, ..., $h_t = 2^t - 1$, *is* $\Theta(n^{3/2})$.

Proof. The upper bound alone is proved here. Lemma 3 establishes that the running time of a pass when the increment is $h_k$is $O(n^2/h_k)$. We will use this fact for all increments h_k such that $h_k > n^{1/2}$. This is not a tight enough upper bound for the smaller increments, and we need to work harder to get a tighter bound for them. The general idea will be to show that by the time the increments are that small, most elements do not need to be moved very far.

So we turn to the case when $h_k \leq n^{1/2}$. By the time that the array is $h_k - sorted$, it has already been h_{k+1}-sorted and h_{k+2}-sorted. Now consider the array elements at positions p and $p - i$ for all $i \leq p$. If i is a multiple of h_{k+1} or h_{k+2}, then $a[p - i] < a[p]$ because these two elements were part of the same slice, either for increment h_{k+1} or increment h_{k+2}, and so they were sorted with respect to each other. More generally, suppose that i is a non-negative linear combination of h_{k+1} and h_{k+2}, i.e., $i = c_1 h_{k+1} + c_2 h_{k+2}$, for some non-negative integers c_1 and c_2. Then

$$p - i = p - (c_1 h_{k+1} + c_2 h_{k+2}) = p - c_1 h_{k+1} - c_2 h_{k+2}$$

and

$$(p-i) + c_2 h_{k+2} = p - c_1 h_{k+1}$$

which implies that after the h_{k+2}-sort, $a[p-i] < a[p-c_1 h_{k+1}]$ because they are part of the same h_{k+2}-slice. Similarly, after the h_{k+1}-sort, $a[p-c_1 h_{k+1}] < a[p]$ because they are part of the same h_{k+1}-slice and $p - c_1 h_{k+1} < p$. Thus, $a[p-i] < a[p-c_1 h_{k+1}] < a[p]$.

Since $h_{k+2} = 2h_{k+1}$ by the definition of Shell's increment scheme, h_{k+2} and h_{k+1} are relatively prime. (If not, they have a common factor >1 and so their difference $h_{k+2} - h_{k+1} = 2h_{k+1} + 1 - h_{k+1} = h_{k+1} + 1$ would have a common factor with each of them, and in particular h_{k+1} and $h_{k+1} + 1$ would have a common factor, which is impossible.) Because h_{k+2} and h_{k+1} are relatively prime, their greatest common divisor (gcd) is 1. An established theorem of number theory is that if $c = \gcd(x, y)$ then there are integers a and b such that $c = ax + by$. When x and y are relatively prime, this implies that there exist a, b such that $1 = ax + by$, which further implies that every integer can be expressed as a linear combination of x and y. A stronger result is that all integers at least as large as $(x-1)(y-1)$ can be expressed as non-negative linear combinations of x and y. Let

$$\begin{aligned} m &\geq (h_{k+2} - 1)(h_{k+1} - 1) \\ &= (2h_{k+1} + 1 - 1)(2h_k + 1 - 1) \\ &= 2(h_{k+1})(2h_k) \\ &= 4h_k(2h_k + 1) = 8h_k^2 + 4h_k \end{aligned}$$

.

By the preceding statement, m can be expressed in the form $c_1 h_{k+1} + c_2 h_{k+2}$ for non-negative integers c_1 and c_2. From the preceding discussion, we can also conclude that $a[p-m] < a[p]$. What does this mean? For any index $i > 8h_k^2 + 4h_k$, $a[p-i] < a[p]$. During the h_k-sort, no element has to be moved more than $8h_k - 4$ times (it moves h_k positions each time.) There are at most $n - h_k$ such positions, and so the total number of comparisons in this pass is $O(nh_k)$.

How many increments h_k are smaller than $n^{1/2}$? Roughly half of them, because if we approximate n as a power of 2, say 2^t, then $\sqrt{n} = 2^{t/2}$ and the increments 1, 3, 7, ..., $2^{t/2} - 1$ are half of the total number of increments. For simplicity, assume t is an even number. Then the total running time of Shell Sort is

$$\begin{aligned} & O\left(\sum_{k=1}^{t/2} nh_k + \sum_{k=t/2+1}^{t} \frac{n^2}{h_k}\right) \\ = \; & O\left(n\sum_{k=1}^{t/2} h_k + n^2 \sum_{k=t/2+1}^{t} \frac{1}{h_k}\right) \end{aligned}$$

Since the first sum is a geometric series with an upper term of $h_{t/2} = \Theta(n^{1/2})$, it is bounded by a constant times $n^{1/2}$and so it is $O(n \cdot n^{1/2}) = O(n^{3/2})$. Since $h_{t/2+1} = \Theta(n^{1/2})$, the second sum can be approximated as $O(n^2 \cdot (1/n^{1/2}) = O(n^{3/2})$. This shows that the upper bound in the worst case running time using Hibbard's sequence is $O(n^{3/2})$. That there is a sequence that achieves it is an exercise. □

Many other sequences have been studied. Studies have shown that the sequence defined by

$$(h_i, h_{i+1}) = (9 \cdot (4^{i-1} - 2^{i-1}) + 1,\ 4^{i+1} - 6 \cdot 2^i + 1)\ \ i = 1, 2, \ldots$$

which generates 1, , 5, 19, 41, 109, ... has $O(n^{4/3})$ worst case running time (Sedgewick, 1986). The notation above means that the formula generates pairs of gaps.

Example

This shows how Shell sort works on an array using the gap sequence 13,4,1:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
After 13-sort	A	E	O	R	T	I	N	G	E	X	A	M	P	L	S
After 4-sort	A	E	A	G	E	I	N	M	P	L	O	R	T	X	S
After 1-sort	A	A	E	E	G	I	L	M	N	O	P	R	S	T	X

7.4 Heap Sort

The idea underlying heap is to create a ***maximum heap*** in $O(n)$ steps. A maximum heap is a heap in which the heap -order property is reversed – each node is larger than its children. Then repeatedly swap the maximum element with the one at the end of the array, shortening the array by one element afterwards, so that the next swap is with the preceding element.

7.4.1 The Algorithm

There are less efficient ways to do this. This algorithm builds the max-heap, and then pretends top delete the maximum element by swapping it with the element in the last position, and decrementing the variable that defines wehere the heap ends in the array, effectively treating the largest element as not being part of the heap anymore.

```
// First build the heap using the same algorithm described in
   Chapter 6.
// Assume the array is named A and A.size() is its length.
int N = A.size();
for (int i = N/2; i > 0; i--)
    // percolate down in a max heap stopping if we reach N
    PercolateDown (A, i, N );

// A is now a heap
// Now repeatedly swap the max element with the last element in
   the heap
for ( int j = N - 1; j > 0; j--) {
    swap(A[0], A[j]);
    PercolateDown(A, 0, j);
}
```

The `PercolateDown` function must be changed slightly from the one described in *Chapter 6, Priority Queues.* because it needs to prevent an element from being dropped "too far down" the heap. It therefore has a parameter that acts like an upper bound on the heap.

```
void PercolateDown( vector<Comparable> & array,  int hole, int last )
{
    int        child;
    Comparable tmp = array[ hole ];

    while ( hole * 2 <= last-1 ) {
        child = hole * 2;
        if ( child != last-1 && array[child + 1] < array[child])
            child++;
        if( array[ child ] < tmp )
            array[ hole ] = array[ child ];
        else
            break;
        hole = child;
    }
    array[ hole ] = tmp;
}
```

7.4.2 Analysis

In Chapter 6 we proved that building the heap takes $O(n)$ steps (to be precise, $2n$ steps.) In the second stage of the algorithm, the swap takes constant time, but the `PercolateDown` step, in the worst case, will require a number of comparisons and moves in proportion to the current height of the heap. The j^{th} iteration uses at most $2 \cdot \log(N - j + 1)$ comparisons. We have

$$\sum_{j=1}^{N-1} \log(N - j + 1) = \sum_{j=2}^{N} \log(j) = \log N!$$

It can be proved that $\log(N!) = \Theta(N \log \mathrm{N})$, so heapsort in the worst case has a running time that is $\Theta(N \log N)$. A more accurate analysis will show that the most number of comparisons is $2N \log N - O(N)$.

Experiments have shown that heapsort is consistent and averages only slightly fewer comparisons than its worst case. The following theorem has been proved, but the proof is omitted here.

Theorem 6. *The average number of comparisons performed to heapsort a random permutation of n distinct items is $2n \log n$–$O(n \log \log n)$.*

7.5 Quicksort

Quicksort is the fastest known sorting algorithm when implemented correctly, meaning that the non-recursive version should be used and the longer partition should be stacked rather than the shorter one. Quicksort has an average running time of $\Theta(N \log N)$ but a worst case performance of $O(N^2)$. The reason that quicksort is considered the fastest algorithm for sorting when used carefully, is that a good design can make the probability of achieving the worst case negligible.

Here we review the algorithm and analyze its performance.

7.5.1 The Algorithm

Basic idea:

Let S represent the set of elements to be sorted, i.e., S is an array. Let quicksort*(S)* be defined as the following sequence of steps:

1. If the number of elements in S is 0 or 1, then return, because it is already sorted.
2. Pick any element v in S. Call this element v the ***pivot***.
3. Partition S–v) into two disjoint sets $S_1 = \{x \in S \mid x \leq v\}$ and $S_2 = \{x \in S \mid x \geq v\}$.
4. Return quicksort(S_1) followed by v followed by quicksort(S_2).

Picking a Pivot

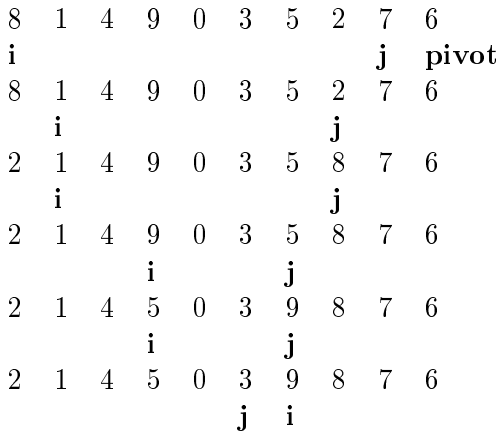
A poor choice of pivot will cause one set to be very small and the other to be very large. The result will be that the algorithm achieves its $O(n^2)$ worst case behavior. If the pivot is smaller than all other keys or larger than all other keys this will happen. We want a pivot that is the median without the trouble of finding the median. One could pick the pivot randomly, but then there is no gaurantee about what it will be and the cost of the random number generation buys little in the end.

A good compromise between safety and performance is to pick three elements and take their median. Doing this almost assuredly eliminates the possibility of quadratic behavior. A good choice is the first, last, and middle elements of the array.

Partitioning

The best way to partition is to push the pivot element out of the way by placing it at the beginning or end of the array. Having done that, all of the implementations do basically the same thing, except for how they handle elements equal to the pivot.

The general idea is to advance `i` and `j` pointers towards the middle swapping elements larger than the pivot until `i` and `j` cross. The following example shows an array with the initial placements of the `i` and `j` pointers and the pivot, after it has been moved to the last position in the array. In each step, first `j` travels down the array looking for a culprit that doesn't belong, and then `i` travels up the array doing the same thing. When they each stop, the elements are swapped and they each advance one element.



It stops at this point and the pivot is swapped, so the final array is

2 1 4 5 0 3 6 8 7 9

The real issue is handling elements that are equal to the pivot. The safest way to handle elements equal to the pivot is to swap them as if they were smaller or larger, otherwise the algorithm can degrade to the worst case.

A recursive version of quicksort that uses this strategy is in the listing below.

```
void quicksort( vector<C> & a, int left, int right)
{
    if ( left + 10 <= right ) {
        // pick the pivot and swap it into a[right-1]
        C pivot = median3(a, left, right);

        // partition step
        int i = left;
        int j = right-1;
        while ( true )  {
            while ( a[++i] < pivot ) { }
            while ( pivot < a[--j] ) { }
            if ( i < j )
                swap( a[i], a[j] );
            else
                break;
        }
        // move pivot into position
        swap( a[i], a[right - 1] );

        // recursively sort each set
        quicksort(a, left, i-1);
        quicksort(a,i+1, right);
    }
    else // the array is too small to quicksort - use insertionsort
        instead
        insertionsort(a, left, right);
```

```
        // insertionsort must have two extra parameters to receive left
           and right bounds
}
```

Notes.

1. That `i` starts at `left+1` is intentional. The `median3` function rearranges the array so that `a[left]` $\leq$ `pivot`, so it is not necessary to examine it.
2. A symmetric statement is true of `j`: `median3` places the pivot in `a[right-1]` so `j` can start by comparing `pivot` to `a[right-2]`.
3. Neither `i` nor `j` can exceed the array bounds because of the above reasoning. The ends act as sentinels for `i` and `j`.

7.5.2 Analysis of Quicksort

A recurrence relation describes the total number of comparisons. Letting $T(n)$ denote the number of comparisons with an array of size n, and let i denote the number of elements in the lower part of the partition, S_1. $T(1)$ is some constant, say a. Then

$$T(n) = T(i) + T(n - i–1) + cn$$

Worst Case

The worst case is when the partition always creates one array of size 0 and the other of size $n - 1$. The recurrence in this case is

$$T(n) = T(n - 1) + cn$$

for $n > 1$, which implies that

$$\begin{aligned} T(n) &= T(n-2) + cn + c(n-1) \\ &= T(n-3) + c(n + n - 1 + n - 2) \\ &= \cdots \\ &= T(1) + c(n + n - 1 + n - 2 + \ldots + 1) \\ &= a + \frac{cn(n+1)}{2} = \Theta(n^2) \end{aligned}$$

Average Case

The average case analysis assumes that all sizes for S_1 are equally likely. We now let $T(n)$ represent the average running time for an array of size n. It follows that $T(n)$ is the average of the running

times for all possible sizes of S_1, which are each of probability $1/n$.

$$\begin{aligned} T(n) &= \frac{1}{n}\sum_{j=0}^{n-1}(T(j)+T(n-j-1))+cn \\ &= \frac{2}{n}\sum_{j=0}^{n-1}T(j)+cn \qquad iff \\ nT(n) &= 2\sum_{j=0}^{n-1}T(j)+cn^2 \qquad iff \\ (n-1)T(n-1) &= 2\sum_{j=0}^{n-2}T(j)+c(n-1)^2 \qquad iff \end{aligned}$$

Now subtract the last two equalities and get

$$\begin{aligned} nT(n)-(n-1)T(n-1) &= 2T(n-1)+c(n^2-(n-1)^2) \\ &= 2T(n-1)+2cn-c \end{aligned}$$

Rearranging and dropping the constant c,

$$nT(n) = (n+1)T(n-1)+2cn$$

Dividing by $n(n+1)$, we get a formula that can be telescoped by successive adding:

$$\begin{aligned} \frac{T(n)}{n+1} &= \frac{T(n-1)}{n}+\frac{2c}{n+1} \\ \frac{T(n-1)}{n} &= \frac{T(n-2)}{n-1}+\frac{2c}{n} \\ &\vdots \\ \frac{T(2)}{3} &= \frac{T(1)}{2}+\frac{2c}{3} \end{aligned}$$

Adding these equations gives

$$\frac{T(n)}{n+1} = \frac{T(1)}{2}+2c\sum_{i=3}^{n+1}\frac{1}{i} \approx \frac{T(1)}{2}+2c\log(n+1) = O(\log n)$$

so that

$$T(n) = O(n\log n)$$

7.6 A Lower Bound for Sorting

The question is, can we ever find a sorting algorithm better than the $O(n\log n)$ algorithms we have seen so far, or is it theoretically impossible to achieve this?

The answer is that if our sorting algorithm sorts by making binary comparisons, then the number of comparisons is $\Theta(n\log n)$. The proof is based on the following argument:

Any sorting algorithm that uses only comparisons requires $\lceil\log(n!)\rceil$ comparisons in the worst case and $\log(n!)$ on average. To prove this we use a decision tree.

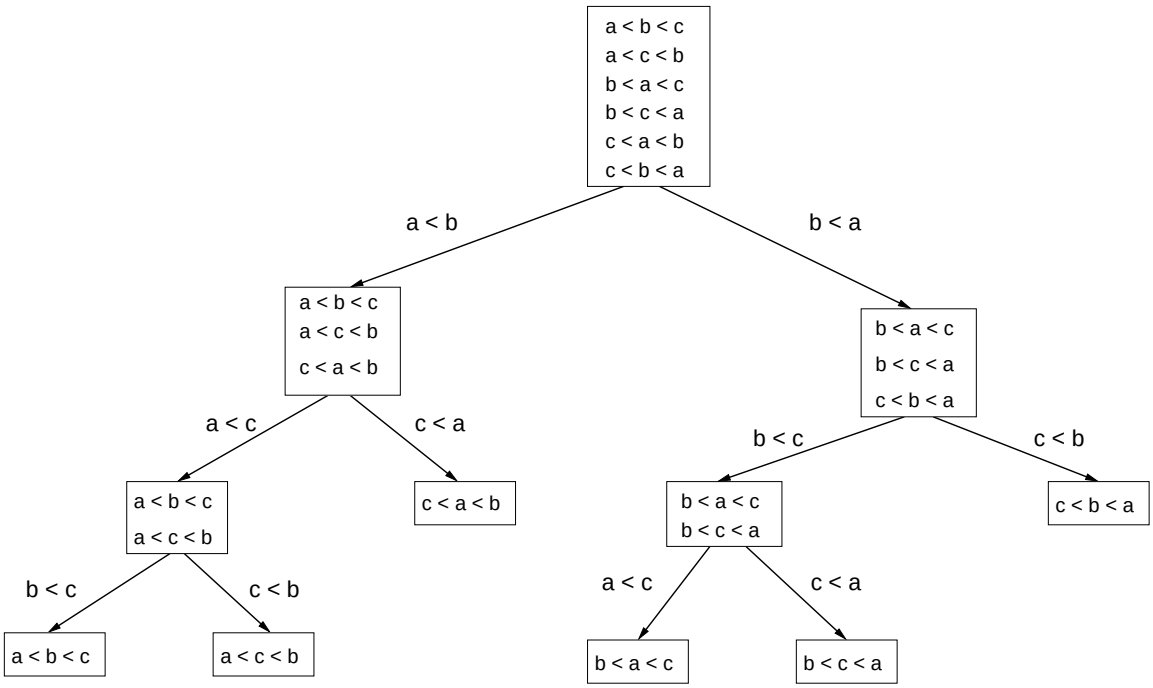


Figure 7.2: A decision tree that represents the decisions to sort three numbers.

7.6.1 Decision Trees

A ***decision tree*** is a tree representation of an algorithm that solves a problem by a sequence of successive decisions. In a decision tree, the nodes represent logical assertions and an edge from a node to a child node is labeled by a proposition. A binary decision tree is based on binary decisions. Because a binary decision tree is a binary tree, we can use reasoning about binary trees to arrive at a theorem about algorithms that sort using binary comparisons. Figure 7.2 illustrates a decision tree that represents the comparisons that would sort three distinct numbers. We need some preliminary lemmas.

Lemma 7. *A binary tree of height d has at most 2^d leaves.*

Proof. We can prove this by induction on the height of the tree. If $d = 0$, the tree consists of a root, and it has $1 = 2^0$ leaves. If $d > 0$, assume it is true for $d - 1$. The tree must have a root and two subtrees of height at most $d - 1$, which by assumption have at most 2^{d-1} leaves each. Since the root is not a leaf, the tree has at most $2 \cdot 2^{d-1} = 2^d$ leaves. □

Lemma 8. *If a binary tree has n leaves then it must have height at least $\lceil \log n \rceil$.*

Proof. Suppose a tree with n leaves has height less than $\lceil \log n \rceil$. If n is not a power of 2, its height is at most $\lfloor \log n \rfloor$ and if n is a power of 2 then it is at most $\log(n) - 1$. In either case, let d be the height. From the above lemma, the tree has at most 2^d leaves. But $2^{\log n - 1} < n$ and $2^{\lfloor \log n \rfloor} < n$ which contradicts the fact that the tree has n leaves. □

Lemma 9. *Any sorting algorithm that uses only binary comparisons between elements requires at least $\lceil \log n! \rceil$ comparisons for an array of size n.*

Proof. There are $n!$ leaves in a decision tree to sort n elements because there are $n!$ different possible outcomes. By Lemma 8, the height of this tree is at least $\lceil \log n! \rceil$. □

Theorem 10. *Any sorting algorithm that uses only comparisons between elements requires* $\Omega(n \log n)$ *comparisons.*

Proof. We show that $\log(n!)$ is $\Omega(n \log n)$.

$$\begin{aligned}
\log(n!) &= \log n + \log(n-1) + \log(n-2) + \cdots \log 2 + \log 1 \\
&\geq \log n + \log(n-1) + \log(n-2) + \cdots \log(n/2) \\
&\geq \frac{n}{2} \cdot \log \frac{n}{2} \\
&\geq \frac{n}{2} \cdot (\log n - 1) \\
&\geq \frac{n}{2} \cdot \log n - \frac{n}{2} \\
&= \Omega(n \log n)
\end{aligned}$$

□

Chapter 8 Disjoint Sets and the Union/Find Problem

Equivalence Relations

A binary relation ***R*** on a set ***S*** is a subset of the Cartesian product ***S*** × ***S***. If $(a,b) \in \boldsymbol{R}$ we write $a\boldsymbol{R}b$ and say *a* relates to *b*. Relations can have many properties. An equivalence relation is a symmetric, reflexive, and transitive binary relation. All equivalence relations are isomorphic to the equality relation, and so equivalence may be thought of like the equality relation.

An equivalence relation creates a partition of the set ***S*** into subsets $S_1, S_2, \ldots, S_n$ such that each S_i is the transitive closure of the set of items equivalent to a member of ***S.*** Conversely, any partition defines the equivalence R defined by $a\boldsymbol{R}b$ iff *a* and *b* are in the same subset.

Examples of equivalence relations:

- Two fractions are equivalent if they reduce to the same irreducible fraction.
- For any positive number n, the relationship p ~ q iff n divides (p-q) is an equivalence relation.
- Two cities are equivalent if it is possible to walk from one to the other without crossing a bridge.
- Two people are in the same clique if they know each other directly or if each of them knows someone who is in the clique. The world is divided into cliques. There is a conjecture that the whole world consists of a single clique in which the separation is at most six edges.
- Two cells in a maze are equivalent if there is a path from one to the other.
- Two stars are equivalent if they lie in the same galaxy.

Dynamic Equivalence Problem

A problem arising in many application areas is the *dynamic equivalence* problem. Given two members of a set ***S***, we need to know whether they are equivalent. One of the earliest instances of this problem arose in the development of the first compiler, the Fortran compiler. Fortran has an `EQUIVALENCE` declaration that tells the compiler that two or more entities share the same logical storage locations. As the compiler processes the source code, it needs to construct the sets of equivalent variables. The problem is dynamic because each `EQUIVALENCE` statement is, in effect, a union operation, combining two or more disjoint and inequivalent sets of variables into a single set. As it processes the code, the compiler also needs to identify which variables are sharing the same locations, which means deciding whether two variables are in the same equivalence class.

If ***S*** has n elements, we could create an n by n Boolean array to represent this information, by setting ***R***[i][j] = True *iff* $(i,j) \in \boldsymbol{R}$. That solves only half the problem. We also need to change the relationships dynamically so that, for example, we add the relation a***R***b or remove the relation a***R***b. Again, the Boolean matrix will provide a fast solution, since we can just turn off the appropriate bits.

Now add the requirement that we should be able to find the set of all elements that are related to a member by obtaining the name of the set to which it belongs and displaying all members of that set. The Boolean matrix no longer provides a fast solution, since we will need to take the transitive closure, at best super-quadratic running time. Alternatively when we add a new relation a***R***b, we could complete the matrix so that it is transitive, but that turns that operation into an *O(N)* operation.

A more efficient solution is to represent the equivalence classes as disjoint sets and arbitrarily choose one member of the set as the name of the set. In the dynamic equivalence problem, there are only two operations that must be defined and implemented on this collection of sets: `find` and `union`. Given an element x of ***S***, `find(x)` returns the name of the set containing x. Given the names of two sets x and y in ***S***, `union(x,y)` forms the set union of x and y. The reason that union is important is that union can add new relations to ***S***: if we want to add a***R***b, we form `union(find(a), find(b))` if a and b are in different sets to start.

If an algorithm can get to see the entire sequence of union and find operations before it processes any of them, as if it were reading them from a file and storing and analyzing them first, it is called an offline algorithm. If an algorithm does not have this opportunity, and must process each instruction immediately when it sees it, it is called an online algorithm. This would be analogous to interactive input, in which a user types the sequence of instructions at a terminal and the algorithm must respond immediately to each entered instruction. The problem we solve in this chapter is the ***online, disjoint set union/find problem***, which is formalized as follows:

Union/Find Problem: Given N disjoint, singleton sets {0}, {1}, {2}, …, {N-1}, process a sequence of `find()` and `union()` operations ***on-line***, meaning one after the other. The names of the elements are arbitrary; although they are numbers they have no numeric properties, and we could just as easily name them a,b,…,z except that there are just 26 letters but infinitely many integers.

A Naive Solution

Maintain an array `S` such that `S[i]` is the name of the set to which element `i` belongs. `find()` will be *O(1)* but `union()` will be *O(N)*. To form `union(a,b)` we would find the names of the sets to which `a` and `b` belong. Suppose `a` is in set `A` and `b` is in set `B`. Scan down the array changing all `A`s to `B`s. A sequence of *N-1* unions would take $O(N^2)$ steps. Some gain in performance can be achieved by maintaining the size of each set and when performing a `union`, always renaming the elements of the smaller set instead of the larger one. For example, if set `A` has 10 elements and set `B` has 5 elements, then when `union(a,b)` is processed, every element of `B` would be changed to belong to `A`. This implies that no element can have its set changed more than *log N* times, because each time its set is changed, it is part of a set that is at least double the size of the one it was in before the operation. As there are *N* elements, there are at most *log N* doublings that can take place. (There are no operations that undo the unions.) Therefore a sequence of *N* unions takes at most *O(N log N)* time, and a sequence of *M* finds and *N-1* unions take *O(M + N log N)* time.

An Efficient Solution

In this approach, the `find()` will take more time and union will be constant, but the total amortized time for N-1 unions and M finds will be slightly more than O(M+N).

Parent Trees

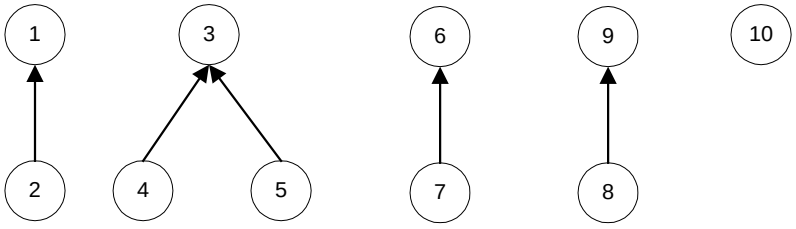
We represent a set by a ***parent tree***, in which the direction of edges is *towards the root* rather than towards the leaves.

The root of the tree is the name of the set containing all elements in the tree.

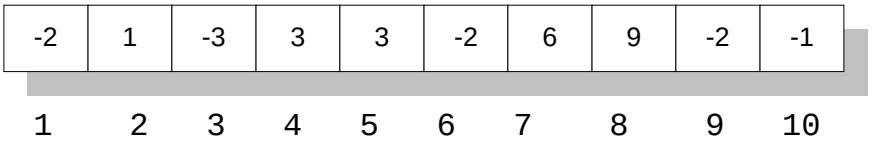
Since the name of the containing set is the only information we need to maintain for each node, and the root is the name, we can use an array to represent a tree; If `s[]` is the array, then `s[x]` is the index of the parent of `x` in the tree of which x is a member.
The root `s[j]`of a tree will have `s[j]` = -1 to indicate that it has no parent.

A `find(x)` operation will require traversing the path from `x` to the root of the tree. If the tree is maintained in an inefficient way, `find(x)` could be *O(N)*, but a very clever solution makes the running time much less.

The collection of sets {1,2}, {3,4,5}, {6,7,}, {8,9}, {10} is represented by the following forest:



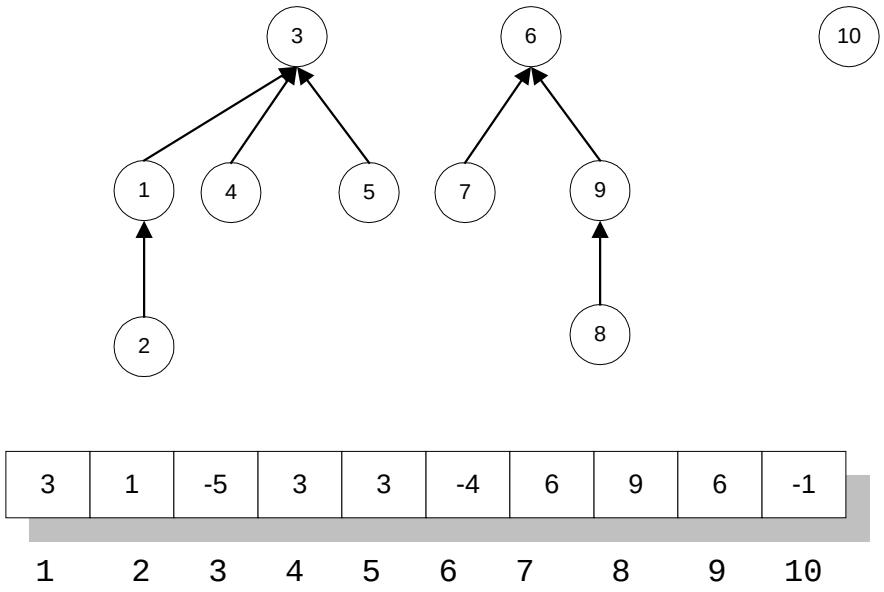
The corresponding array would be



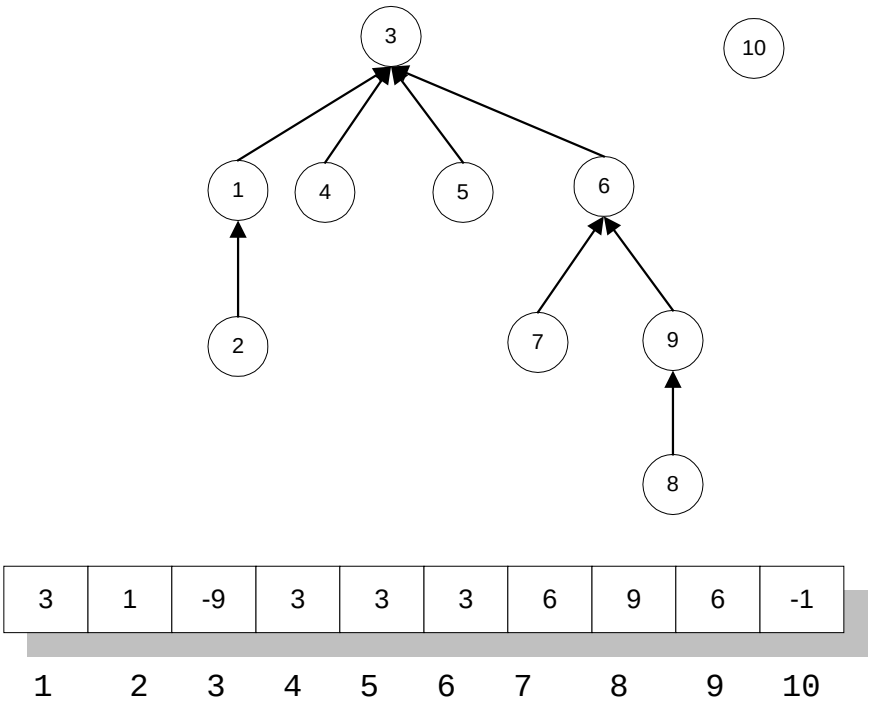
Smart Union

The naïve union operation will simply take the two trees and make one the child of the root of the other. A smarter solution makes the tree with fewer nodes the child of the larger tree. This is called ***union-by-size***. An alternative is to make the shorter tree the child of the deeper tree, which is called ***union-by-height***. Union-by-height is a slight modification of union-by-size.

I present the union-by-size algorithm. In the case that the two sets are the same size, either can be made the child of the other, so some fixed rule can be used. For example, after the smart union-by-size of the trees with roots 1 and 3, followed by the union of the trees rooted at 6 and 9, the resulting forest and array would look like



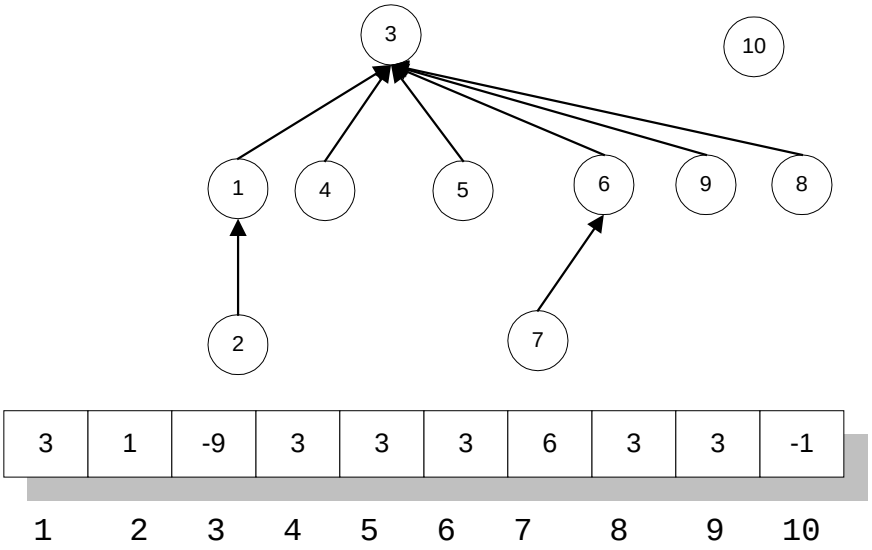
because the tree rooted at 3 has 5 nodes (-5) and node 1 now has node 3 as a parent. The union of the 3 tree and the 6 tree results in



It is very trivial to implement either algorithm. The extra work to do this transforms the solution into an extremely efficient one, as we will see soon.

Path Compression

In most abstract data types, a clean line is drawn between accessors and mutators; operations that access data do not change the state of the object. For example, the find operation on search trees and hash tables does not modify those objects. Although find is an accessor, a significant improvement in running time can be achieved by allowing it to change the object. Robert Tarjan realized this when he invented the ***path compression*** algorithm for parent trees. When path compression is employed in the find algorithm, all of the nodes that are visited on the path from the node to the root are turned into children of the root. Thus, after the call find(8) on the set above, the forest will look like



It is easy to implement these algorithms. Smart union is very easy. find() with path compression uses a clever bit of recursive coding to avoid the need for a stack. It is a little slower than a non-recursive algorithm because of the function call overhead. The find algorithm would be different if union-by-height were used because it would have to recalculate the height of the tree. That is why it is easier to use union-by-size.

Source Code

```
class DisjSets
{
public:
        DisjSets( int);
        void union( int, int);
        int find(int);
private:
        vector<int> s;
};

/* Construct the disjoint sets object.
 *  numElements is the initial number of disjoint sets. */
```

```
DisjSets::DisjSets( int numElements ) : s( numElements )
{
        for( int i = 0; i < s.size( ); i++ )
          s[ i ] = -1;
 }

 /** Union two disjoint sets.
  * For simplicity, we assume root1 and root2 are distinct
  * and represent set names.
  * root1 is the root of set 1.
  * root2 is the root of set 2.
  */
  void DisjSets::union( int root1,  int root2 )
  {
        if (root1 != root2) {
                if ( s[ root2 ] < s[ root1 ] )  {
                        // root2 is deeper
                        s[root2] += s[root1];
                        s[root1] = root2;
                } else {
                        // root1 is deeper
                        s[root1] += s[root2];
                        s[root2] = root1;
                }
        }
}
  /**
  * Perform a recursive find with path compression.
  * Error checks omitted again for simplicity.
  * Return the set containing x.
  */
    int DisjSets::find( int x )
    {
      if( s[ x ] < 0 )
         return x;
      else
         return s[ x ] = find( s[ x ] );
    }
```

This version of `find()` makes a recursive all when `x` is not the root of its tree (`s[x]< 0`). In this case, `find` is called with the parent of `x`, which is `s[x],` moving it one step closer to the root of the tree. The path compression takes place in the `return` statement, which assigns to `s[x]` the return value of the recursive call, which is the root of the tree to which `x` belongs. Thus, all nodes of the parent tree on the way from `x` to the root are made children of the root of that tree.

Analysis

Theorem. The running time of a sequence of M unions and finds using path compression and union by size, in the worst case, on a collection of N sets is $O(M\alpha(N))$, where $\alpha(n)$ is the inverse of Ackermann's function.

Ackermann's function is defined by

$A(0, n) = n + 1$
$A(m+1, 0) = A(m, 1)$
$A(m+1, n+1) = A(m, A(m+1, n))$

but Weiss uses an alternate definition of it that grows faster:

$A(1, n) = 2^n$
$A(m, 1) = A(m-1, 2)$
$A(m, n) = A(m-1, A(m, n-1))$

I will stick with the original definition. $\alpha(n)$ is the inverse of $A'(m) = A(m,m)$. I.e., $\alpha(n)$ is the value of m such that $A'(m) = n$. To give an example of how quickly $A(m,n,)$ grows:

$A(0,n) = n+1$

$A(1,n) = n+2$

$A(2,n) = 2n+3$

$A(3,n) = 2^{n+3} - 3$

$A(4,n) = 2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}} - 3$ where the tower of 2's is repeated n+3 times.

The following table shows the values of A(m,n) for the first seven values of m and first 6 values of n.

$A(m,n)$	n=0	n=1	n=2	n=3	n=4	n=5
m=0	1	2	3	4	5	6
m=1	2	3	4	5	6	7
m=2	3	5	7	9	11	13
m=3	5	13	29	61	125	253
m=4	13	65533	2^{65536}-3	2^{265536}-3	$A(3,2^{265536}$-3)	$A(3,A(4,4))$
m=5	65533	$A(4,65533)$	$A(4,A(4,65533))$	$A(4,A(5,2))$	$A(4,A(5,3))$	$A(4,A(5,4))$
m=6	$A(4,65533)$	$A(5,A(4,65533))$	$A(5,A(6,1))$	$A(5,A(6,2))$	$A(5,A(6,3))$	$A(5,A(6,4))$

A(m,m) is the diagonal through this table. You can see that A(4,4) = $A(3,2^{265536}$-3) which is orders of magnitude greater than 2^{265536}-3. Since 2^{265536}-3 is larger than any number you will

encounter in this universe, for practical purposes, $\alpha(n)<4$ for almost all n, and so it is essentially a constant function.

I do not include a proof of the theorem, which is quite lengthy. Because $\alpha(n)$ is essentially a constant, the theorem implies that a sequence of M unions and finds takes $O(M)$ time.

An Application

One simple application of the disjoint set union/find problem is the generation of mazes. Imagine a rectangular M by N grid `G` in which walls surround all cells, including the perimeter cells. The equivalence relation is that two cells are in the same set if there is a path from one cell to the other in the grid. Initially every cell is in a set by itself, as all cells are surrounded by walls, so there are MN disjoint sets. If a wall is removed between cells `G[i,j]` and `G[i+1,j]` then these two cells are now part of the same set. This is a union operation. As walls are removed, the sizes of the sets increase.

One can generate a random maze by choosing a wall to remove randomly. By repeatedly removing a random wall until the entrance cell and the exit cell are connected, a maze is formed with at least one path from the entrance to the exit. By repeating until all cells are in the same ser, meaning there is a path from any cell to any other cell, a more challenging maze is generated.

Chapter 9 Graph Algorithms

Graph Basics

Definitions

A ***graph*** G = (V,E) is a set of ***vertices*** V and a set of ***edges*** E. Each edge e in E is a pair (v,w) where v ∈ V and w ∈ V. (Formally, $E \subseteq V \times V$, the Cartesian product of V with itself.) Edges are also called **arcs**.

If E is a set of *ordered* pairs, the graph is called a ***directed graph*** (a ***digraph***, for short) and the edges are called ***directed edges***.
A graph that is not directed is called an ***undirected graph***.

In a digraph, if (v,w) is an edge then w is said to be ***adjacent*** to v, but not vice versa. In an undirected graph, if (v,w) is an edge then v is adjacent to w and w is adjacent to v. The directed edge (v,w) is said to be ***incident*** on the vertex w, and the undirected edge (v,w) is incident on both v and w.

In a ***weighted graph*** (directed or undirected), edges have ***weights***, or ***costs***. A weight is a numeric value assigned to an edge. You can think of a weighted graph as having a weight, or cost, function that assigns an integer to each edge. Weighted graphs are very common because they model many real world relationships. For the remainder of this chapter, I will assume that if a graph is weighted, then there is a function c: E → Z that assigns an integer, positive, negative, or zero, to each edge.

Example 1

Let G be the graph in which V is the set of all American cities with airports, and let E be the set of all edges (v,w) such that there is a scheduled flight from city v to city w. The weight c(v,w) assigned to the edge (v,w) might be (1) the air distance between v and w, (2) the air time between them, or (3) the cost of a minimum cost ticket. Online reservation systems often let the user choose from among many criteria for weighting flights.

Example 2

Let G be a computer network and let V be the set of routers and endpoint computers. E is the set of connections between routers and routers and routers and computers. Weights might be the latency of a transmission across an edge, or the bandwidth of an edge.

A ***path*** in a graph is a sequence of vertices, $v_1, v_2, v_3, \ldots, v_N$ such that for each i, $1 \leq i < N$, $(v_i, v_{i+1}) \in E$. The ***length*** of a path is the number of *edges* (not vertices) in the path, which is N-1 if there are N vertices. A path of length 0 is a path from a vertex to itself with no edges, so it is just a single vertex. In contrast, (v,v) is a path of length 1, which is different from a path of length 0. A path (v,v) is called a ***loop***.

A ***simple path*** is a path such that all vertices are distinct except possible the first and the last.

A ***cycle*** in a directed graph is a path of length at least 1 such that the first and last vertices are the same. A cycle is simple if it is a simple path.

A ***cycle*** in an undirected graph is a path in which the first and last vertices are the same and the edges are distinct (to avoid the problem that a path like {u,v,u} is called a cycle even though the second edge is the same as the first edge.)
A graph is ***acyclic*** if it has no cycles.

A ***directed acyclic graph*** is called a ***DAG*** for short.

An undirected graph is called ***connected*** if there is a path from every vertex to every other vertex.
A directed graph with this property is called ***strongly connected***.

If a directed graph G is not strongly connected but the graph obtained from it by replacing every directed edge (v,w) by an undirected edge (v,w) (which really means two directed edges (v,w) and (w,v)), then the graph is called ***weakly connected***.
A graph is ***complete*** if there is an edge between every pair of distinct vertices.

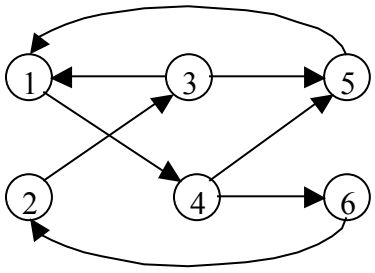
Graph Representations

There are two common representations.

Adjacency Matrix

A boolean or numeric-valued square matrix A with a row for every vertex. A[u][v] is true if there is an edge from u to v. If numeric valued, then A[u][v] would be the weight of the edge (u,v). The following asymmetric boolean matrix represents the directed graph to its right.

	1	**2**	**3**	**4**	**5**	**6**
1	0	0	0	1	0	0
2	0	0	1	0	0	0
3	1	0	0	0	1	0
4	0	0	0	0	1	1
5	1	0	0	0	0	0
6	0	1	0	0	0	0



Advantages

An adjacency matrix has a very simple implementation. That is about its only advantage.

Disadvantages

1. Graphs are usually sparse (there are N^2 entries in a matrix of N vertices, but the number of edges may only be a multiple of N.)
2. Many problems that have efficient solutions cannot be solved efficiently when this representation is used, because they will all tend to be $O(N^2)$.

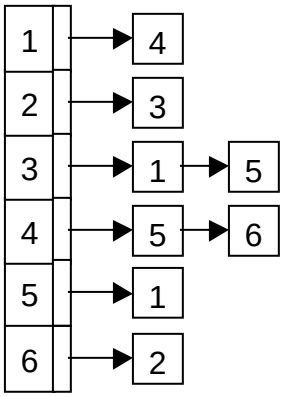
Adjacency Lists

An adjacency list is a linear array with an entry for each vertex, such that each entry is a pointer to a list of vertices adjacent to that vertex. This is the more common representation because it is the most efficient for most purposes. Formally, an adjacency list for a graph G = (V,E) is an array A of size n = |V| such that A[i] is a pointer to a linked list of nodes (v_1, v_2, ..., v_k) such that (A[i],v_j) is an edge in E for each j = 1, 2, ... k.

If the graph is a weighted graph, then the node for each edge in the list would have a member variable that stores the weight.

Vertices often have labels or names such as the name of the cities they represent. In this case the properties are stored in the array entry with the pointer to the list.

The graph above would have the following adjacency list:



Topological Sort

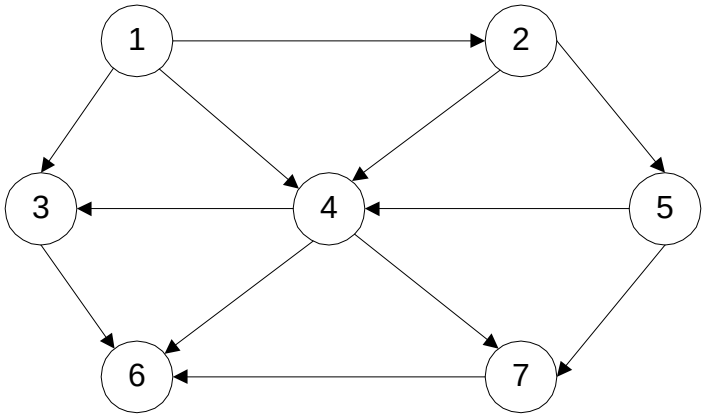
One of the simplest problems requiring solving is to find a topological (partial) ordering of the set of vertices. (A partial ordering of a set is a binary relation that is reflexive, transitive, and asymmetric. It is not necessarily complete.) A topological sort of the vertices { v_1, v_2, v_3, …, v_N } is an ordering such that if there is a path from v_i to v_j in the graph, then v_i must appear before v_j in the ordering. The output of a topological sort of a graph G = (V,E) is an assignment of the numbers 1, 2, .., N to the vertices v_1, v_2, v_3, …, v_N.

The best way to get an intuition for what a topological ordering is, is to assume that a node is a task to be accomplished, and an edge from node v to node w means that v's task must be completed before w's can be started. If there is only one person who can perform all of the tasks in the graph, and that person can only do one task at a time, then a topological ordering is a linear sequence of the nodes that guarantees that the tasks will be done in the right order. In other words, it is an ordering in which no task will be started until all of its prerequisite tasks have been completed.

If a graph has cycles it cannot have a topological ordering, because then for any two vertices v and w in a cycle, v precedes w and w precedes v. There is no way to create a sequence because if v precedes w, then w cannot be before v, and vice versa.

If a graph is acyclic then it may have more than one topological ordering. This will be the case whenever the underlying partial order is not a complete ordering. Any topological ordering will do.

Example



In the above, two topological orderings are 1, 2, 5, 4, 7, 3, 6 and 1, 2, 5, 4, 3, 7, 6

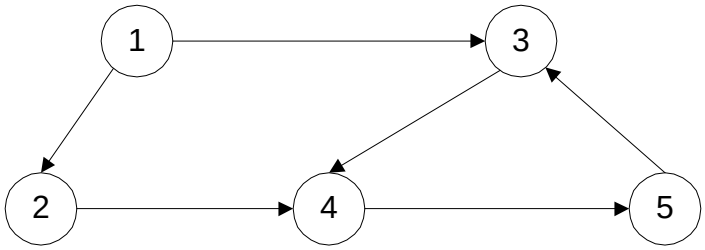
Algorithm

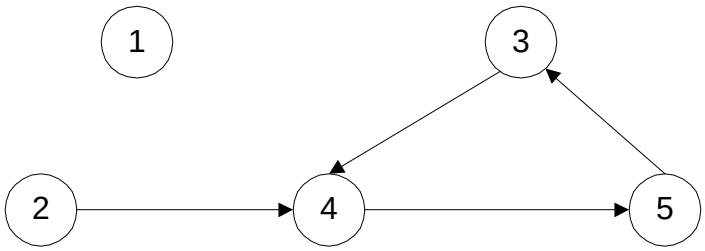
Define the indegree of a node to be the number of edges incident upon the node. In the above graph, node 1 has indegree 0 whereas node 4 has indegree 3.

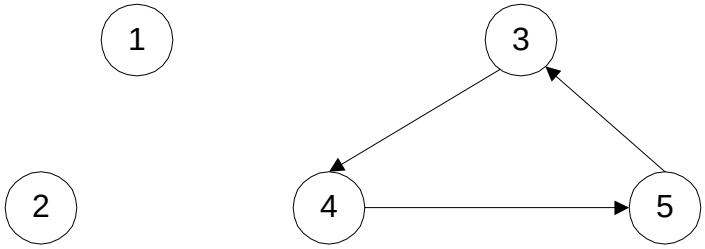
```
counter = 1;
for each vertex in V
{
    let v be any vertex with indegree == 0;
    if no such vertex exists,
        return CycleFound
    else {
        v.topologicalNumber = counter++;
```

```
        for each w that is adjacent to v,
            decrement its indegree by 1
        }
}
```

If the graph has no cycle, then there is at least one node with indegree 0. Taking any one of these nodes, we assign it a 1 and reduce the indegree of all vertices adjacent to it by 1. Reducing the indegree by 1 is like removing the edge from the node to its successors. We try to repeat this until every vertex has been given a number. If we cannot find a vertex with indegree 0 but we have not visited every vertex, then there must be a cycle, because it means that every node in the remaining nonempty graph has at least one incoming edge, which means that the remaining graph has a cycle.







In the above graph, we assign a 1 to node 1, and remove edges (1,2) and (1,3). We then process vertex 2, and remove the edge (2,4). Then there are no vertices with indegree 0, and so this graph cannot be sorted topologically.

The numbering is correct. We can argue by induction on the length of a path from v to w. If there is a path of length 1 from v to w then w is a successor of v and w cannot be visited before v because the edge (v,w) cannot be removed until v is visited, so indegree(w) will not be 0 until after v is visited. So the number assigned to w > number assigned to v.

If the length of the path from v to w > 1, then by assumption there is a vertex u such that {v,…,u} is a path of length n-1 and (u,w) is an edge. By hypothesis, toponum(v) < toponum(u), and by the previous argument, toponum(u) < toponum(w).

The simplest way to implement this algorithm is to use a queue for the vertices. The indegree of each vertex is computed by reading every adjacency list and incrementing the indegree of A[v] every time that v appears in the adjacency list of some other vertex. This is O(E) since it looks at each edge once.

Having computed indegrees, all vertices with indegree 0 are enqueued and a loop is entered. In the loop, the front of the queue is removed, it is given a topological number, its successors' indegrees decremented, and if 0, they are enqueued.

Minimum Spanning Trees

A spanning tree for an undirected graph G = (V,E) is a graph G' = (V,E') such that G' is a tree. In other words, G' has the same set of vertices, but edges have been removed from E so that the resulting graph is a tree. This amounts to saying that G' is acyclic. If G is directed, it means that cycles have been removed and forward edges have been removed.

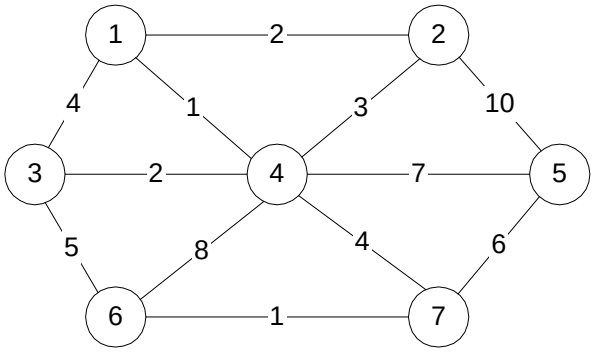
Removal of any single edge from a spanning tree causes the graph to be unconnected.

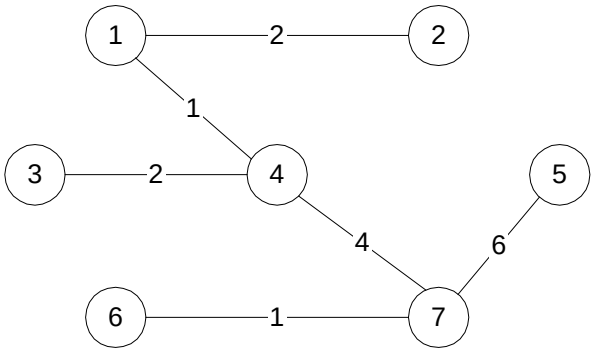
A spanning tree is minimum if there is no other spanning tree with smaller cost. If the graph is unweighted, then the cost is just the number of edges. If it has weighted edges, then the cost is the sum of the edge weights of the edges in the spanning tree.

An example of an application of spanning trees are for finding the least wiring to wire the electrical connections in a building.

Example

A graph and one of its minimum spanning trees.





A relatively simple algorithm for finding a minimum spanning tree is Kruskal's Algorithm. It is a greedy algorithm in that it always tries to find the best solution at each step. Not all greedy approaches work. Here it does. It uses the Union-Find algorithm from Chapter 8.

Kruskal's Algorithm

The idea is to start out with a forest in which each vertex is a tree by itself. Then look for the edge with least weight and connect its two vertices into a tree with two vertices. Find the next smallest weight edge and connect the two vertices together if they are not already in a tree together. If they are in the same tree, then adding this edge would form a cycle, so the edge should not be added. If we continue this process until there is just one tree, then this tree will be minimum cost because the edges were added in order of ascending cost.

The algorithm treats the trees like sets of vertices. To form a new tree the union operation is used. A disjoint set ADT is used to represent the vertices. A heap is used for picking the minimum edge at each step.

```
void kruskal()
{
     int edgesAccepted;
     DisjointSet s(NUN_VERTICES );
     PriorityQueue h( NUM_EDGES );
     Vertex u,v;
     SetType uset, vset;
     Edge e;

     readGraphIntoHeapArray(h);
     h.buildHeap();

     edgesAccepted = 0;
     while ( edgesAccepted < NUM_VERTICES –1 ) {
          h.deleteMin( e );   // assume e = (u,v)
          uset = s.find(u);
```

```
            vset = s.find(v);
            if ( uset != vset ) {
                 // add the edge
                 edgesAccepted++;
                 s.union(uset, vset);
            }
       }
}
```

The algorithm takes O(|E|) steps to build the heap and O(|E|) deleteMins to build the minimum spanning tree. Each deleteMin takes log |E| steps, so this is O(|E| log |E|) steps. Since |E| = $O(|V^2|)$, this is O(|E| log |V|).

Shortest Path Algorithms

We will assume in this section that the graphs in question are directed. There are a few different types of shortest path problems. The simplest one is the single-source shortest path problem. The most general statement of the problem is the weighted path shortest path problem. This is one of the hardest versions of the problem.

Single Source Shortest Path Problem

> *Given a weighted graph G = (V,E), and a distinguished vertex s, find the shortest weighted path between s and every other vertex in the graph.*

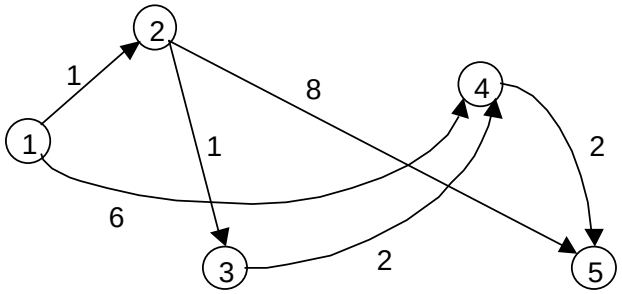
If we let the weight of every edge be 1, then this statement of the problem is reduced to finding the paths whose lengths are least.

Edgar Dijkstra proposed an algorithm to solve the weighted graph version of this problem provided that edges do not have negative edge weights. In his algorithm, it does not matter whether the graph is directed or undirected. The output of the algorithm is a list of the shortest (weighted) distances to each vertex. If the output needs to include the set of paths to each vertex, then the algorithm can be modified to record this information.

The idea of the algorithm is to maintain a temporary set of vertices, T, with the property that the shortest path from s to every vertex in T has been correctly determined, and to enlarge this set iteratively. This set can be thought of as the ***known*** set, because for any vertex in this set, its shortest path from s is known to be correct. Initially, only the source vertex s will be in T, since the distance from s to s is 0. In each iteration, a new vertex is added to T. When the size of the known set is equal to |V|, the algorithm stops.

The algorithm also maintains, for each vertex v not in T, a temporary least distance, d(v), from s. The value d(v) is the weight of the shortest path from s to v that, except for v, passes only through vertices in T. The algorithm may discover as it proceeds that d(v) is too large, and it will reduce it when that happens.

After initializing the set T, and recording the initial distances d(v) to each node, the algorithm enters a loop. In each iteration of the loop, a vertex $v \in V - T$ with minimal d(v) is added to T,



and the distances d(u) to each u $\in$ V –T are updated. Thus, after |V-1| iterations of the loop, all vertices have been added to T and the algorithm terminates.

In the description of the algorithm that follows, the cost function c(v,w), which is defined only on edges (v,w) in the edge set E, is extended to a function c'(v,w) that is defined on all pairs of vertices v,w $\in$V as follows:

$$c'(v,w) = \begin{cases} 0 & if\ v = w \\ c(v,w) & if\ (v,w) \in E \\ \infty & if\ (v,w) \notin E \end{cases}$$

In other words, the cost of an edge (v,w) is infinite if the edge does not exist. In practice, a special value could denote when an edge does not exist. The cost of all other edges is simply the weight of the edge itself.

The Algorithm

```
// Initialize the function d(v):
foreach v ≠ s {
     d(v) = c'(s,v);   // set initial distances
}
d(s) = 0;    // distance from s to s = 0

// Initialize T to contain only s
T = {s};
while ( T ≠ V ) {
    choose a vertex v ∈ V – T with least d(v); // may be more than one
    set T = T  ∪ {v};
    for all vertices u ∈ V – T adjacent to v {
        if ( d(u) > d(v) + c'(v,u) )
             d(u) = d(v) + c'(v,u);

    }
}
```

When we find the vertex v whose d(v) is minimal among all vertices not yet in T, we look at all vertices adjacent to it, and for each one u, if the weight of the path from s to u going through v is

less than it was without going through v, we decrease the weight of its potential shortest path so that it is the weight of the path from s to v to u.

Correctness

The correctness of the algorithm follows from an induction argument on the size of the set T. The inductive hypothesis is that, for every vertex v in the set T, d(v) is the weight of the shortest path from s to v. It is true in the base case obviously since s is the only vertex in the set. Assume that it is true when |T| has size k. Suppose that vertex v is chosen to be added to T by the algorithm when |T| has size k. By the algorithm, d(v) is no larger than any d(w) for all vertices w not in T. Suppose that the weight of the shortest path from s to v is not d(v). Then there is a shorter path p = ($s=v_1, v_2, v_3, \ldots, v_j = v$) from s to v. This path p must contain a vertex that is not in T besides v, otherwise it would just be the path the algorithm chose. In the list of vertices in p = ($s=v_1, v_2, v_3, \ldots, v_j = v$), let $w = v_i$ be the first vertex in p that is not in T. Then, since d(w) must be no smaller than d(v) since otherwise it would be chosen instead of v. Since edge weights cannot be zero, the path through from s through w to v would have to have greater weight than the path to v that does not pass through w, which is a contradiction.

Running Time

The running time of the algorithm greatly depends on how it is implemented. If the vertices are stored in a priority queue based on the values d(v), then choosing the vertex v with least d(v), combined with removing it from V-T is a deleteMin operation requiring O(log|V|) steps. The time to update the values d(u) for each u adjacent to v is also O(log|V|), because the update can be accomplished by doing a *percolateUp(u)* on the vertex in the heap, which is at worst O(log|V|). But there will be O(|E|), since each edge will cause exactly one update in the worst case. Hence the updates cost O (|E| log |V|) but the total of the deleteMins cost O (|V| log |V|), so the running time is O (|E| log |V| + |V| log |V|) .

The problem is that when a vertex is chosen, how can you find the places in the heap where the vertices that are adjacent to it are located. This is not easy, but it is possible. An alternative is , when a vertex u has to be updated, instead of actually finding it in the heap, changing its value, and percolating it up, we just create a new instance of it with smaller key value and insert it into the heap, where it will rise above its current position. Of course now there would be multiple copies of it in the heap, but it doesn't matter if we keep track of whether it has been moved into the set T – we just ignore the ghost copies whose bodies have moved into the set T. The smaller ones will always reach the top before the larger ones. The downside of this strategy is that the heap can get big, O(|E|), to be precise. But for sparse graphs, it is a good solution. In most applications, graphs are sparse.

Chapter 10 The Complexity Classes P and NP

10.1 Introduction

Some problems have a minimum running time that is exponential in the size of their input simply because the size of their output is an exponential function of the size of the input. There is nothing we can do to reduce their time complexity because they must output exponentially many bytes. There are trivial examples of this:

- The problem of, given n, printing all numbers from 1 to 2^n.
- The problem of, given n, printing all $n!$ many permutations of the numbers from 1 to n.

The first is very uninteresting and I am not sure why anyone would want a program to do that. The second is more useful because sometimes we want to generate all such permutations to use as input to another problem, perhaps to test some other program in fact.

The fact that a program has an enormous amount of output does not mean the program itself is enormous itself, as you are well aware, because we can write a program with a tiny loop that produces infinite output. A program that prints all permutations appears in Listing 10.1 below.

Listing 10.1: Program that prints all permutation of 1..N for given N

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>

/* genperms()  prints all permutations of numbers 1 to size */
void genperms( int a[], int size, int k )
{
    int i,j;
    static int depth = -1;

    depth++;
    a[k] = depth;
    if ( depth == size )  {
        for ( j = 1; j <= size; j++ )
            printf("%d ",a[j]);
        printf("\n");
    }
    for ( i = 1; i <= size; i++ )
        if ( a[i] == 0 )
            genperms(a, size, i);
    depth--;
    a[k] = 0;
}

int main ( int argc, char* argv[] )
{
```

```
    int n;  // number of permutations to generate

    /* Check usage  */
    if ( argc < 2 ) {
        printf("usage: %s n, where n is number of permutations\n",
               argv[0]);
        exit(1);
    }

    /* get user input and convert to number */
    errno = 0;
    n = strtol( argv[1], NULL, 10 );
    if ((errno == ERANGE  )  || (errno != 0 && n == 0)) {
        perror("strtol");
        exit(EXIT_FAILURE);
    }

    /* allocate a dynamic array of size n+1 */
    int* p = malloc((n+1)*sizeof(int) );
    if ( NULL == p ) {
        printf("problem allocating storage\n");
        exit(1);
    }
    /* zero-fill the array */
    memset(p,0,n+1);

    /* call recursive function to generate permutations */
    genperms(p,n,0);

    free(p);
    return 0;
}
```

Problems such as the ones stated above, in which the size of the output is an exponential function of the size of the input, assuming we have some reasonable way to define size, are excluded from the remainder of the discussion here; they must take time proportional to the size of the output.

There are many problems that take a long time not because they produce an exponential amount of output, but because they require searching through an enormous number of candidate solutions to find the true solution. For example, imagine that we are given a complete graph with n vertices and we are told that each time we visit a vertex, we pick up a clue, and that if we pick up the clues in the correct order we win a big prize. The graph might be cities to visit, or trees in a forest, or train stations. The obvious solution, perhaps the only solution, is to generate all possible simple paths in the graph and check each one. The above program could be used to do just that, but instead of outputting each path, it would stop when it finds the path that collects the clues in the correct order. In the worst case it requires generating all permutations.

This is a rather simple problem and has a rather simple, though not efficient solution. There are problems for which many known algorithms take an exponential amount of time to solve them, but for which no one has yet to find an efficient algorithm. We are about to reconsider our notion of "efficient" because as you will soon see, we have not really seen what a difficult problem is. So far we have treated an $O(n^3)$ algorithm or an $O(n^2)$ algorithm as an inefficient one if we could find an $O(n^2)$ or $O(n \log n)$ alternative respectively. For the problems we are about to describe, we

would be quite happy with an algorithm that ran in $O(n^{40})$ time, because no one has yet to devise an algorithm that can run in time proportional to n^k for any k! The best that anyone has been able to do has been to find algorithms that solve these problems in exponential time, meaning time proportional to c^n for some real number $c > 1$.

Is this bad? Consider a problem such that the fastest known algorithm that solves it runs in time proportional to 2^n. Suppose that with the fastest computers available today, it takes an hour to solve an instance of the problem of size $n = 100$. To solve an instance of size $n = 101$ would take 2 hours, and an instance of size $n = 116$ would take about 7.5 years. Suppose that in ten years, we could have computers that would run $100,000,000$ times faster, that would only allows us to solve a problem of size $n = 126$ in an hour's time! We would never be able to solve an instance of this problem of size $n = 200$ with this algorithm.

In general, we call problems such as these ***intractable***.

Definition 1. A problem is intractable if an exponential amount of time is needed to discover its solution, and the size of its output is no more than a polynomial function of the size of its input.

What kinds of problems are like this, you might wonder. Some are very easy to state. The most well-known problem of this nature is called the ***traveling salesman problem***, which we can define as follows:

> A traveling salesman wants to visit n distinct cities, starting in her home town and returning to her home town, but visiting every other city exactly once. The problem is to find a shortest route that visits every city exactly once.

It sounds quite simple, and it arises in a variety of contexts, but in fact there are no ways to solve this problem that do not need to find the lengths of a great many paths through the graph, which are usually called tours when discussing this problem.

A similar problem is the ***Hamiltonian circuit*** problem:

> Given an undirected graph, is there any path that connects all of the nodes with a simple cycle?

A simple cycle is a cycle in which each vertex appears exactly once except the first and the last, which are the same vertex. For example, $1, 2, 3, 4, 5, 1$ is a simple cycle. It is not hard to cycle through a graph and visit every node if you do not care about passing through nodes more than once. It is a completely different problem when you do not have this luxury.

The Hamiltonian circuit problem is an example of a ***decision problem***.

Definition 2. A decision problem is a problem that has a yes/no answer.

The traveling salesman problem requires a path as its answer. They are similar, but not exactly the same. The traveling salesman problem as stated above is a minimization problem or optimization problem, but it has a corresponding decision problem:

> Given a finite set of n cities and a bound B, does there exist a tour starting in her home town and returning to her home town, visiting every other city exactly once, of length at most B?

In Chapter 9 we defined spanning trees for graphs. A simple question to ask is whether a graph has a spanning tree in which no vertex has degree larger than some fixed constant k? This sounds like it should not be hard to solve, but it takes an exhaustive search as well. No one has found an efficient algorithm to solve it.

A problem not related to graphs is the ***bin-packing problem***:

> Given a finite set $U = \{u_1, u_2, \ldots, u_n\}$ of n items of various sizes, $s(u_i)$, and a set of L bins into which these items must be placed, what is the least number of bins needed to pack all of the boxes?

A corresponding decision problem for this is

> Given a finite set $U = \{u_1, u_2, \ldots, u_n\}$ of n items of various sizes, $s(u_i)$, a set of L bins into which these items must be placed, and a capacity B, is there a distribution of items into bins such that the sum of the sizes of the items in each bin is at most B?

A second problem not related to graphs is called ***circuit satisfiability***:

> Given a combinational circuit built out of AND, OR, and NOT gates, is there a way to set the circuit inputs so that the output is 1?

Notice that this is also a decision problem and that it is another problem that requires exhaustive searching. In fact this problem is a variant of a very important problem that we will return to shortly.

Sometimes there is a very fine line between an easy problem and a hard problem. A good example is the following pair of problems:

- Given a weighted graph $G = (V, E)$, two vertices v and w, and a weight c, is there a simple path from v to w whose weight *is at most* c.
- Given a weighted graph $G = (V, E)$, and two vertices v and w, and a weight c, is there a simple path from v to w whose weight *is at least* c.

The first problem has an efficient solution; a breadth-first search of the graph will give an answer in linear time. The second has no known solution that runs in less than exponential time. The best that we can do is generate all possible paths and see whether any are sufficiently long.

10.2 Preliminaries

We now make more precise the difference between easy and intractable problems. To begin we have to make the idea of input size more precise.

Definition 3. The ***size of the input*** to an algorithm is the number of bits needed to encode the input, using a *reasonable* encoding method.

Reasonable generally means using the conventional methods, such as a binary encoding, hexadecimal or base ten or some other constant radix. An unreasonable would be a unary encoding, requiring m bits to encode the number m.

Definition 4. An algorithm A ***solves a problem*** p if, for every input to the algorithm, it terminates and outputs a solution to p for that input.

In this case we will simple say that the algorithm *solves* p, or that p *is solved by* the algorithm.

Definition 5. An algorithm A solves a problem p in ***polynomial time*** if there exists a number k such that, for every input of size n given to the algorithm, it solves the problem for that input in time at most n^k.

Notice that this implies a ***worst case running time***, because it says that for all inputs of size n, the upper bound on the running time is n^k. In particular, whichever one takes the longest is included.

Next we want to define what a deterministic algorithm is. We assume that all instructions have unique addresses, or numbers, and that all variables can be uniquely identified by their addresses in memory.

Definition 6. The ***state*** of an algorithm is a complete description of the values of all variables and memory locations used by the algorithm, together with the address of the instruction that it is about to execute. The ***initial state*** of the algorithm is the state in which the first instruction is about to be executed for the first time[1] and all of its memory has been created.

Even things like open files are part of the state of an algorithm. If it has a file open, then the contents of the file are part of its state, as well as the next byte in the file to be read or written. The state leaves nothing out.

Definition 7. An algorithm is ***deterministic*** if, given its current state, executing the current instruction in that state uniquely determines the algorithm's next state.

In other words, a deterministic algorithm leaves nothing to chance; given a particular input, it always produces the same output. In essence the collection of input/output pairs of the algorithm is a function. The algorithms we have learned about in this course, and probably most of the algorithms you have learned about in your other undergraduate courses, are all deterministic algorithms. Soon we will see what a nondeterministic algorithm looks like.

Lastly, for the remainder of this chapter, ***we will restrict all problems under consideration to be decision problems***.

10.3 The Complexity Class P

We define a very important collection of decision problems that have a common property.

Definition 8. The set of all decision problems that can be solved by a deterministic algorithm in polynomial time is the complexity class **P**.

Notice that **P** is a set of problems, not algorithms. **P** is called a ***complexity class***, because it is a class of problems that have the same computational complexity. The computational complexity of a problem is the asymptotic worst case running time of the best algorithm known to solve

[1] It might be re-executed at a later time.

that problem. The definition of **P** states that a problem belongs to this class **P** if there is some deterministic algorithm that solves it in polynomial time.

Almost all of the problems that you typically study in an undergraduate program in computer science run in polynomial time. The ones that are decision problems belong to the class **P**. Many of the problems have decision problems that correspond to them. These includes problems such as sorting, all of the various shortest-path problems in graphs, algorithms to determine whether graphs are acyclic, strongly-connected, and so on, and searching for keys in trees and tables.

10.4 The Complexity Class NP

You now have to let your imagination loose a bit, to understand the idea of nondeterminism, because we will use it to describe an imaginary computer or algorithm, one that cannot exist as a physical reality. But it plays a very important role in theoretical computer science and dates back many decades. In a nondeterministic algorithm the next state that it enters is not determined by the current state; it is endowed with the ability to enter any one of a set of multiple next states, and this choice occurs, in a sense, simultaneously.

Imagine that a person is walking in a forest and the path branches. The person is forced to choose a path and follow it. Suppose that somehow the person could walk both paths simultaneously, not by making a copy of him or herself, but by cloning time. Then the person would be walking nondeterministically. It is not the same thing as parallel execution, which would be like copying the person. It is not the same thing as a probabilistic algorithm, which would make a random guess and follow it. It is an abstraction that we can only imagine.

We apply this idea to algorithms. Imagine an algorithm that can guess a candidate solution to a problem in one stage and then check if the guess is correct in a second stage. We can call the first stage the ***guessing stage***, and the second, the ***checking stage***. The idea is that the algorithm can make all guesses simultaneously and then check them all simultaneously. This means, in effect, that if there is a solution to the problem, it will be one of its guesses, and that it will discover this in its checking stage, and that the total time it takes to do this is the same as if it made a single guess and checked it.

All of this can be formalized very rigorously, but to do so would require a fair amount of background theory. The original formulation was based upon Turing Machines. We do not need to use Turing Machines to explain the ideas we present here, but we make lack rigor in doing so. *Although it is not necessary to do so, for the remainder of this section we will restrict the problems under consideration to decision problems.*

As an example, a nondeterministic algorithm could solve the traveling salesman decision problem by first guessing an arbitrary sequence of cities that start in a given city, end in the same city and visit each city exactly once, and then in the checking stage, it would check whether the guess had length at most B. If there is a tour of length at most B, then one of the guesses will be correct and will be verified by the checking stage and the output will be *yes*. If there is no tour of that length, no guess will be found that is of the correct length, and the output will be *no*. We can make this more formal with the following definition.

Definition 9. A nondeterministic algorithm ***solves a decision problem*** d if,

- For each input to d for which the answer should be yes, there is a guess that it can make in the guessing stage that when it is checked in the checking stage, will result in the answer "yes", and
- For each input to d for which the answer should be no, there is no guess that it can make in the guessing stage that when it is checked in the checking stage, will result in the answer "yes".

This is a reasonable definition. It basically says that the algorithm always finds a correct guess if it exists and cannot find one if it does not exist. We can define a polynomial-time nondeterministic algorithm as follows.

Definition 10. A nondeterministic algorithm solves a decision problem d in polynomial time if there exists a polynomial p such that, for every input to d for which the answer should be yes, there is a guess that leads in the checking stage to a yes answer within $p(n)$ time.

This implies that the size of the guess is also bounded by the polynomial, because it would be impossible to spend a polynomially-bounded amount of time checking a structure whose size was more than polynomial. We can now define the class **NP**:

Definition 11. The set of all problems that can be solved by a nondeterministic algorithm in polynomial time is the complexity class **NP**.

One immediate consequence of this definition is that every problem in **P** is also in **NP**: if a problem can be solved with a deterministic algorithm in polynomial time, it can be solved with a nondeterministic algorithm in polynomial time also. Therefore $\mathbf{P} \subseteq \mathbf{NP}$. The most important open question in computer science is whether $\mathbf{P} = \mathbf{NP}$, i.e., whether there are problems in **NP** that are not in **P**. No one has proved or disproved this statement, though many have tried. This is a big problem, because there are many important, practical problems known to be in **NP**, not yet known to be in **P**, and if someone were to prove that $\mathbf{P} = \mathbf{NP}$, this would imply that these problems did indeed have efficient solutions. If someone proved that $\mathbf{P} \neq \mathbf{NP}$, this would not necessarily mean that any specific problem did not have an efficient solution, but that certain ones did not, which leads to the concept of ***NP-completeness***.

10.4.1 Exponential Time

If a problem is in **NP**, then it can always be solved by an algorithm that runs in exponential time deterministically. To see this informally, suppose that D is a problem in **NP**. Then there is some polynomial $p(n)$ such that instances of size n can be decided by nondeterministic algorithm that runs in time $p(n)$. This means that the size of the guess that is handed to the checking stage is no more than $p(n)$ bits in size. A deterministic algorithm can enumerate every possible guess of at most that size and try each one in polynomial time. There are at most $O(2^{p(n)})$ such guesses, so such an algorithm can generate the guesses and check them in exponential time. This shows that every problem in **NP** can be solved by an exponential time algorithm in **P**.

10.5 NP-Completeness and NP-Hardness

There are certain decision problems that are known to belong to **NP**, but which may or may not belong to **P**. No one has found efficient algorithms for any of them, nor has anyone proved that

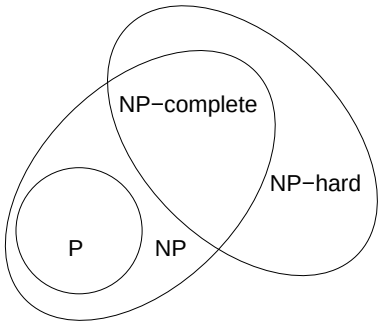


Figure 10.1: An Euler diagram showing the world if $P \neq NP$

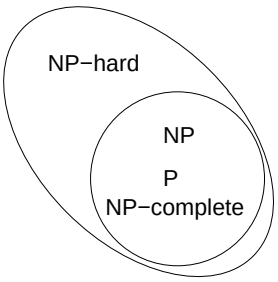


Figure 10.2: An Euler diagram showing the world if $P = NP$.

these algorithms cannot exist, and many people have tried to do one or the other. So there is no proof that they are in **P** and there is no proof that they are not. The list of such problems is quite large, but among them are:

1. circuit satisfiability, defined above
2. bin-packing, defined above
3. Hamiltonian circuit, defined above
4. subset sum: Given a set X of integers and an integer t, does X have a subset whose elements sum to t?
5. longest path: Given a non-negatively weighted graph G and two vertices u and v, is there a simple path from u to v in the graph of length greater than B?
6. graph isomorphism: Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, is there a one-to-one, onto function $f : V_1 \rightarrow V_2$ such that $(u, v) \in E_1$ if and only if $(f(u), f(v)) \in E_2$?
7. integer factorization: Given an integer n and an integer m with $1 \leq m \leq n$, does n have a factor q with $1 < q < m$?

There are certain problems for which it has been proved that if any of them can be solved in polynomial time with a deterministic algorithm, then so can all problems known to be in **NP**, meaning that **P** = **NP**. Such problems are called ***NP-hard*** problems. Informally, a problem is NP-hard if it is at least as hard as any problem in **NP**. A problem is ***NP-complete*** if it is NP-hard and also in **NP**.

If someone was able to prove that any single problem known to be in **NP** could not be solved with a polynomial-time algorithm, i.e., that such a problem was in **NP** but not in **P** (**NP** − **P**), this would also imply that every NP-complete problem was in **NP** − **P**. These NP-complete problems are an important tool in understanding the relationship between these complexity classes.

Problems 1 through 5 above have been proven to be NP-complete. Problems 6 and 7 are known to be in **NP** but no one has proved they are NP-complete and no one has proved they are in **P**.

Are there decision problems that are NP-hard but not NP-complete? All undecidable problems fall into this category, such as the ***Halting Problem***[2], but there are also decidable problems that are NP-hard but not NP-complete. One example is called the ***true quantified Boolean formula problem***, which asks whether a boolean formula containing existential and universal quantifiers has an assignment that makes it true. See Figure 10.1 for a picture of how the sets are related if **P** ≠ **NP**.

10.5.1 Proving NP-Completeness

Suppose that a decision problem D is known to be NP-complete and we are given a problem D' in **NP**. Suppose that we can find a transformation T that can convert any instance d of the old problem D into an instance d' of the new problem D' in such a way that $d' = T(d)$ has a yes answer as an instance of D' if and only if d has a yes answer as an instance of D. Suppose also that the transformation can be done in polynomial time. Then we have a procedure for solving all instances of the original problem D as follows:

1. Transform the instance d into a problem $T(d)$ of D' using polynomial time.
2. Use the algorithm for D' to solve the problem $T(d)$.
3. Report the answer is yes for d if and only if the answer is yes for $T(d)$.

If D' has a polynomial time deterministic algorithm (meaning it is in P), this would imply that the old problem is also in P and that P=NP. In other words, D' is NP-complete.

Example 12. Suppose that we know that the Hamiltonian circuit problem is NP-complete but we do not know whether the traveling salesman problem is. The transformation we use is the following:

Given a graph $G = (V, E)$ with $|V| = n$ that is an instance of the Hamiltonian circuit problem, create a traveling salesman problem instance by creating a city for each vertex in V, and for the distances between the cities, define the distance between city v and city w to be 1 if there is an edge $(v, w) \in E$ and 2 if there is no edge (v, w). Clearly this is a polynomial-time transformation.

Run the algorithm for the traveling salesman problem to decide if there is a tour of weight at most n on this constructed instance. If it finds a tour, this implies that there was a cycle in the original graph that connected all of the vertices in the graph (because the traveling salesman visits every city and so traversed all vertices and used edges of weight 1 each.) If it finds no tour of weight at most n, then there is no path in the original graph that is a cycle that visits every vertex.

Therefore, if the traveling salesman algorithm has a deterministic polynomial time algorithm, since we can use it to solve the Hamiltonian circuit problem with this polynomial-time transformation, it implies that the Hamiltonian circuit problem also has a polynomial-time deterministic algorithm.

[2]The Halting Problem, stated in informal terms, is the problem in which we are given a program and an input to that program and asked whether the program will eventually halt when run on that input. The original version was stated in terms of Turing Machines.

The procedure described informally above is called a ***polynomial-time reduction*** of D to D'. If we can reduce a known NP-complete problem to another problem in NP, then the second problem must be NP-complete as well. This shows that we can extend the set of known NP-complete problems by finding suitable polynomial time reductions of known NP-complete problems. But this is only useful if there is a starting problem known to be NP-complete, one that was not proved to be NP-complete by reduction from some other NP-complete problem! How did it get there?

10.5.2 Cook's Theorem

In 1971, Stephen Cook proved the existence of the first NP-complete problem. He showed that the satisfiability problem is NP-complete, not by a reduction but by a direct proof. In short, he proved that if satisfiability has a polynomial-time deterministic algorithm, then so does every other problem in **NP**.

We cannot understand his proof unless we know what a Turing Machine is, but we can describe the basic idea. He showed how the computation performed by a nondeterministic Turing Machine (*NDTM*) could be expressed as a boolean formula in such a way that, the formula has an assignment that makes it true if and only if the Turing Machine outputs a solution to the given problem. Because a Turing Machine can be described in precise mathematical terms, it was possible to create a single boolean expression whose length was a polynomial in the size of the input to the machine such that the expression represented the running of the machine on that input in polynomial time. Because a *NDTM* can be used to solve any problem in **NP** in polynomial time, his construction showed that for every problem in **NP**, there was an instance of the satisfiability problem such that a yes answer to the satisfiability problem on that instance corresponded to a yes answer to the original problem. This established the first NP-complete problem, which is known as *SAT* for short.

Assignment 1: Polynomial Multiplication

Overview

The purpose of this project is to give you practice in designing and implementing classes, and using existing templates classes. You will design and implement a class to represent polynomials and perform polynomial multiplication. You will also write a main program that is a client of the class. The main program will read a sequence of polynomial definitions and operations from a file, create polynomials, and perform the specified operations. If you need to brush up on polynomial arithmetic, refer to any elementary algebra textbook.

Polynomial Arithmetic

A ***polynomial*** (in one variable) is a function of the form

$$p(x) = c_n x^n + c_{n-1}x^{n-1} + \cdots + c_1 x + c_0$$

where n is a non-negative integer, $c_n \neq 0$, and the other coefficients c_k may or may not be zero. The degree of $p(x)$ is n. No exponent in a polynomial is allowed to be negative. In this assignment, all coefficients are assumed to be integers. (In this case it is called a polynomial in an integer domain.) Note that if the degree of the polynomial is 0, then the polynomial is just a constant. For the remainder of this specification, I write p as a shorthand for $p(x)$.

Given two polynomials p and q, the following binary operation is defined:

```
p * q   is defined to be the product of polynomials p and q
```

Example

If $p(x) = 2x + 3$ and $q(x) = -x + 1$ then

$$p * q = (2x + 3) * (-x + 1) = -2x^2 – x + 3$$

Project Details

The Class Interface

You are to design the interface to a `Polynomial` class. The class must contain the following member functions.

Name	Description
`eval(double x)`	Evaluate the polynomial using argument `x` and return the value. This should be implemented as an overloaded function call `operator().`
`operator*`	Given polynomials `p` and `q`, return `p*q.`
`operator<<`	Given an output stream `out` and polynomial `p`, display `p` in symbolic form on `out.`

In addition, the class needs

- a default constructor that creates a zero polynomial,
- a constructor that takes a coefficient `c` and an exponent `e` and constructs a polynomial with the single term cx^e, e.g., `Polynomial p(c,e);`
- a copy constructor that takes a polynomial `q` and makes a new polynomial that is a copy of `q`, e.g. `Polynomial p(q);`
- an `operator=` for the class that will (copy) assign a polynomial to an existing polynomial;
- a destructor, which deletes the polynomial.

Note that this description makes no mention of the private part of the class. That is up to you. You will probably find it necessary to implement other private members of the `Polynomial` class.

Design and Implementation Requirements

A polynomial must be implemented using a list. The *C++* language includes a list template class, and you *must* use this template class instead of writing your own list class. That is my intention in giving you this assignment.

The list provides all of the functions that you will need. I suggest that your list nodes be ***terms***. Each term is completely defined by its coefficient and exponent. No two terms can have the same exponent; i.e., every node must have a unique exponent. It is a good idea to keep the nodes sorted by exponent value. When two polynomials are added, some terms may cancel. For instance, if $p = 2x + 1$ and $q = -2x + 2$, and we let$r = p + q$, then $r = 3$ because the terms $2x$ and $-2x$ canceled. We can end up with a result with fewer (and possibly no) terms. This implies that you need to check when a coefficient of the result is zero, and delete the node. There are other issues that need to be resolved regarding how to perform the arithmetic.

The main program, class implementation, and interface must each be in its own file. For this project there is no need to have more than three source code files.

Input and Output

The main program will create a vector or array of at most 100 polynomials. I will name it `Poly` here. You can name it whatever you like. The main program will then read input from a text file whose name is specified on the command line. If no file is specified or if the file that is specified does not exist or cannot be opened for one reason or another, the main program must display an error message and then exit. Each line of the file will be in one of the following four formats:

1. k : c_1 e_1 c_2 e_2 ... c_n e_n
2. `n = m * k`
3. `eval n(6)`
4. `show n`

The first format is the definition of a new polynomial. The number before the colon, `k`, is the place in the vector (array) where the polynomial should be stored. There will be whitespace separating all tokens on the line, including the colon and the coefficients and exponents. The line will be well-formed, so you do not have to do input validation in this assignment. I.e., there will be a set of pairs of numbers after the colon, each representing a term of the form $c_k x^k$. The terms are not sorted by increasing or decreasing exponents. The array index will always be a number in the range $[0, 99]$. These are both legal definitions:

```
3 : 6  5  -4  3  -2  1  2  0
4 : 1  2   3  4   -9  1
```

which define `Poly[3]` as $6x^5 - 4x^3 - 2x + 2$ and `Poly[4]` as $3x^4 + x^2 - 9x$.

The second format is an instruction to compute a product. It states that `Poly[n]` should contain the product of `Poly[m]` and `Poly[k]`, deleting anything that might have been in Poly[n] previously. You can assume all indices will be valid at the time the line is reached (both operand polynomials exist.)

The `eval` instruction specifies that the polynomial `Poly[n]` is to be evaluated with argument 6 and its value displayed in the standard output stream. The argument can be any floating-point number, such as `3.2`. The output should be something like

```
Poly[n](6) = whatever the value is
```

The last format is an output instruction. The specified polynomial, `Poly[n]`, must be displayed in the output stream in the format below. It does not need to be wrapped onto a new line if it seems long. The stream will do whatever wrapping needs to be done.

```
Poly[n] = a1 x^e1  +  a2 x^e2  +  ... +   am x^em
```

where the terms should be displayed in decreasing order of exponent.

For example, the input could look like

```
0 : 1 2 2 1 1 0
1 : 1 1 1 0
2 = 0 * 1
show 2
0 = 1 * 2
1 = 0 * 2
show 1
```

Testing Your Program

You should design your own input files and test your program using your own input. You should carefully check that the output of your program is correct for the inputs you gave to it. Try it on the simplest of polynomials to be sure. Try it on constants, on polynomials that cancel terms when multiplied, and so on.

Programming Rules

Your program must conform to the programming rules described in the Programming Rules document on the course website. It is to be your own work alone.

Grading

The program will be graded based on the following rubric.

- If the program does not compile on a cslab machine, it receives only 25%.
- For programs that compile:
  - Correctness and implementation 60%
  - Design (modularity and organization) 20%
  - Documentation: 10%
  - Style and proper naming: 10%

Submitting the Assignment

This assignment is due by the end of the day (i.e. 11:59PM, EST) on February 24, 2014. Create a directory named named *username*`_hwk1`. Put all project-related source-code files into that directory. ***Do not place any executable files or object files into this directory.*** You will lose 1% for each file that does not belong there, and you will lose 2% if you do not name the directory correctly. With all files in your directory, run the command

```
zip -r username_hwk1.zip ./username_hwk1
```

This will compress all of your files into the file named `username_hwk1.zip`.

Before you submit the assignment, make sure that it compiles and runs correctly on one of the `cslab` machines. Do not enhance your program beyond this specification. Do not make it do anything except what is written above.

You are to put your zip file into the directory

```
/data/biocs/b/student.accounts/cs335_sw/projects/project1
```

Give it permission 600 so that only you have access to it. To do this, `cd` to the above directory and run the command

```
chmod 600 username_hwk1.zip
```

where *username_hwk1.zip* is the name of your zip file.

If you put a file there and then decide to change it before the deadline, just replace it by the new version. Once the deadline has passed, you cannot do this. I will grade whatever version is there at the end of the day on the due date. You cannot resubmit the program after the due date.

Assignment 2: Extended AVL Trees

Overview

In this assignment, you will implement an ***enhanced AVL*** (***EAVL***) tree. The EAVL tree differs from an AVL tree in that it has member variables that store

- its current height,
- the number of nodes in the tree (called its size), and
- the internal path length of the tree.

It also provides a method for reporting the values of each of these metrics, as well as the number of nodes visited by find operations. Lastly, it has a method to provide the average number of nodes visited in all find operations so far.

Your main program will read a text file that contains commands, one per line, and will process those commands one after the other. The file includes commands to insert, remove, and find a specific word, to print the tree's contents in sorted order, and to report on the tree's statistical properties. The syntax and semantics of the commands are described in the detailed requirements below. Only the main program is permitted to read from an input file or write to an output file. The EAVL tree is permitted to write to an output stream that is passed to it, but not to any files.

Detailed Requirements

Input and Output

The program must get the name of the input file from its only command line argument. Specifically, the program must parse the command line and extract the input file name from the first command line argument. If there is no command line argument, it must report this as an error and exit. If the file name is supplied but cannot be opened for any reason, it must report this error and exit. The program is to put all output on the standard output stream, not in any file; to repeat this, it is not to place its output into a file.

Input File Syntax, Semantics, and Error Handling

The input file will consist of an unlimited sequence of lines, each of which starts with a command. The allowable commands are listed in the table below. The lines are free form, which means that any number of white space characters may precede or separate the tokens in the line. The lines are case-sensitive, i.e., "insert" and "Insert" are considered to be two different words. More importantly, the data is case-sensitive; the words "apple" and "Apple" are two different words and would have to be stored separately if they were each inserted.

The table below defines the semantics of each command. For each command, your program must take the action indicated in the right hand column. In this assignment, a *word* is any sequence of one or more non-blank, non-control characters, including letters, digits, and punctuation marks. Words may be up to 32 characters long. In the table, **boldface** indicates command keywords and *italics* represent placeholders for data.

Command	**Description**
`insert` *word*	If *word* is not already in the tree, create a new entry for it with frequency 1; otherwise increment its frequency in the tree and output the new frequency in the form: *word* <tab> *frequency*

Command	Description
`remove` *word*	If *word* is in the tree, decrement the frequency of *word*, and delete it if the frequency is zero. Then display a line of output in the form: *word* <tab> *frequency* If *word* is not in the tree, display *word* `<tab> not found`
`find` *word*	Search the tree for *word*. If it is found, display *word* and its frequency on a line of output. If it is not found, display *word* with a zero frequency. In either case, output the number of nodes visited in the search.
`display`	Display the contents of the tree in sorted order, including the frequencies of the items in the tree. Use the default collating sequence. This means that uppercase will precede lowercase. The libraries use this ordering by default in C and C++.
`report`	Produce a report for the tree consisting of: 1. the size of the tree 2. the height of the tree 3. the internal path length of the tree, and 4. the average number of nodes visited by the find command so far. The report should list each of these metrics, on a single line, in the above order, with a label that indicates what it is, such as `size = 120`, in the output stream.
`quit`	Clean up resources and terminate the program.

The main program should catch any line that is not one of the forms listed in the table above and display an error message on the ***standard error stream*** for each such line that it finds. It should continue to the next line of input after flagging the bad line.

Main Program and Project Structure

The main program should do all I/O and act like a client to the EAVL class. Specifically, it needs to repeatedly read the next command from input file, process the command, and write the appropriate output. It must also clean up all memory used before exiting.

The EAVL Tree Class Interface

Your EAVL Tree class must contain the following public methods. You may add others if you choose. You must implement the deletion algorithm by using replacement by the in-order successor, not the in-order predecessor. The methods are described by how they would be called by a client in the left column.

Public Method	Description
`int n = insert(`*item*`)`	If *item* is not in the tree, insert *item* into the tree, otherwise increment *item*'s frequency. In either case, return the frequency of *item* after the insertion has been performed.

Public Method	Description
`int n = remove(`*item*`)`	If *item* is not in the tree, return -1. If *item* is in the tree, decrement its frequency, and if the new frequency is zero, remove *item* from the tree. In either case return the frequency of *item* after the removal has been performed (so that a zero indicates the item is removed completely).
`int n = find(&`*item*`, &`*freq*`)`	Find the node containing *item* and, if it is found, update the frequency of *item* in the tree with the new frequency, *freq*. If *item* is not in the tree, set *freq* to 0. In either case, return the number of nodes visited.
`int n = height()`	Return the current height of tree.
`int n = int_pathlength()`	Return the current internal path length of the tree.
`int n = size()`	Return the total number of nodes in tree.
`float x = avge_nodevisits()`	Return the average number of nodes visited by all completed `find` operations on the tree so far. The average is, by definition, the total number of nodes visited divided by the total number of completed `find` operations. A `find` operation is complete if it returns a value of any kind.
`display(`*ostream*`)`	Write the items in the tree in sorted order, one per line, onto the given output stream.

Testing Your Program

You should design your own input files and test your program using your own input. You should carefully check that the output of your program is correct for the inputs you gave to it. Include files with bad lines, files with no lines, files that cannot be opened, files with all kinds of spacing, and so on.

Programming Constraints

- You are free to use either the code from my notes or the book, but you must cite this in the preamble if you do.
- You are not permitted to use any features of C++-11; the program must compile with the GNU C++ compiler in the lab, version 4.7.2 without the need to specify C++-11.
- For full credit your solution must maintain the size, height, and internal pathlength information as efficiently as possible.
- Your program must conform to the programming rules described in the Programming Rules document on the course website. It is to be your own work alone.

Grading

The program will be graded based on the following rubric.

- If the program does not compile on a cslab machine, it receives only 25%.
- For programs that compile:
  - Correctness and implementation 50%
  - Performance 10%
  - Design (modularity and organization) 20%
  - Documentation: 10%
  - Style and proper naming: 10%

Submitting the Assignment

This assignment is due by the end of the day (i.e. 11:59PM, EST) on April 7, 2014. Create a directory named named *username*_hwk2. Put all project-related source-code files into that directory. ***Do not place any executable files or object files into this directory.*** You will lose 1% for each file that does not belong there, and you will lose 2% if you do not name the directory correctly. With all files in your directory, run the command

```
zip -r username_hwk2.zip ./username_hwk2
```

This will compress all of your files into the file named `username_hwk2.zip`.

Before you submit the assignment, make sure that it compiles and runs correctly on one of the `cslab` machines. Do not enhance your program beyond this specification. Do not make it do anything except what is written above.

You are to put your zip file into the directory

```
/data/biocs/b/student.accounts/cs335_sw/projects/project2
```

Give it permission 600 so that only you have access to it. To do this, `cd` to the above directory and run the command

`chmod 600` ***username_hwk2.zip***

where ***username_hwk2.zip*** is the name of your zip file.

If you put a file there and then decide to change it before the deadline, just replace it by the new version. Once the deadline has passed, you cannot do this. I will grade whatever version is there at the end of the day on the due date. You cannot resubmit the program after the due date.

Assignment 3

Overview

This assignment combines several different data abstractions and algorithms that we have covered in class, including priority queues, online disjoint set operations, hashing, and sorting. The problem is a simplification of the problem of inferring relationships from social networking data.

In this assignment, you are to write a program that processes communication data. The input is obtained from a file named on the command line. The file consists of lines of text of various types. A ***data line*** consists of the word `data` followed by two *distinct* telephone numbers followed by a positive integer. Each telephone number is in the form *xxx-xxx-xxxx*, where each '*x*' is a decimal digit. The positive integer represents an amount of money transferred from the first number to the second number, in whole dollar amounts. For example, the line

```
data    807-444-2100    201-222-1200    15400
```

represents the fact that $15,400 was transferred from the owner of 807-444-2100 to the owner of 201-222-1200. The number more generally represents the strength of the relationship between the two numbers. A telephone number x is ***linked*** to a telephone number y if either x and y are the same number or a transfer of money was made between them. Two numbers x and y are ***connected*** if x and y are ***linked*** or if there is a number z such that x and z are linked and z is connected to y. A ***cohort*** is the largest set of telephone numbers such that every number in the set is connected to every other number in the set. This implies that if a telephone number is not in the cohort, then no transfer was ever made between that number and any number in the cohort. It also implies that every pair of cohorts is disjoint, and that no cohort is empty.

From these definitions, we can conclude that the *connected* relation is symmetric, reflexive, and transitive, and hence, an equivalence relation, and that the set of all cohorts forms a partition of the set of all telephone numbers.

The ***size*** of a cohort is the cardinality of the set (the number of telephone numbers in it.) The ***volume*** of a cohort is the total amount of money transferred between members of the cohort. From the definition of connectivity, cohorts of size 1 have volume 0. The ***activity*** of a cohort is the volume divided by the total number of unordered pairs of distinct numbers in the cohort, which is $N(N-1)/2$ where N is the size of the cohort; it measures the average amount of money that has been transferred between any pair of members of the cohort.

When a data line is read, the program must update all cohorts to reflect possible new relations. The update must include any changes in the size, volume, and activity of all cohorts. Every cohort must be identified by a ***positive integer ID***, called its ***cohort-id***.

The other types of lines that may appear in the input file are as follows. For each command, your program must take the action indicated in the right hand column. In the table, **boldface** indicates command keywords and *italics* represent placeholders for data. `Fixed font` indicates actual data.

Command	Argument Type	Description
`find`	*phone-number*	The program should display the ID of the cohort to which the phone-number belongs, in the form *phone-number*`:` *ID* If there is no such number, it should display *phone-number*`: no cohort`

Command	Argument Type	Description
`members`	*cohort-id*	The program should display a list of all telephone numbers in the given cohort, one per line. If there is no such cohort, it should display a line indicating there is no such cohort.

`max`	one of the following strings:
	`activity`
	`size`
		The program should display the cohort-id, activity, size, and volume of the cohort having the maximum value of the given property. If two or more cohorts have the same maximum value, then all that are maximal should be displayed. If there are no cohorts, it should display a line indicating there are no cohorts.

`cohort-ids`		The program should display a list of all cohort ids in order of increasing id.
`info`	*cohort-id* / `0`	The program should display 1. the cohort-id, 2. the activity, 3. the size, and 4. the volume of the cohort whose cohort-id is given. If no such id exists, it should display an error message. If the argument is 0, then the program should display this information for all of the cohorts, sorted by increasing cohort-id. These metrics should be listed on a single line, in the above order, separated by tab characters.

Further Details About Input And Output

The program must get the name of the input file from its only command line argument. Specifically, the program must parse the command line and extract the input file name from the first command line argument. If there is no command line argument, it must report this as an error and exit. If the file name is supplied but cannot be opened for any reason, it must report this error and exit. The program is to put all output on the standard output stream, not in any file; to repeat this, it is not to place its output into a file.

Telephone numbers will have no space in them, and will have hyphens as indicated above. Tokens in the input line may be preceded or followed by any number of white space characters. The number of distinct telephone numbers will not exceed 5000. From the preceding information, you can see that there are a total of six different words that can start a line in the file. The program should catch any line that does not start with one of these words and display an error message on the ***standard error stream*** for each such line that it finds. It can assume that the remainders of all lines are in the proper form. It should continue to the next line of the file after flagging the bad commands.

Implementation Requirements

1. Telephone numbers are strings, not integers. The union-find algorithm description is based on sets whose names are array index values. How are strings associated with these index values? When a telephone number is given to the program, the program needs to store it and access it easily. Although you could use a vector and some type of $O(\log n)$ look-up structure for it, the proper solution is to devise a hash table and hashing scheme that will allow nearly $O(1)$ look-ups. For the union-find algorithm, each new set should occupy the next free position in an array, which suggests that some

type of object must keep track of that position and associate it with each new telephone number, which is a set of size 1. The idea is therefore that, given a telephone number, you associate a new set id with it and store it in such a way that you can retrieve the set id from the telephone number in $O(1)$ time. But you also need to find all of the telephone numbers in a given set easily, which suggests that the array used for the union-find algorithm does not store just simple integer values.

2. You must use the union-find algorithm, with path compression and union-by-size, to solve this problem. All set manipulation must be carried out in a suitably defined class.

3. You must use a priority queue for finding the maximum cohort when the `max` command is issued. Since the maximum may be with respect to one of two different properties, this suggests using some type of indirection in the queue nodes. When sets are updated as a result of reading new data, the priority queue may become invalid because one set may become larger or smaller with respect to one or more properties than before, and because a new set may be created, or an existing one removed. Although you could update the priority queues after each data operation, you could use the on-demand approach of leaving them in their old states, and only when the `max` command is issued, heapifying and find the maximum element. For the sake of efficiency, you might consider keeping a 'dirty' flag to test if it is necessary to re-heapify. This is your choice; you can either update your queue after every operation that changes the properties of the sets, or only "on-demand."

4. You must use heapsort to sort cohorts for display.

5. All interaction with the file system and the terminal must be performed by the main program. Classes are free to create strings to pass to the main program, or to write onto ostream objects that are passed as parameters.

6. Activity must be computed with floating point division and reported to two decimal places of precision. Size and volume are whole numbers.

Programming Constraints

- You are not permitted to use any features of C++-11; the program must compile with the GNU C++ compiler in the lab, version 4.7.2 without the need to specify C++-11.

- Your program must conform to the programming rules described in the Programming Rules document on the course website. It is to be your own work alone.

Testing Your Program

You should design your own input files and test your program using your own input. You should carefully check that the output of your program is correct for the inputs you gave to it. Include files with bad lines, files with no lines, files that cannot be opened, files with all kinds of spacing, and so on. Include data that creates several cohorts of equal size, volume, and activity.

Grading

The program will be graded based on the following rubric.

- If the program does not compile on a cslab machine, it receives only 25%.

- For programs that compile:

  - Correctness 40%
  - Conformance to Implementation Requirements 30%
  - Design (modularity and organization) 10%
  - Documentation: 10%
  - Style and proper naming: 10%

Submitting the Assignment

This assignment is due by the end of the day (i.e. 11:59PM, EST) on May 12, 2014. Create a directory named named *username*`_hwk3`. Put all project-related source-code files into that directory. ***Do not place any executable files or object files into this directory.*** You will lose 1% for each file that does not belong there, and you will lose 2% if you do not name the directory correctly. With all files in your directory, run the command

```
zip -r username_hwk3.zip ./username_hwk3
```

This will compress all of your files into the file named `username_hwk3.zip`.

Before you submit the assignment, make sure that it compiles and runs correctly on one of the `cslab` machines. Do not enhance your program beyond this specification. Do not make it do anything except what is written above.

You are to put your zip file into the directory

```
/data/biocs/b/student.accounts/cs335_sw/projects/project3
```

Give it permission 600 so that only you have access to it. To do this, `cd` to the above directory and run the command

`chmod 600` *`username_hwk3.zip`*

where *`username_hwk3.zip`* is the name of your zip file.

If you put a file there and then decide to change it before the deadline, just replace it by the new version. Once the deadline has passed, you cannot do this. I will grade whatever version is there at the end of the day on the due date. You cannot resubmit the program after the due date.