

Basics of C++

1.1 Summary

C++ has `for`, `while` and `do for` loops, `if` and `switch` for conditionals. We will use `char` for characters, `int` for integers, and `double` for floating-point numbers. (The latter come in alternative sizes.) There is also `bool` for boolean. The language is case-sensitive.

C++ is an extension of C. So the simplest program just has a `main` function. The program is compiled with `g++` which produces an executable `a.out`, which is run from the current directory.

The standard output is accessed by `cout`. The standard input is accessed by `cin`. These require inclusion of `iostream` library.

1.2 Sample program: `adder.cpp`

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

int main()
{
    int num1, num2;
    cout << "Enter two numbers: ";
    cin >> num1 >> num2;

    int sum = num1 + num2;
    cout << "The sum of " << num1 << " and " << num2
        << " is " << sum << endl;

    return 0;
}
```

Arrays, Pointers and Functions

2.1 Arrays

Arrays in C++ are declared to hold a specific number of the same type of object. The valid indices are 0 up to 1 less than the size of the array. The execution does no checking for references going outside the bounds of the array. Arrays can be initialized at declaration.

2.2 Functions

A function is a self-standing piece of code. It can return a variable of a specified type, or have type `void`. It can have arguments of specific type. In general, variables are passed *by value*, which means that the function receives a copy of the variable. This is inefficient for large objects, so these are usually passed *by address* (such as automatically occurs for arrays) or *by reference* (discussed later). To aid the compiler, a *prototype* of a function is used at the start of a program to tell the compiler of the existence of such a function: the prototype specifies the name, arguments, and type of the function.

2.3 Pointers

A *pointer* stores an address. A pointer has a type: this indicates the type of object stored at the address to which the pointer points. A pointer is defined using the `*`, and is dereferenced thereby too. An array name is equivalent to a pointer to the start of that array. Primitive arithmetic can be applied to pointers.

2.4 C-Strings

There are two options to store strings in C++. The first is the way done in C, now called C-strings; the second is with the object `string`, discussed later. A C-string is stored in an array of `chars`, terminated by the null character, denoted `'\0'` or simply `0` itself. The user is responsible for ensuring that the null terminator remains present. Constant strings defined by the user using quotation marks are automatically C-strings. With the `cstring` library, strings can be compared, they can be cin-ed and cout-ed, they can be copied, appended, and several other things. C-strings are passed to functions by reference: that is, by supplying the address of the first character using the array name or a char pointer.

2.5 Sample program: primality.cpp

```
// print out primes less than 100
// wdg 2009
#include <iostream>
using namespace std;
```

```
bool isPrime(int x);
const int MAX=100;
```

```
int main( )
{
    int test=2;
    while( test<MAX ) {
        if( isPrime(test) )
        cout << test << " ";
        test++;
    }
    cout << endl;
    return 0;
}
```

```
bool isPrime(int x)
{
    for(int y=2; y<x; y++)
        if( x%y==0 )
        return false;
    return true;
}
```

2.6 Sample program: strstr.cpp

```
// strstr.cpp - adapted from nkraft by wdg 2009
// Find first occurrence of substring needle in string haystack.
#include <iostream>
using namespace std;

char *strstr ( char *haystack, char *needle );
```

```

int main ( )
{
    char *one = "concatenate";
    char *two = "cat";
    char *three = "dog";

    char *ans = strstr( one, two );
    if ( !ans )
        cout << "needle1 not found\n";
    else
        cout << "needle1 starts at index " << ans-one
                << " of haystack" << endl;
    ans = strstr( one, three );
    if ( !ans )
        cout << "needle2 not found\n";
    else
        cout << "needle2 starts at index " << ans-one
                << " of haystack" << endl;

    return 0;
}

// The function returns pointer to beginning of substring,
// or NULL if substring is not found.
char *strstr ( char *haystack, char *needle )
{
    char *start;
    for(start = haystack; *start != '\0'; start++ ) {
        char *p = needle;
        char *q = start;
        while ( *p != '\0' && *q != '\0' && *p == *q ) {
            p++;
            q++;
        }
        if( *p == '\0' )
            return start;    // reached end of needle without mismatch
    }
    return NULL;
}

```

Algorithm Analysis

3.1 Order Notation

We define big-O:

$f(n)$ is $O(g(n))$ if the growth of $f(n)$ is at most the growth of $g(n)$.

The **order** of a function is the simplest smallest function that it is O of. It ignores coefficients and everything except the dominant term. Terminology: **Linear** means proportional to n . **Sublinear** means that the ratio $f(n)/n$ tends to 0 as $n \rightarrow \infty$.

|| Example. Some would say $f(n) = 2n^2 + 3n + 1$ is $O(n^3)$ and $O(n^2)$. But its order is n^2 .

3.2 Combining Functions

- **ADD.** If $T_1(n)$ is $O(f(n))$ and $T_2(n)$ is $O(g(n))$, then $T_1(n) + T_2(n)$ is $\max(O(f(n)), O(g(n)))$.
That is, when you add, the larger order takes over.
- **MULTIPLY.** If $T_1(n)$ is $O(f(n))$ and $T_2(n)$ is $O(g(n))$, then $T_1(n) \times T_2(n)$ is $O(f(n) \times g(n))$.

|| Example. $(n^4 + n) \times (3n^3 - 5) + 6n^6$ has order n^7

3.3 Logarithms

The **log base 2** of a number is how many times you need to multiply 2 together to get that number. That is, $\log n = L \iff 2^L = n$. Unless otherwise specified, computer science log is always base 2. So it gives **number of bits**. $\log n$ grows forever, but it grows slower than any power of n .

|| Example. Binary search takes $O(\log n)$ time.

3.4 Algorithm Analysis

The goal of order analysis is to determine how the running time behaves as n gets large. The value n is usually the size of the structure or the number of elements it has. For example, traversing an array takes $O(n)$ time.

We want to measure either *time* or *space* requirements of an algorithm. Time is the number of *atomic* operations executed. We cannot count everything: we just want an estimate. So, depending on the situation, one might count: arithmetic operations (usually assume addition and multiplication atomic, but not for large integer calculations); comparisons; procedure calls; or assignment statements. Ideally, pick one which is the *correct order* but *simple* to count.

|| Long Arithmetic. Long addition of two n -digit numbers is linear. Long multiplication of two n -digit numbers is quadratic.

(Check!)

3.5 Loops and Consecutiveness

- *Loop*: How many times \times average case of loop
- *Consecutive blocks*: this is the sum and hence the maximum

|| Primality Testing. The obvious algorithm is

```
bool isPrime(int N)
{
    int F=2;
    while(N%F!=0)
        F++;
    return F==N;
}
```

|| This takes $O(\sqrt{N})$ time if the number is not prime, since then the smallest factor is at most \sqrt{N} . But if the number is prime, then it takes $O(N)$ time. And, if we write the input as a B -bit number, this is $O(2^{B/2})$ time. (Can one do better?)

Example. A sequence of positive integers is a *radio sequence* if two integers the same value are at least that many places apart. Meaning, two 1s cannot be consecutive; two 2s must have at least 2 numbers in between them; etc. Here is a test of this: this method is *quadratic*, meaning $O(n^2)$.

```
for(int x=0; x<len; x++)
    for(int y=x+1; y<len; y++)
        if(array[x]==array[y] && y-x<=array[x])
            return false;
return true;
```

Program Development

4.1 Objects

Object-oriented programming rests on the three basic principles of *encapsulation*:

- *Abstraction*: ignore the details
- *Modularization*: break into pieces
- *Information hiding*: separate the implementation and the function

OOP uses the idea of classes. A *class* is a structure which houses data together with operations that act on that data. We strive for *loose coupling*: each class is largely independent and communicates with other classes via a small well-defined interface. We strive for *cohesion*: each class performs one and only one task (for readability, reuse).

We strive for *responsibility-driven design*: each class should be responsible for its own data. You should ask yourself: What does the class need to know? What does it do?

The power of OOP also comes from two further principles which we will discuss later:

- *Inheritance*: classes inherit properties from other classes
- *Polymorphism*: there are multiple implementations of methods and the correct one is executed

4.2 Literate Programming

Good programming requires extensive comments and documentation. At the very least:

explain the purpose of each instance variable, and for each method explain its purpose, parameters, returns, where applicable.

You should also strive for a consistent layout and for expressive variable names. For a class, one might list the functions, constructors and public fields, and for each method explains what it does together with pre-conditions, post-conditions, the meaning of the parameters, exceptions that may be thrown and other things.

UML is an extensive language for modeling programs especially those for an object-oriented programming language. It is a system of diagrams designed to capture objects, interaction between objects, and organization of objects, and then some.

4.3 Testing

One needs to test extensively. Look at the *boundary* values: make sure it handles the smallest or largest value the program must work for, and suitably rejects the value just out of range. Add *watches* or *debug statements* so that you know what is happening at all times. Especially look at the empty case, or the 0 input.

Basics of Classes

5.1 Data Members and Member Functions

An object is a particular instance of a class and there may be multiple instances of a class. An object has (i) data, called attributes, fields or *data members*, and (ii) functions, called methods or *member functions*. A member function is called with the `.` notation. For example:

```
object.method();
```

Every member function can access every data member. But, usually all data and some member functions are labeled **private**; user methods are labeled **public**. (There are other possibilities.) Private means that the external user of this class cannot see or use it.

5.2 More on Member Functions

Most objects have *accessor* functions: these allow the user to *get* data from the object. *Mutator* functions allow the user to *set* data in the object. One can indicate to the user that a function does not alter its arguments and/or the object by using the `const` modifier.

A variable has a *scope* (where it is accessible from) and a *lifetime* (when it exists). The variables defined in a function are called *local variables* and are accessible only within the function and exist only while the function is being executed. An exception is *static* variables whose lifetime is the program's execution: they are always available, and there is only one copy per class. If public, a static variable can be accessed by using the *scope resolution* operator: prefixing it with the classname followed by `::`.

A function should normally check its arguments. It notifies the caller of a problem by using an *exception* (discussed later) or a special return value. The programmer should try to avoid exceptions: consider error recovery and avoidance.

5.3 Constructors

A *constructor* is a special function that initializes the state of the object; it has the same name as the class, but does not have a return type. There can be more than one constructor. Note that the compiler will provide a default constructor if none is coded.

When a constructor is called, any member data is initialized before any of the commands in the body of the constructor. In particular, any member that is a class

has its constructor called automatically. (This occurs in the order that the member data is listed when defined.) So one should use specific *initializers*; this is a list after the header before the body.

```
class Foo {
public:
    Foo() : Bar(1) , ging('d')    // default constructor
    {
    }
private:
    Iso Bar;
    char ging;
};
```

5.4 Strings

There is a `string` class in the `string` library. These can be compared, cin-ed and cout-ed, assigned C-string, appended, and many other things.

5.5 Sample program: Citizen.cpp

The output is

```
The drinking age is 21
GI Joe can drink
Barbie can't drink
```

```
// Citizen.cpp - wdg - 2009
#include <iostream>
#include <string>
using namespace std;

class Citizen
{
public:
    static const int DRINKING_AGE=21;

    // constructors
    Citizen() : name("UNKNOWN"), age(0)
    {}
```

```

Citizen(string nam, int ag) : name(nam), age(ag)
{}

//mutators
void setName(string nam)
{
    name = nam;
}

void setAge(int ag)
{
    age = ag;
}

//accessors
string getName() const
{
    return name;
}

int getAge() const
{
    return age;
}

bool canDrink() const
{
    return (age>=DRINKING_AGE);
}

private:
    string name;
    int age;
};

int main( )
{
    Citizen one;

```

```
one.setName("GI Joe");
one.setAge(50);
Citizen two("Barbie",13);

cout << "The drinking age is " << Citizen::DRINKING_AGE << endl;
cout << one.getName();
if ( one.canDrink() )
    cout << " can drink\n";
else
    cout << " can't drink\n";

cout << two.getName();
if ( two.canDrink() )
    cout << " can drink\n";
else
    cout << " can't drink\n";

return 0;
}
```

C++ Files

6.1 Header Files

A C++ class is usually split over a *header file* and an *implementation file*. The header file lists the data and gives prototypes for the functions. Many people put the public members first. The implementation file gives the member function implementation. These are specified with the :: *scope resolution* operator. Your own header files should be loaded after the system ones, and the name should be in quotation marks.

Multiple inclusion of header files leads to multiple definitions of the same class, which the compiler cannot handle. So, header files should have `ifndef` as a preprocessor directive. The standard is to define a value with the same name as the file, but capitalized and with period replaced by underscore:

```
// file myHeader.h
#ifndef MYHEADER_H
#define MYHEADER_H
    ... code as before ...
#endif
```

6.2 Libraries

Mathematics. Mathematical functions are available in the library `cmath`. Note that angles are represented in radians.

More on output. Including `<iomanip>` allows one to format stream output using what are called *manipulators*. For example, `setw()` sets the minimum size of the output field, `setprecision()` sets the precision, and `fixed` ensures that a fixed number of decimal places are displayed. For example

```
double A = 4.999;
cout << setprecision(2) << showpoint;
cout << A;
```

produces 5.0 as output.

6.3 File Input

File input can be made through use of an input file stream. This is opened with the external name of the file. There are `cin`-style operators: that is, `stream >> A` reads

the variable A); note that the `stream` becomes null if the read fails. There are also functions taking the stream and a variable. The file should be closed after using. Here is code to copy a file to the standard output:

```
ifstream inFile;
inFile.open( "testing.txt" );
if( !inFile )
    cout << "Could not open file" << endl;
else {
    string oneLine;
    while( inFile.peek() != EOF ) {
        getline(inFile, oneLine);
        cout << oneLine << endl;
    }
    inFile.close();
}
```

6.4 Sample Program: DoubleList

A simple implementation of a List of double's. Note the use of a `main` method in the implementation file as a quick way to do some testing.

```
// DoubleList.h - wdg - 2009
// header for primitive List

#ifndef DOUBLELIST_H
#define DOUBLELIST_H

class DoubleList
{
public:
    static const double NO_SUCH_ELEMENT = -99.99;
    DoubleList();
    void add(double item);
    double getItem(int position) const;
    void loadFile(string fileName);

private:
    static const int MAX=100;
    double A[MAX];
    int count;
};
```

```

};

#endif

```

```

// DoubleList.cpp - wdg - 2009
// implementation code for a list of doubles

#include <fstream>
#include <iostream>
using namespace std;
#include "DoubleList.h"

DoubleList::DoubleList() : count(0)
{
}

void DoubleList::add(double item)
{
    if(count==MAX)
        return;
    A[count]=item;
    count++;
}

double DoubleList::getItem(int position) const
{
    if(position<0 || position>=count)
        return NO_SUCH_ELEMENT;
    else
        return A[position];
}

void DoubleList::loadFile(string fileName)
{
    ifstream inFile;
    inFile.open( fileName.c_str() );
    if( !inFile ) {
        cout << "Could not open file" << fileName << endl;
        return;
    }
}

```

```

    cout << "Loading ";
    double temp;
    while( (count<MAX) && (inFile >> temp) ) {
        cout << temp << " ";
        A[count]=temp;
        count++;
    }
    cout << endl;
    inFile.close();
}

// a very inadequate test program
int main( )
{
    DoubleList B; // calls constructor
    B.loadFile("numbers.txt"); // integers 1 up to 10
    B.add(1.414);
    cout << B.getItem(0) << " " << B.getItem(10) << endl;
    return 0;
}

```

More on Classes

We discuss the problems of comparing, copying, passing, outputting, and destructing objects.

7.1 Pointers and References

In C++ functions, you can pass parameters by value or by address/reference. Pass-by-value uses a separate copy of the variable; changing this copy does not affect the variable in the calling function. Pass by value is inefficient if the object is large.

Pass-by-reference/address provides access to the original variable; changing the variable in the function does affect the variable in the calling function.

In C, pass-by-reference/address is achieved by pointers. This is still used. For example, we saw that arrays are implicitly passed this way. C++ introduced the idea of references or aliases. This is a simplified version of a pointer.

In C++, a function can return an object or a pointer to it. However, it is also common to have pass-by-reference arguments that are changed.

It is also possible to return a reference to an object. However, note that an object created using `new` exists in global memory; the `new` operator returns a pointer (and `malloc` and `calloc` are not needed). On the other other, an object created with a declaration exists in local memory; in particular it is automatically destroyed at the end of its scope. Thus one gets a compiler warning if one returns a reference to a local variable. Nevertheless, there are times when return a reference can be used. For example, a nice way to print an object is to overload the `<<` operator.

7.2 Operator Overloading

In general, the term **overloading** means having multiple functions with the same name (but different parameter lists). For example, this can be used to add the usual mathematical operators for a user-defined class. Thus, one might have the prototype:

```
Fraction operator+(const Fraction & other) const;
```

If the user has some code where two fractions are added, e.g. `A+B`, then this function is called on `A`, with `B` as the argument. In the actual code for the function, the data members of the first fraction are accessed direct; those of the second are accessed with `other.` notation.

7.3 Equality Testing

To allow one to test whether two objects are equal, one should provide a function that tests for equality. In C++, this is achieved by overloading the `==` function. The argument to the `==` function is a reference to another such object.

```
class Foo {
    int bar;
    bool operator== ( const Foo &other ) const
    {
        return (bar == other.bar);
    }
};
```

Most binary operators are *left-to-right associative*. It follows that when in the calling function we have

```
Foo X,Y;
if( X==Y )
```

the boolean condition invokes `X.operator==(Y)`

7.4 Outputting a Class

Output of a class can be achieved by overloading the stream insertion operator `<<`. This is usually a separate function which needs to be made a *friend* of your class: a friend has access to private variables and functions.

```
class Foo {
    private:
        int bar1,bar2;
    friend ostream &operator<< (ostream &, const Foo &);
};

ostream &operator<< (ostream &out, const Foo &myFoo)
{
    out << myFoo.bar1 << ":" << myFoo.bar2 << endl;
    return out;
}
```

Note that the arguments are passed by reference, and the stream itself is returned by reference (so that it works with successive `<<`).

7.5 Copying and Cloning

When a class is passed by value (into or out of a function), a copy is made using the *copy constructor*. Often the default compiler-inserted copy constructor is fine. However, if the class has pointers to other classes, then probably one should create a copy constructor oneself.

```
class Foo
{
    private:
        Bar *barPtr;
    public:
        Foo( const Foo &other ) {
            barPtr = new Bar( *(other.barPtr) );
        }
};
```

7.6 Destructors

In C++, anything that is created implicitly (such as passing by value or declaring a variable) is automatically deleted when no longer in use. However, anything created with a `new` command must be released to the system to avoid memory leaks. The `delete` command takes a pointer and deletes what the pointer points to.

A class should have a *destructor*; this has the name of the class preceded by a tilde, and is called to properly release memory. For example, our linked list classes will need a destructor. This frees up the nodes when the linked-list object is deleted.

7.7 Sample Code: Fraction.cpp

We create a class called `Fraction`. In what follows we have first the header file `Fraction.h`, then the implementation file `Fraction.cpp`, and then a sample program that uses the class `TestFraction.cpp`. Note that the compiler is called by

```
g++ Fraction.cpp TestFraction.cpp
```

The fraction is stored in simplest form.

```
// Fraction.h - wdg 2009
#ifndef FRACTION_H
#define FRACTION_H

#include <iostream>
```

```

using namespace std;

class Fraction
{
public:
    Fraction(int whole);
    Fraction(int n,int d);
    Fraction(const Fraction & other);
    Fraction operator+(const Fraction & other) const;
    Fraction operator*(const Fraction & other) const;
    bool operator==(const Fraction & other ) const;

    friend ostream & operator<< (ostream & out, const Fraction & fraction);

private:
    int numer;
    int denom;
};

#endif



---


// Fraction.cpp - wdg 2009
#include "Fraction.h"

int gcd(int a,int b)
{
    if(b==0)
        return a;
    else
        return gcd (b, a%b) ;
}

Fraction::Fraction(int whole) : numer(whole), denom(1)
{
}

Fraction::Fraction(int n,int d)
{
    if(d<0) {

```

```

        d=-d; n=-n;
    }
    int h = gcd(n<0?-n:n,d);
    numer=n/h;
    denom=d/h;
}

Fraction::Fraction(const Fraction & other) : numer(other.numer), denom(other.denom)
{
}

ostream & operator<< (ostream & out, const Fraction & fraction)
{
    if(fraction.denom==1)
        out << fraction.numer;
    else
        out << fraction.numer << "/" << fraction.denom;
    return out;
}

Fraction Fraction::operator+(const Fraction & other) const
{
    int newNumer = numer*other.denom + denom*other.numer;
    int newDenom = denom * other.denom;
    return Fraction(newNumer,newDenom);
}

Fraction Fraction::operator*(const Fraction & other) const
{
    int newNumer = numer*other.numer;
    int newDenom = denom * other.denom;
    return Fraction(newNumer,newDenom);
}

bool Fraction::operator==(const Fraction & other) const
{
    return (numer==other.numer && denom==other.denom);
}

```

```

#include <iostream>

```

```
using namespace std;
#include "Fraction.h"

int main( )
{
    Fraction F(1), G(3,-5), H(7,28), I(H);
    F = F+G;
    G = I*H;
    cout << F << " " << G << " " << H << " " << I << endl;
    cout << boolalpha << (F==I);
    return 0;
}
```

Collections

8.1 Basic Collections

There are three basic collections.

1. The basic collection is often called a *bag*. It stores objects with no ordering of the objects and no restrictions on them.
2. Another unstructured collection is a *set* where repeated objects are not permitted: it holds at most one copy of each item. A set is often from a predefined universe.
3. A collection where there is an ordering is often called a *list*. Specific examples include an *array*, a *vector* and a *sequence*. These have the same idea, but vary as to the methods they provide and the efficiency of those methods.

8.2 The Bag ADT

An ADT or *abstract data type* defines a way of storing data: it specifies only how the ADT can be used and says nothing about the implementation of the structure. (An ADT is more abstract than a Java specification or C++ list of class member function prototypes.)

The Bag ADT might have:

- accessor methods such as `size`, `countOccurrence`, possibly an iterator (which steps through all the elements);
- modifier methods such as `add`, `remove`, and `addAll`; and
- also a `union` method which combines two bags to produce a third.

8.3 The Array Implementation

A common implementation of a collection is a *partially filled array*. This is often expanded every time it needs to be, but rarely shrunk. It has a pointer/counter which keeps track of where the real data ends.

0	1	2	3	4	5	6
Amy	Bo	Carl	Dana	NULL	NULL	NULL

count=4

8.4 Sample Program: StringSet

An array-based implementation of a set of strings.

```
// sample code for a set of strings
// uses partially filled array
#ifndef STRINGSET_H
#define STRINGSET_H
#include <string>
using namespace std;

class StringSet {
public:
    StringSet();
    StringSet(const StringSet &other);
    ~StringSet();
    bool contains(string item) const;
    void add(string item);
    void remove(string item);
    bool operator== ( const StringSet &other ) const;

private:
    static const int MAX_COUNT=100;
    int count;
    string *A;

    friend ostream &operator<< (ostream &, const StringSet &myStringSet);
};

#endif

// sample code for a set of strings
// uses partially filled array
// not order preserving
// not the greatest
// wdg 2009

#include <string>
#include <iostream>
#include "StringSet.h"
using namespace std;
```

```

StringSet::StringSet() : count(0), A(new string[MAX_COUNT])
{
}

StringSet::StringSet( const StringSet &other)
                    : count( other.count), A(new string[MAX_COUNT])
{
    for(int i=0; i<count; i++)
        A[i] = other.A[i];
}

StringSet::~~StringSet()
{
    delete [] A;
}

bool StringSet::contains(string item) const
{
    for(int i=0; i<count; i++)
        if( item==A[i] )
            return true;
    return false;
}

void StringSet::add(string item)
{
    if( contains(item) )
        return;
    A[count++]=item;
}

void StringSet::remove(string item)
{
    for(int i=0; i<count; i++)
        if( item==A[i] ) {
            A[i]=A[count-1];
            count--;
            return;
        }
}

```

```

        }
    }

bool StringSet::operator==( const StringSet &other ) const
{
    if( count != other.count )
        return false;
    for(int i=0; i<count; i++)
        if( !other.contains(A[i]) )
            return false;
    return true;
}

ostream &operator<< (ostream &out, const StringSet &myStringSet)
{
    for(int i=0; i<myStringSet.count-1; i++)
        out << myStringSet.A[i]+":";
    out << myStringSet.A[myStringSet.count-1];
    return out;
}

```

```

// TestStringSet.cpp - wdg - 2008

```

```

#include <iostream>
using namespace std;
#include "StringSet.h"

```

```

int main( )
{
    StringSet S;
    S.add("A"); S.add("B"); S.add("A"); S.add("D"); S.add("C");
    S.remove("D"); S.remove("A");
    StringSet T(S);
    T.remove("dummy"); T.remove("C"); T.add("C");

    cout << "S is " << S << endl;
    cout << "T is " << T << endl;
    cout << (S==T ? "Equal" : "OOOPPPSSS" ) << endl;
    return 0;
}

```

Recursion

Often in solving a problem one breaks up the problem into subtasks. *Recursion* can be used if one of the subtasks is a *simpler version* of the original problem.

9.1 An Example

Suppose we are trying to sort a list of numbers. We could first determine the minimum element; and what remains to be done is to sort the remaining numbers. So the code might look something like this:

```
void sort(Collection &C) {
    min = C.getMinimum();
    cout << min;
    C.remove(min);
    sort(C);
}
```

Every recursive method needs a *stopping case*: otherwise we have an infinite loop or an error. In this case, we have a problem when C is empty. So one always checks to see if the problem is simple enough to solve directly.

```
void sort(Collection &C) {
    if( C.isEmpty() )
        return;
    ... // as before
```

Example. Printing out a decimal number. The idea is to extract one digit and then recursively print out the rest. It's hard to get the most significant digit, but one can obtain the least significant digit (the “ones” column): use `num % 10`. And then `num/10` is the “rest” of the number.

```
void print( int n ) {
    if( n>0 ) {
        print ( n/10 );
        cout << n%10 ;
    }
}
```

9.2 Tracing Code

It is important to be able to *trace* recursive calls: step through what is happening in the execution. Consider the following code:

```
void g( int n ) {  
    if( n==0 ) return;  
    g(n-1);  
    cout << n;  
}
```

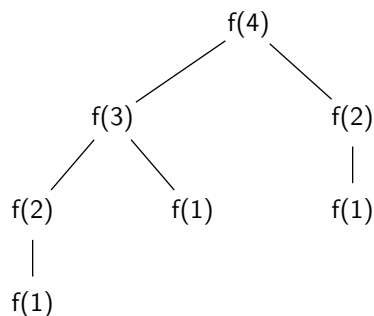
It is not hard to see that, for example, `g(3)` prints out the numbers from 3 down to 1. But, you have to be a bit more careful. The recursive call occurs before the value 3 is printed out. This means that the output is from smallest to biggest.

1
2
3

Here is some more code to trace:

```
void f( int n ) {  
    cout << n;  
    if(n>1)  
        f(n-1);  
    if(n>2)  
        f(n-2);  
}
```

If you call the method `f(4)`, it prints out 4 and then calls `f(3)` and `f(2)` in succession. The call to `f(3)` calls both `f(2)` and `f(1)`, and so on. One can draw a *recursion tree*: this looks like a family tree except that the children are the recursive calls.



Then one can work out that `f(1)` prints 1, that `f(2)` prints 21 and `f(3)` prints 3211. What does `f(4)` print out?

Exercise

Give recursive code so that `brackets(5)` prints out `(((((()))))`.

9.3 Methods that Return Values

Some recursive methods return values. For example, the sequence of Fibonacci numbers 1, 1, 2, 3, 5, 8, 13, 21, ... is defined as follows: the first two numbers are 1 and 1, and then each next number is the sum of the two previous numbers. There is obvious recursive code for the Fibonacci numbers:

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

WARNING: Recursion is often easy to write (once you get used to it!). But occasionally it is very inefficient. For example, the code for `fib` above is terrible. (Try to calculate `fib(30)`.)

9.4 Application: The Queens problem

One can use recursion to solve the *Queens* problem. The old puzzle is to place 8 queens on a 8×8 chess/checkers board such that no pair of queens attack each other. That is, no pair of queens are in the same row, the same column, or the same diagonal.

The solution uses search and *backtracking*. We know we will have exactly one queen per row. The recursive method tries to place the queens one row at a time. The main method calls `place(0)`. Here is the code/pseudocode:

```
void place(int row) {
    if(row==8)
        celebrateAndStop();
    else {
        for( queen[row] = all possible vals ) {
            check if new queen legal;
            record columns and diagonals it attacks;
            // recurse
            place(row+1);
        }
    }
}
```

```
        // if reach here, have failed and need to backtrack
        erase columns and diagonals it attacks;
    }
}
```

9.5 Application: The Steps problem

One can use recursion to solve the Steps problem. In this problem, one can take steps of specified lengths and has to travel a certain distance exactly (for example, a cashier making change for a specific amount using coins of various denominations).

The code/pseudocode is as follows

```
bool canStep(int required)
{
    if( required==0 )
        return true;
    for( each allowed length )
        if( length<=required && canStep(required-length) )
            return true;
    //failing which
    return false;
}
```

The recursive boolean method takes as parameter the remaining distance required, and returns whether this is possible or not. If the remaining distance is 0, it returns true. Else it considers each possible first step in turn. If it is possible to get home after making that first step, it returns true; failing which it returns false. One can adapt this to actually count the minimum number of steps needed. See code below.

One can also use recursion to explore a maze or to draw a snowflake fractal.

Example Program: StepsByRecursion.cpp

```
// StepsByRecursion.cpp
// a recursive program to solve Steps problem
// wdg 2008
#import <iostream>
using namespace std;

static const int IMPOSSIBLE=-1;
```

```

// @pre: required>=0
// @returns: minimum number of steps; a value of -1 means impossible
int minSteps(int required, int allowed[], int numAllowed) {
    if(required==0)
        return 0;
    else {
        int min = required+1; //guaranteed to be too large
        // consider all valid first steps
        for(int poss=0; poss<numAllowed; poss++)
            if( allowed[poss]<=required ) {
                // call recursively to determine best way to do remainder
                int temp = minSteps( required-allowed[poss], allowed, numAllowed);
                if(temp<min && temp!=IMPOSSIBLE)
                    min=temp;
            }
        // return
        if(min==required+1)
            return IMPOSSIBLE;
        else
            return 1+min;
    }//else
}

int main( )
{
    int coins[] = {1,5,10};
    cout << "USCoins do 17 in " << minSteps(17,coins,3) << endl;
        // dime, nickel and 2 pennies
    int silly[] = {2,4,6};
    cout << "Impossible coins returns " << minSteps(17,silly,3) << endl;
        // odd is impossible
}

```

Linked Lists

10.1 Links and Pointers

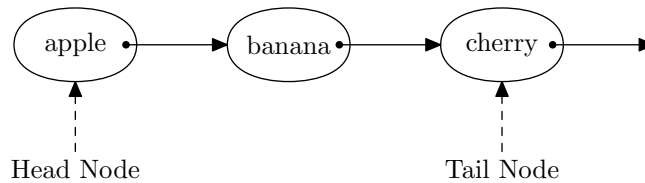
The linked list is not an ADT in its own right; rather it is a way of implementing many data structures. It is designed to *replace* an array.

A linked list is

a sequence of nodes each with a link to the next node.

These links are also called pointers. Both metaphors work. They are links because they go from one node to the next, and because if the link is broken the rest of the list is lost. They are called pointers because this link is (usually) one-directional—and, of course, they are pointers in C/C++.

The first node is called the *head node*. The last node is called the *tail node*. The first node has to be pointed to by some external holder; often the tail node is too.



One can use a **struct** or **class** to create a node. We use a **struct**, the predecessor to objects; the syntax is similar to classes (except that its members are public).

```

struct Node {
    <data>
    Node *link;
};
  
```

(where <data> means any type of data, or multiple types). The class using or creating the linked list then has the declaration:

```

Node *head;
  
```

10.2 Insertion and Traversal

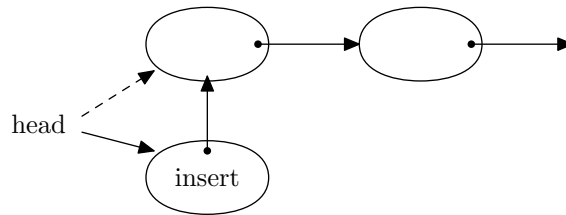
For traversing a list, the idea is to initialize a pointer to the first node (pointed to by **head**). Then repeatedly advance to the next node. **NULL** indicates you've reached the end. Such a pointer/reference is called a *cursor*. There is a standard construct for a *for-loop* to traverse the linked list:

```

for( cursor=head; cursor!=NULL; cursor=cursor->link ){
  <do something with object referenced by cursor>
}

```

For insertion, there are two separate cases to consider: (i) addition at the root, and (ii) addition elsewhere. For addition at the root, one creates a new node, changes its pointer to where head currently points, and then gets head to point to it.



In code:

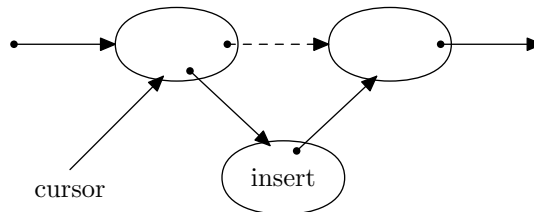
```

Node *insertPtr = new Node;
update insertPtr's data
insertPtr->link = head;
head = insertPtr;

```

This code also works if the list is empty.

To insert elsewhere, one needs a reference to the node **before** where one wants to insert. One creates a new node, changes its pointer to where the node before currently points, and then gets the node before to point to it.



In code, assuming `cursor` references node before:

```

Node *insertPtr = new Node;
update insertPtr's data
insertPtr->link = cursor->link;
cursor->link = insertPtr;

```

10.3 Traps for Linked Lists

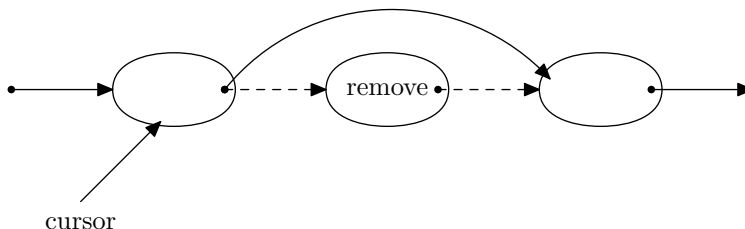
1. You **must** think of and test the exceptional cases: The empty list, the beginning of the list, the end of the list.
2. Draw a diagram: you have to get the picture right, and you have to get the order right.

10.4 Removal

The easiest case is removal of the first node. For this, one simply advances the head to point to the next node. However, this means the first node is no longer referenced; so one has to release that memory:

```
Node *removePtr = head;
head = head->link;
delete removePtr;
```

In general, to remove a node that is elsewhere in the list, one needs a reference to the node **before** the node one wants to remove. Then, to skip that node, one needs only to update the link of the node before: that is, get it to point to the node after the one wants to delete.



If the node before is referenced by `cursor`, then `cursor->link` refers to the node to be deleted, and `cursor->link->link` refers to the node after. Hence the code is:

```
Node *removePtr = cursor->link;
cursor->link = cursor->link->link;
delete removePtr;
```

The problem is to organize `cursor` to be in the correct place. In theory, one would like to traverse the list, find the node to be deleted, and then back up one: but that's not possible. Instead, one has to look one node ahead. And then beware `NULL` pointers. See sample code.

10.5 And Beyond

Arrays are better at *random access*: they can provide an element, given a position, in constant time. Linked lists are better at additions/removals at the cursor: done in constant time. Resizing arrays can be inefficient (but is “on average” constant time).

Doubly-linked lists have pointer both forward and backward. These are useful if one needs to traverse the list in both directions, or to add/remove at both ends.

Dummy header/tail nodes are sometimes used. These allow some of the special cases (e.g. empty list) to be treated the same as a typical case. While searching takes a bit more care, both removal and addition are simplified.

Exercise. Develop code for making a copy of a list.

Sample Code: MyLinkedBag

```
// MyLinkedBag.h - wdg 2008
#ifndef MY_LINKED_BAG_H
#define MY_LINKED_BAG_H
#include <iostream>
using namespace std;

typedef int ListType; // produces a bag of ints
struct BagNode
{
    ListType data;
    BagNode *next;
};

class MyLinkedBag
{
public:
    MyLinkedBag();
    ~MyLinkedBag(); // should be virtual; discussed later
    void add(ListType x);
    bool remove(ListType x);
    bool isEmpty() const;

    friend ostream & operator<< ( ostream & , const MyLinkedBag & );

private:
    BagNode *head; // head of the list
};
#endif



---


#include "MyLinkedBag.h"
//#include <cstdint>
//using namespace std;

// Constructor
MyLinkedBag::MyLinkedBag() : head(NULL)
{
```

```

}

// Destructor
MyLinkedListBag::~MyLinkedListBag()
{
    while( head!=NULL ) {
        BagNode *hold = head->next;
        delete head;
        head = hold;
    }
}

// Mutator methods:
void MyLinkedListBag::add(ListType x) // adds at front
{
    BagNode *newPtr = new BagNode;
    newPtr->data = x;
    newPtr->next = head;
    head = newPtr;
}

bool MyLinkedListBag::remove(ListType x)
{
    if( head==NULL )
        return false;
    if( head->data==x ) {
        BagNode *hold = head;
        delete head;
        head = hold;
        return true;
    }
    // so list not empty and not first node to be deleted
    BagNode *cursor = head;
    while( cursor->next != NULL && cursor->next->data != x )
        cursor = cursor->next;
    if(cursor->next == NULL)
        return false;
    else {
        BagNode *hold = cursor->next->next;

```

```

        delete ( cursor->next );
        cursor->next = hold;
        return true;
    }
}

// Accessor methods:
bool MyLinkedBag::isEmpty() const    // Returns: whether the list is empty.
{
    return (head==NULL);
}

ostream & operator<< ( ostream & out, const MyLinkedBag & other)
{
    for( BagNode *cursor = other.head; cursor; cursor=cursor->next)
        out << cursor->data << " ";
    out << endl;
}

```

Stacks

A *linear* data structure is one which is ordered. There are two special types with restricted access: a stack and a queue.

11.1 Stacks Basics

A *stack* is a data structure of ordered items such that items can be inserted and removed only at one end (called the *top*). It is also called a *LIFO* structure: last-in, first-out.

The standard (and usually only) modification operations are:

- **push**: add the element to the top of the stack
- **pop**: remove the top element from the stack and return it

If the stack is empty and one tries to remove an element, this is called *underflow*. Another common operation is called **peek**: this returns a reference to the top element on the stack (leaving the stack unchanged).

A simple stack algorithm could be used to *reverse* a word: push all the characters on the stack, then pop from the stack until it's empty.

$$\text{t h i s} \rightarrow \begin{array}{|c|} \hline \text{s} \\ \hline \text{i} \\ \hline \text{h} \\ \hline \text{t} \\ \hline \end{array} \rightarrow \text{s i h t}$$

11.2 Implementation

A stack is commonly and easily implemented using either an array or a linked list. In the latter case, the head points to the top of the stack: so addition/removal (push/pop) occurs at the head of the linked list.

11.3 Application: Balanced Brackets

A common application of stacks is the parsing and evaluation of arithmetic expressions. Indeed, compilers use a stack in *parsing* (checking the syntax of) programs.

Consider just the problem of checking the brackets/parentheses in an expression. Say $[(3+4)*(5-7)]/(8/4)$. The brackets here are okay: for each left bracket there is a matching right bracket. Actually, they match in a specific way: two pairs of matched brackets must either nest or be disjoint. You can have $[()]$ or $[]()$, but not $([])$

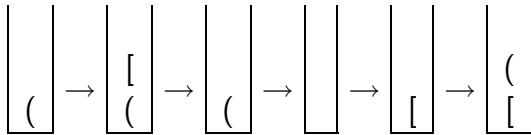
We can use a stack to store the unmatched brackets. The algorithm is as follows:

Scan the string from left to right, and for each char:

1. If a left bracket, push onto stack
2. If a right bracket, pop bracket from stack
(if not match or stack empty then fail)

At end of string, if stack empty and always matched, then accept.

For example, suppose the input is: $([])[()]$ Then the stack goes:



and then a bracket mismatch occurs.

11.4 Application: Evaluating Arithmetic Expressions

Consider the problem of evaluating the expression: $((3+8)-5)*(8/4)$. We assume for this that the brackets are compulsory: for each operation there is a surrounding bracket. If we do the evaluation by hand, we could:

repeatedly evaluate the first closing bracket and substitute

$$(((3+8)-5)*(8/4)) \rightarrow ((11-5)*(8/4)) \rightarrow (6*(8/4)) \rightarrow (6*2) \rightarrow 12$$

With two stacks, we can evaluate each subexpression when we reach the closing bracket:

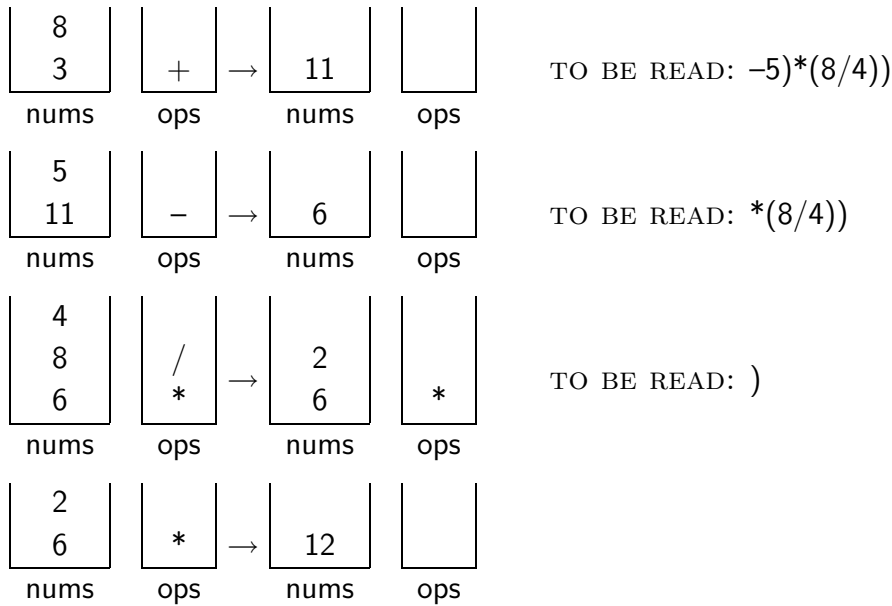
Algorithm (assuming brackets are correct!) is as follows:

Scan the string from left to right and for each char:

1. If a left bracket, do nothing
2. If a number, push onto *numberStack*
3. If an operator, push onto *operatorStack*
4. If a right bracket, do an evaluation:
 - a.) pop from the *operatorStack*
 - b.) pop two numbers from the *numberStack*
 - c.) perform the operation on these numbers (in the right order)
 - d.) push the result back on the *numberStack*

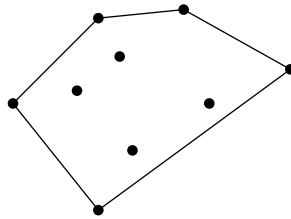
At end of string, the single value on the *numberStack* is the answer.

The above example $((3+8)-5)*(8/4)$: at the right brackets



11.5 Application: Convex Hulls

The *convex hull* of a set of points in the plane is a polygon. One might think of the points as being nails sticking out of a wooden board: then the convex hull is the shape formed by a tight rubber band that surrounds all the nails. For example, the highest, lowest, leftmost and rightmost points are on the convex hull. It is a basic building block of several graphics algorithms.



One algorithm to compute the convex hull is Graham's scan. It is an application of a stack. Let 0 be the leftmost point (which is guaranteed to be in the convex hull) and number the remaining points by angle from 0 going counterclockwise: 1, 2, ..., n - 1. Let nth be 0 again.

GRAHAM SCAN

1. Sort points by angle from 0
2. Push 0 and 1. Set $i=2$
3. While $i \leq n$ do:
 - If i makes left turn w.r.t. top 2 items on stack
 - then { push i ; $i++$ }
 - else { pop and discard }

We do not attempt to prove that the algorithm works. The running time: Each time the while loop is executed, a point is either stacked or discarded. Since a point is looked at only once, the loop is executed at most $2n$ times. There is a constant-time method for checking, given three points in order, whether the angle is a left or a right turn. This gives an $O(n)$ time algorithm, apart from the initial sort which takes time $O(n \log n)$.

One day I'll add an example.

11.6 Sample Code: ArrayStack and Balanced

Here is code for an array-based stack, and a balanced brackets tester.

```
// ArrayStack.h
// wdg 2009
#ifndef ARRAYSTACK_H
#define ARRAYSTACK_H

const int MAX_STACK = 100;
typedef int StackType;
const StackType ERROR = -1;

class ArrayStack
{
public:
    ArrayStack( );           // constructor
    void push(StackType item); // pushes object onto stack
    bool isEmpty() const;    // returns whether stack is empty or not
    StackType pop();         // pops top element from stack
    StackType peek() const;  // returns value of data on top of stack
    void dump() const;       // outputs representation on stdout

private:
    StackType arr[MAX_STACK]; // stores data
    int count;                // number of elements in stack
    // note that valid data always in 0..count-1
};
#endif
```

```
// ArrayStack.cpp
// minimal array implementation of Stack; no test for overflow
```

```

// wdg 2009
#include <iostream>
using namespace std;
#include "ArrayStack.h"

// constructor
// arbitrarily sets upper limit
ArrayStack::ArrayStack( ) : count(0)
{
    // arr not initialized
}

// pushes object onto stack
// @post: reference to item placed at right-end of array
void ArrayStack::push( StackType item )
{
    arr[count++] = item;    // count incremented after array access
}

//@ returns whether stack is empty or not
bool ArrayStack::isEmpty( ) const
{
    return (count==0);
}

// pops top element from stack
// @returns: previous top element or ERROR if problem
// @post: the top element is removed
StackType ArrayStack::pop( )
{
    if( isEmpty() )
        return ERROR;
    else
        return arr[--count];    // count decremented before array access
}

// @returns reference to data on top of stack
// or ERROR (constant in header) if stack is empty
StackType ArrayStack::peek( ) const

```

```

{
    if( isEmpty() )
        return ERROR;
    else
        return arr[count-1];
}

// outputs representation
void ArrayStack::dump() const
{
    if( isEmpty() )
        cout << "-empty-" << endl;
    else {
        for(int x=0; x<count; x++)
            cout << arr[x] << " ";
        cout << endl;
    }
}
}

```

```

// Balanced.cpp
// a hurried implementation of balanced brackets
#include <iostream>
using namespace std;
#include "ArrayStack.h"

bool test(char *argg); // uses C-strings

int main(int argc, char *argv[])
{
    if( argc==1)
        cout << "Add string on command-line! (in quotes)" << endl;
    else
        cout << boolalpha << test( argv[1] ) << endl;
    return 0;
}

bool test(char *argg) // accepts balanced strings of ()[]{}
{
    ArrayStack S;
    char D;

```

```

while ( *argg ) {
    switch( *argg ) {

        case '[': case '{': case '(':
            S.push( *argg );
            break;

        case ']':
            if( S.isEmpty() )
                return false;
            D = S.pop();
            if( D!='[' )
                return false;
            break;

        case '}':
            if( S.isEmpty() )
                return false;
            D = S.pop();
            if( D!='{' )
                return false;
            break;

        case ')':
            if( S.isEmpty() )
                return false;
            D = S.pop();
            if( D!='(' )
                return false;
            break;

        default:
            return false;
    } // end switch
    argg++;
} // end while

return S.isEmpty(); // return true if reach here with empty stack
}

```

Queues

A queue is a *linear* data structure that allows items to be added only to the rear of the queue and removed only from the front of the queue. Queues are **FIFO** structures: First-in First-out. They are used in operating systems to schedule access to resources such as a printer.

12.1 Queue Methods

The two standard modification methods are:

- `void enqueue(QueueType ob)`: insert the item at the *rear* of the queue
- `QueueType dequeue()`: delete and return the item at the *front* of the queue (sometimes called the first item).

A simple task with a queue is *echoing* the input (in the order it came): repeatedly insert into the queue, and then repeatedly dequeue.

12.2 Queue Implementation as Array

The natural approach to implementing a queue is, of course, an array. This suffers from the problem that as items are enqueued and dequeued, we reach the end of the array but are not using much of the start of the array.

The solution is to allow wrap-around: after filling the array, you start filling it from the front again (assuming these positions have been vacated). Of course, if there really are too many items in the queue, then this approach will also fail. This is sometimes called a *circular array*.

You maintain two markers for the two ends of the queue. The simplest is to maintain instance variables:

- `double data[]` stores the data
- `int count` records the number of elements currently in the queue, and `int capacity` the length of the array
- `int front` and `int rear` are such that: if $\text{rear} \leq \text{front}$, then the queue is in positions `data[front] ... data[rear]`; otherwise it is in `data[front] ... data[capacity-1] data[0] ... data[rear]`

For example, the enqueue method is:

```

void enqueue(double elem)
{
    if (count == capacity)
        return;
    rear = (rear+1) % capacity ;
    data[rear] = elem ;
    count++;
}

```

Practice. As an exercise, provide the dequeue method.

12.3 Queue Implementation as Linked List

A conceptually simpler implementation is a linked list. Since we need to add at one end and remove at the other, we maintain *two* pointers: one to the front and one to the rear. The front will correspond to the head in a normal linked list (doing it the other way round doesn't work: why?).

12.4 Application: Simulation

There are two very standard uses of queues in programming. The first is in implementing certain searching algorithms. The second is in doing a simulation of a scenario that changes over time. So we examine the CarWash simulation (taken from Main).

The idea: we want to simulate a CarWash to gain some statistics on how service times etc. are affected by changes in customer numbers, etc. In particular, there is a single Washer and a single Line to wait in. We are interested in *how long on average it takes to serve a customer*.

We assume the customers arrive at random intervals but at a known rate. We assume the washer takes a fixed time.

So we create an artificial queue of customers. We don't care about all the details of these simulated customers: just their arrival time is enough.

for currentTime running from 0 up to end of simulation:

1. *toss coin to see if new customer arrives at currentTime;*
 if so, enqueue customer
2. *if washer timer expired, then set washer to idle*
3. *if washer idle and queue nonempty, then*
 dequeue next customer
 set washer to busy, and set timer
 update statistics

It is important to note a key approach to such simulations:

Wherever possible, you look ahead.

Thus when we “move” the Customer to the Washer, we immediately calculate what time the Washer will finish, and then update the statistics. In this case, it actually allows us to discard the Customer: the only pertinent information is that the Washer is busy.

Trees

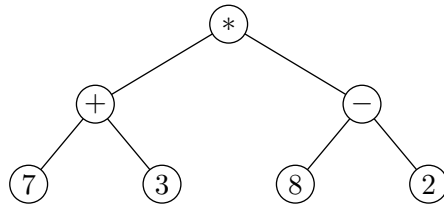
13.1 Binary Trees

A *tree* is a container of positions arranged in child–parent relationship. A tree consists of *nodes*: we speak of *parent* and *child* nodes. In a *binary* tree, each node has two possible children: a *left* and *right* child. A *leaf* node is one without children; otherwise it is an *internal* node. There is one special node called the *root*.

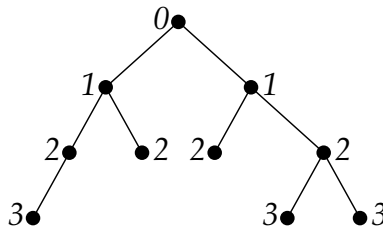
Examples include:

- father/family tree
- UNIX file system: each node is a level of grouping
- decision/taxonomy tree: each internal node is a question

For example, here is an expression tree that stores the expression $(7 + 3) * (8 - 2)$:



The *descendants* of a node are its children, their children etc. A node and its descendants form a *subtree*. A node u is *ancestor* of v if and only if v is descendant of u . The *depth* of a node is the number of ancestors (excluding itself); that is, how many steps away from the root it is. Here is a binary tree with the nodes' depths marked.



Special trees: A binary tree is *proper/full* if every internal node has two children. A binary tree is *complete* if it is full and every leaf has the same depth. (NOTE: different books have different definitions.)

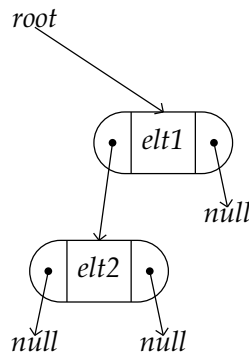


13.2 Implementation with Links

Each node contains some data and pointers to its two children. The overall tree is represented as a pointer to the root node.

```
struct BTreeNode {
    <type> data;
    BTreeNode *left;
    BTreeNode *right;
};
```

If there is no child, then that child pointer is **NULL**. It is common for tree methods to return **NULL** when a child does not exist (rather than print an error message or throw an Exception).



Methods might include:

- gets and sets (data and children)
- isLeaf
- modification methods: add or remove nodes

For a general tree, either each node contains a collection of references to children, or each node contains references to *firstChild* and *nextSibling*.

13.3 Animal Guessing Example

(Based on Main.) The computer asks a series of questions to determine a mystery animal. The data is stored as a **decision tree**. This is a full binary tree where each internal node stores a question: one child is associated with yes, one with no. Each leaf stores an animal.

The program moves down the tree, asking the question and moving to the appropriate child. When a leaf is reached, the computer has identified the animal. The cool idea is that if the program is wrong, it can automatically update the decision tree: If

the program is unsuccessful in a guess, it prompts the user to provide a question that differentiates its answer from the actual answer. Then it replaces the relevant node by a guess and two children.

Code for such a method might look something like:

```
void replace(Node *v, string quest, string yes, string no) {
    v->data = quest;
    v->left = new Node(yes);
    v->right = new Node(no);
}
```

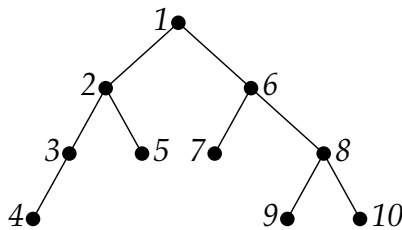
assuming a suitable constructor for the class Node.

13.4 Tree Traversals

A *traversal* is a systematic way of accessing or visiting all nodes. The three standard traversals are called preorder, inorder, and postorder. We will discuss *inorder* later.

In a *preorder traversal*, a node is visited before children (so the root is first). It is simplest when expressed using recursion. The main routine calls preorder(root)

```
preorder(Node *v) {
    visit node v
    preorder ( left child of v )
    preorder ( right child of v )
}
```

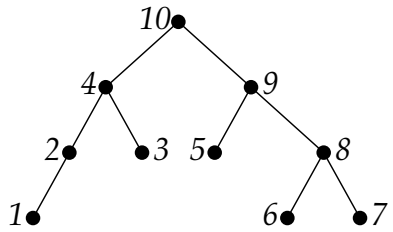


The standard application of a preorder traversal is printing a tree in a special way: for example, the indented printout below:

```

root  — left   — leftLeft
      — right  — leftRight
      — right  — rightLeft
      — right  — rightRight
  
```

The most common traversal is a *postorder traversal*. In this, each node is visited after its children (so the root is last).



Examples include computation of disk-space of directories, or maximum depth of a leaf. For the latter:

```

int maxDepth(Node *v) {
    if( v->isLeaf() )
        return 0;
    else {
        int md = 0;
        if( v->left )
            md = maxDepth (v->left) ;
        if( v->right )
            md = max ( md, maxDepth (v->right) );
        return md+1;
    }
}

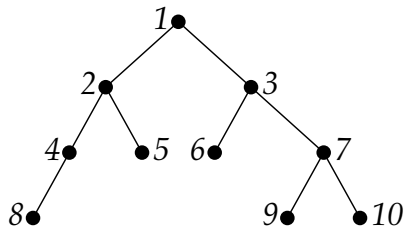
```

For the code, the time is proportional to the size of the tree, that is, $O(n)$.

13.5 Breadth-first and Depth-first Search

A *search* is a systematic way of searching through the nodes for a specific node. The two standard searches are breadth-first search and depth-first search.

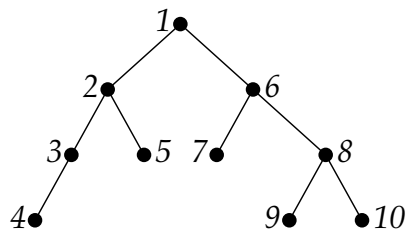
In a *breadth-first search*, the tree is visited one level at a time. It uses a *queue*: each time a node is visited, one adds its children to the queue of nodes to be visited. The next node to be visited is extracted from the front of the queue. Thus one visits the root, then the root's children, then the nodes at depth 2, and so on.



```
Add the root node to the queue.
While (queue not empty) and (goal not found) {
  current = front node removed from queue
  enqueue all of current's children (left then right)
}
```

In a *depth-first search*, the search continues going deeper into the tree whenever possible. When the search reaches a leaf, it backtracks to the last (visited) node that has un-visited children, and continues searching from there. A depth-first-search can use a *stack*: each time a node is visited, its right child is pushed onto the stack for later use while its left child is explored next. When one reaches a leaf-node, one pops off the stack.

```
Add the root node to the stack.
While (Stack not empty) and (Goal not reached) {
  current = node popped from stack
  push all of current's children (right then left)
}
```



Practice. Calculate the size (number of nodes) of the tree using recursion.

Binary Search Trees

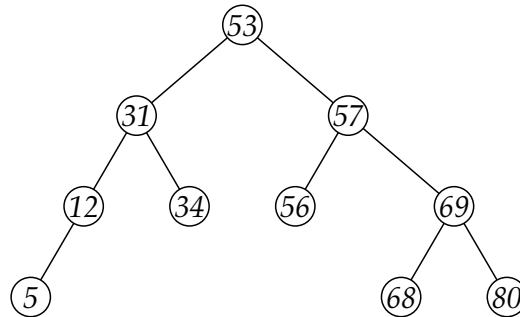
A *binary search tree* is used to store ordered data to allow efficient queries and updates.

14.1 Binary Search Trees

A *binary search tree* is a binary tree with values at the nodes such that

left descendants are smaller (or equal), right descendants are bigger (or equal).

This assumes the data comes from a domain in which there is a *total order*: you can compare every pair of elements (and there is no inconsistency such as $a < b < c < a$). In general, we could have a large object at each node, but the objects are sorted with respect to a *key*.



An *inorder traversal* is when a node is visited after its left descendants and before its right descendants. The following recursive method is started by the call `inorder(root)`.

```

void inorder(Node *v) {
    inorder ( v->left );
    visit v;
    inorder ( v->right );
}
  
```

An inorder traversal of a binary search tree prints out the data in order.

14.2 Insertion in BST

To find an element, you compare it with the root. If larger, go right; if smaller, go left. And repeat. The following method returns NULL if not found:

```
Node *find(key x) {
    Node *t=root;
    while( t!=NULL && x!=t->key )
        t = ( x<t->key ? t->left : t->right );
    return t;
}
```

Insertion is a similar process to searching, except you need a bit of look ahead. Here is a strange-looking *recursive* version:

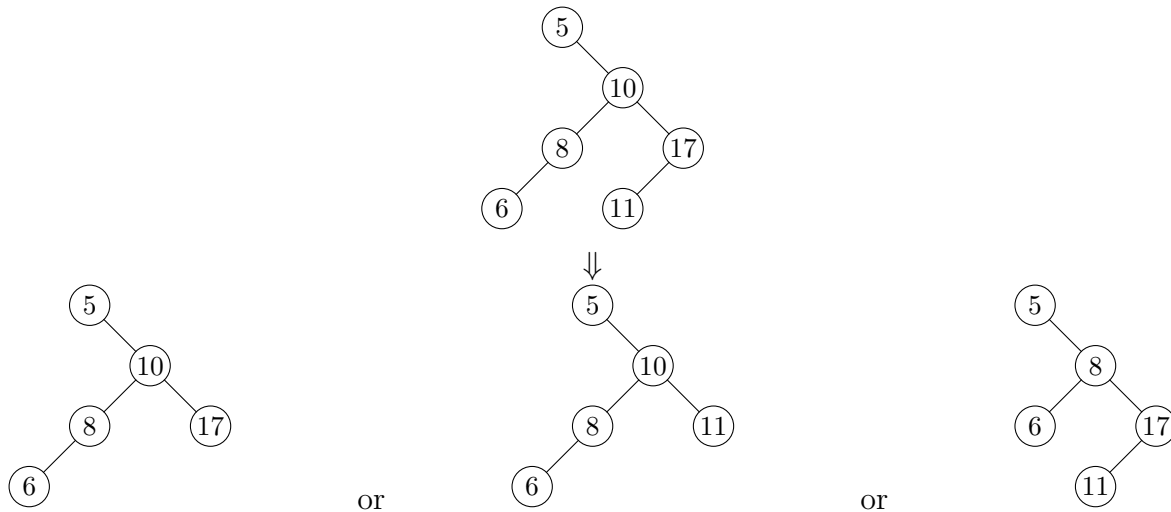
```
Node *insert(ItemType &elem, Node *t) {
    if( t==NULL )
        return new Node( elem );
    else {
        if( elem.key<t->key )
            t->left = insert(elem,t->left);
        else if( elem.key>t->key )
            t->right = insert(elem,t->right);
        return t;
    }
}
```

14.3 Removal from BST

To remove a value, one first finds the node that is to be removed. The algorithm for removing a node is divided into three cases:

- *Node is a leaf.* Then just delete.
- *Node has only one child.* Then delete the node and do “**adoption by grand-parent**” (get old parent to point to old child).
- *Node x has two children.* Then find the node y with the next-lowest value: go left, and then go repeatedly right (why does this work?). This node y cannot have two children (in fact it does not have a right child). So swop the values of x and y , and delete the node y using one of the previous cases.

The following picture shows a binary search tree and what happens if 11, 17, or 10 (assuming replace with next-lowest) is removed.



All modification operations take time proportional to depth. In best case, the depth is $O(\log n)$ (why?). But, the tree can become “lop-sided”—and so in worst case these operations are $O(n)$.

14.4 Finding the k 'th Largest Element in a Collection

Using a binary search tree, one can offer the service of finding the k 'th largest element in the collection. The idea is to keep track at each node of its size (how many nodes counting it and its descendants). This tells one where to go.

For example, if we want the 4th smallest element, and the size of the left child of the root is 2, then the value is the minimum value in the right subtree. (Why?) (This should remind you of binary search in an array.)

14.5 Sample Code: BinarySearchTree

```
// BSTNode.h
// provides node suitable for BST
// wdg 2009
#ifndef BSTNODE_H
#define BSTNODE_H
#include <cstdlib>

typedef char ItemType;
```

```

struct BSTNode
{
    BSTNode (ItemType e, BSTNode *p) :
        element(e), parent(p), left(NULL), right(NULL)
    {}
    ItemType element;
    BSTNode *left, *right, *parent;
};
#endif



---



// implementation of BST
// wdg 2009
#ifndef BINARY_SEARCH_TREE_H
#define BINARY_SEARCH_TREE_H
#include "BSTNode.h"

class BinarySearchTree
{
public:
    BinarySearchTree( );
    virtual void insertItem(ItemType it); // to allow extension
    bool deleteItem(ItemType it);
    bool findItem(ItemType it) const;
    void dumpItemsInOrder( ) const;

protected:          // to enable extension to RedBlack trees
    BSTNode *root;
    int count;

    bool insertItem(BSTNode *newNode);
    BSTNode *search( ItemType it ) const;
    void inOrder(BSTNode *v) const;
};
#endif



---



// implementation of BST
// does not allow duplicate items
// wdg 2008

#include <iostream>

```

```

#include "BinarySearchTree.h"
using namespace std;

// constructor creates an empty tree

    BinarySearchTree::BinarySearchTree( ) : root(NULL) , count(0)
    {
    }

//=====
// GENERAL BINARY SEARCH TREE METHODS
//=====
// method search starts at root looking for that item
// returns either node where found, or failing that,
// position where should be attached
// assumes root!=null

BSTNode *BinarySearchTree::search( ItemType it ) const
{
    BSTNode *v=root;
    bool absent=false;
    while( !absent && it!=v->element ) {
        if( it<v->element && v->left!=NULL )
            v = v->left;
        else if( it>v->element && v->right!=NULL )
            v = v->right;
        else
            absent = true;
    }
    return v;
}

//-----
// inserts item

void BinarySearchTree::insertItem(ItemType it)
{
    insertItem( new BSTNode(it,NULL) );
}

bool BinarySearchTree::insertItem(BSTNode *newNode)

```

```

{
    ItemType it = newNode->element;
    if(count==0) {
        count=1;
    root=newNode;
    }
    else {
        BSTNode *v = search(it);
        if( it < v->element ) {
            count++;
            v->left=newNode;
            newNode->parent = v;
        }
        else if( it > v->element ) {
            count++;
            v->right = newNode;
            newNode->parent = v;
        }
        else {
            cout << it << " duplicate" << endl;
            return false;
        }
    }
    return true;
}
//-----
// find item

bool BinarySearchTree::findItem(ItemType it) const
{
    if(count==0)
        return false;
    ItemType item = search(it)->element;
    if( item == it )
        return true;
    else
        return false;
}
//-----

```

```

// delete item

bool BinarySearchTree::deleteItem(ItemType it)
{
    BSTNode *toBeDeleted=NULL;
    if( root==NULL)
        return false;
    else if( it == root->element && ( root->left==NULL || root->right== NULL ) ) {
        // root deleted
        toBeDeleted = root;
        if( root->left!=NULL )
            root = root->left;
        else
            root = root->right;
        count--;
        delete toBeDeleted;
        return true;
    }
    else { // root not deleted
        BSTNode *parent = NULL;
        BSTNode *curr = root;
        while ( curr!=NULL && curr->element!=it) {
            parent = curr;
            if( it < curr->element )
                curr = curr->left;
            else
                curr = curr->right;
        }
        if( curr==NULL )
            return false;
        else {
            if( curr->left!=NULL && curr->right!=NULL ) {
                BSTNode *hold = curr;
                parent = curr;
                curr = curr->right;
                while ( curr->left!=NULL ) {
                    parent = curr;
                    curr = curr->left;
                }
            }
        }
    }
}

```

```

        hold->element = curr->element;
    }
    toBeDeleted = curr;
    if( curr->left!=NULL )
        curr = curr->left;
    else
        curr = curr->right;
    if( toBeDeleted == parent->left )
        parent->left = curr;
    else
        parent->right = curr;
    count--;
    delete toBeDeleted;
    return true;
}
} // end-if root not deleted
}

//=====
// OUTPUT METHODS
//=====

void BinarySearchTree::dumpItemsInOrder( ) const
{
    if( count==0 )
        cout << "Empty" << endl;
    else {
        inOrder(root);
        cout << endl;
    }
}

void BinarySearchTree::inOrder(BSTNode *v) const
{
    if( v==NULL )
        return;
    inOrder(v->left);
    cout << v->element;
    inOrder(v->right);
}

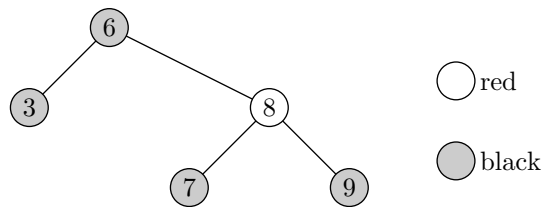
```

More Search Trees

15.1 Red-Black Trees

A **red-black tree** is a binary search tree with colored nodes where the colors have certain properties:

1. Every node is colored either red or black
2. The root is black
3. If node is red, its children must be black
4. Every down-path from root/node to NULL contains the same number of black nodes



Theorem (proof omitted): The height of a Red-black tree storing n items is at most $2 \log(n + 1)$

Therefore operations remain $O(\log n)$.

15.2 Bottom-Up Insertion in Red-Black Trees

The simplest (but not most efficient) method of insertion is called **bottom up insertion**. Start by inserting as per binary search tree and making the new leaf red. The only possible violation is that its parent is red.

This violation is solved recursively with recoloring and/or rotations. Everything hinges on the **uncle**:

1. if uncle is red (but NULL counts as black), then recolor: parent & uncle \rightarrow black, grandparent \rightarrow red, and so percolate the violation up the tree.
2. if uncle is black, then fix with suitable rotations:
 - a) if same side as parent is, then perform single rotation: parent with grandparent and swap their colors.
 - b) if opposite side to parent, then rotate self with parent, and then proceed as in case a).

Deletion is even more complex. In top-down insertion the coloring is corrected as one goes. We omit this.

Highlights of the code for red-black tree are included in the chapter on inheritance.

15.3 B-Trees

Many relational databases use B-trees as the principal form of storage structure. A B-tree is an extension of a binary search tree.

In a B-tree the top node is called the root. Each internal node has a collection of values and pointers. The values are known as *keys*. If an internal node has k keys, then it has $k + 1$ pointers: the keys are sorted, and the keys and pointers alternate. The keys are such that the data values in the subtree pointed to by a pointer lie between the two keys bounding the pointer.

The nodes can have varying numbers of keys. In a B-tree of *order* M , each internal node must have at least $M/2$ but not more than $M - 1$ keys. The root is an exception: it may have as few as 1 key. Orders in the range of 30 are common. (Possibly each node stored on a different page of memory.)

The leaves are all at the same height. This stops the unbalancedness that can occur with binary search trees. In some versions, the keys are real data. In our version, the real data appears only at the leaves.

It is straight-forward to search a B-tree. The search moves down the tree. At a node with k keys, the input value is compared with the k keys and based on that, one of the $k + 1$ pointers is taken. The time used for a search is proportional to the height of the tree.

15.4 Insertion into B-trees

A fundamental operation used in manipulating a B-tree is *splitting* a node. An internal node that is overfull (has M keys) or a leaf that is overfull, is split. In this operation the node is split into two nodes, with the smaller and larger halves respectively, and the middle value is passed to the parent as a key.

The insertion of a value into a B-tree can be stated as follows. Search for correct leaf. Insert into leaf. If overfull then split. If parent full then split it, and so on up the tree. If the root becomes overfull it is split and a new root created. This is the only time the height of the tree is increased.

Deletion from B-trees is similar but harder.

For example, if we set $M = 3$ and insert the values 1 thru 15 into the tree, we get the following preorder traversal: for internal nodes, the keys are printed out.

```
(5,9)
...(3)
.....1,2
.....3,4
...(7)
.....5,6
.....7,8
...(11,13)
.....9,10
.....11,12
.....13,14,15
```

Adding 16 causes a leaf to split, which causes its parent to split, and the height of the tree is increased:

```
(9)
...(5)
.....(3)
.....1,2
.....3,4
.....(7)
.....5,6
.....7,8
...(13)
.....(11)
.....9,10
.....11,12
.....(15)
.....13,14
.....15,16
```

The code for a B-tree implementation is included in the chapter on inheritance.

Inheritance

16.1 Inheritance and Subclasses

In C++ you can make one class an *extension* of another. This is called *inheritance*. The classes form an *is-a* hierarchy. The advantage of inheritance is that it avoids code duplication, promotes code reuse, and improves maintenance and extendibility.

Inheritance allows one to derive classes from a base class without disturbing the implementation of the base class.

```
class Derived : public Base
{
    additional instance variables;
    new constructors;
    //inherited members and functions;
    overriding methods; // replacing those in Base
    additional methods;
}
```

To create a class `Rectangle` which extends a class `Shape`, one starts with

```
class Rectangle : public Shape {
```

The class `Rectangle` then automatically has all the methods of class `Shape`, and you can add some of your own.

The methods and the instance variables of the superclass can be accessed (if not declared as `private`)— they may be declared as `protected` to allow direct access only to extensions. The default constructor for `Rectangle` is automatically called at the start of the constructor for `Shape`, unless you specify otherwise.

The derived class (*subclass*) is a new class that has some *type compatibility* in that it can be substituted for the base class (*superclass*). A pointer has a *static type* determined by the compiler. An object has a *dynamic type* which is fixed at run-time creation. A pointer reference is *polymorphic* since it can reference objects of different dynamic type. A pointer may reference objects of its declared type or any subtype of its declared type; subtype objects may be used whenever supertype objects are expected. There are times an object may need to be *cast* back to its original type. Note that the actual dynamic type of an object is forever fixed.

Suppose class `Rectangle` extends class `Shape`. Then we could do the following assignments:

```

Shape *X;
Rectangle *Y;
Y = new Rectangle();
X = Y; // okay
X = new Rectangle(); // okay
Y = static_cast<Rectangle*>(X); // cast needed

```

16.2 Overriding Functions

An important facet of inheritance is that the derived class can replace a generic function with a tailored function. *Overriding* is providing a function with the same signature as the superclass but different body.

But then a key question is which version of the function gets used. There are two options: if we declare a function is *virtual*, then which version of the function is used is determined at run-time by the object's actual dynamic type. (In Java, all functions are implicitly virtual.) If we declare a function as *nonvirtual*, then which version is determined by the compiler based on the static type of the reference. Usually a *nonvirtual* function is not intended to be overridden. Note that a function declared as virtual in the base class is automatically virtual in the derived class.

You can access in the **Derived** class the **Base** version of an overridden function by using the scope resolution operator: `Base::fooBar()`.

16.3 Interfaces and Abstract Base Classes

In Java, an interface specifies the methods which a class should contain. A class can then implement an interface (actually it can implement more than one). In doing so, it must implement every method in the interface (it can have more). This is just a special case of inheritance: the base class specifies functions but they are not implemented.

C++ uses abstract base classes. An *abstract base class* is one where only some of the functions are implemented. A function is labeled as abstract by setting it equal to 0; this tells the compiler that there will be no implementation of this function. It is called a *pure* function. An object with one or more pure functions cannot be instantiated.

Example: the abstract base class (interface)

```

class Number {
public:
    virtual void increment()=0;
    virtual void add(Number &other)=0;
    ETC
};

```

the implementation:

```
class MyInteger : public Number {
private:
    int x;
public:
    virtual void increment(){x++;}
    ETC
}
```

the calling program (but then one can only execute `Number`'s methods on `ticket`).

```
Number *ticket = new MyInteger();
ticket->increment();
```

16.4 Sample Code: RedBlackTree

Notice that `RNode` extends `BSTNode` and that `RedBlackTree` extends `BinarySearchTree`. The actual code looks ugly in places...

```
// RNode.h
// provides node suitable for red-black tree
// wdg 2008
#ifndef RBNODE_H
#define RBNODE_H

#include <iostream>
using namespace std;
#include "BSTNode.h"

enum Color { RED, BLACK, NONE }; // creates a data type and three constants

class RNode : public BSTNode
{
public:
    Color color;

    RNode (ItemType e) : BSTNode(e,NULL) , color(NONE)
    {
    }
    void dump( )
```

```

    {
        cout << element << "(" << color << ")";
    }
};

```

```
#endif
```

```

// RedBlackTree.h
// does not allow duplicate items
// wdg 2008
#include "RBNode.h"
#include "BinarySearchTree.h"

class RedBlackTree : public BinarySearchTree
{
public:
    RedBlackTree();
    void insertItem(ItemType it);

private:
    void leftRotate(RBNode *x);
    void rightRotate(RBNode *x);
    RBNode *grand(RBNode *v);
    RBNode *uncle(RBNode *v);
    bool isLeftChild(BSTNode *v);
}; // end of class

```

```

/* extract from RedBlackTree.cpp */
....

// returns the grandparent of a node
RBNode *RedBlackTree::grand(RBNode *v)
{
    if( v==root || v->parent==root)
        return NULL;
    else
        return static_cast<RBNode*> (v->parent->parent);
}

...

```

```

void RedBlackTree::insertItem(ItemType it)
{
    RBNode *x = new RBNode(it);
    if( !BinarySearchTree::insertItem(x) )
        return;
    x->color = RED;
    while
        ...
}

```

16.5 Sample Code: BTree

A primitive implementation of a B-tree. We omit the implementations of `BTreeInternal` and `BTreeLeaf`.

```

// BTreeNode.h wdg 2008
#ifndef BTREENODE_H
#define BTREENODE_H
#include <iostream>
using namespace std;

class BTreeNode
{
public:
    static const int MAX=3;
    virtual BTreeNode *insert(int item, int &newKey) = 0;
    virtual void dump(int depth) = 0;
    static void indent(int depth)
    {
        for(int i=0; i<depth; i++)
            cout << "...";
    }
};

class BTreeLeaf : public BTreeNode
{
private:

```

```

    int value[MAX+1]; // need to go temporarily over
    int elements;

public:
    BTreeNode();
    BTreeNode *insert(int item, int &newKey);
    void dump(int depth);
};

class BTreeNode : public BTreeNode
{

private:
    BTreeNode * child[MAX+1];
    int marker[MAX+1];
    int currChildren;
    // note: marker[0] is not used
    // child[0] - marker[1] - child[1] - marker[2] - child[2] etc

    friend class BTree;

public:
    BTreeNode();
    BTreeNode *insert(int item, int &newKey);
    void dump(int depth);
};

class BTree
{
private:
    BTreeNode *root;

public:
    BTree( );
    void insert(int item);
    void dump();
};
#endif

```

```

#include <iostream>
using namespace std;
#include "BTreeNode.h"

BTree::BTree( ) : root(NULL)
{
}

void BTree::insert(int item)
{
    if( root==NULL )
        root = new BTreeLeaf();
    int newKey;
    BTreeNode *split = root->insert(item, newKey);
    if(split!=NULL) {
        cout << "Splitting root " << endl;
        BTreeInternal *newRoot = new BTreeInternal();
        newRoot->child[0] = root;
        newRoot->child[1] = split;
        newRoot->marker[1] = newKey;
        newRoot->currChildren = 2;
        root = newRoot;
    }
}

void BTree::dump()
{
    root->dump(0);
}

```

Heaps

17.1 Priority Queue

The priority queue ADT supports:

- `insertItem(e)`: Insert new item `e`
- `removeMin()`: Remove and return item with minimum key (Error if priority queue is empty)
- standard `isEmpty()` and `size`, maybe peeks

Other possible methods include decrease-key, increase-key, and delete. Applications include selection, and the event queue in discrete event simulation.

There are several inefficient implementations:

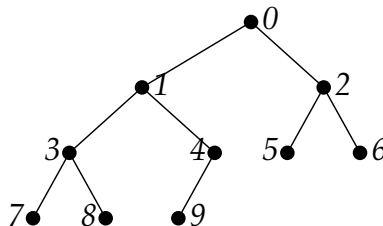
	insert	removeMin
unsorted linked list	$O(1)$	$O(n)$
sorted linked list or array	$O(n)$	$O(1)$
binary search tree	$O(n)$; average $O(\log n)$	

17.2 Heap

In *level numbering* in binary trees, the nodes are numbered such that:

for a node numbered x , its children are $2x+1$ and $2x+2$

Thus a node's parent is at $(x-1)/2$ round down, and the root is 0.

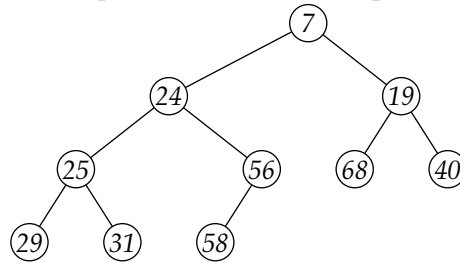


One can store a binary tree in an array/vector by storing each value at the position given by level numbering. But this is wasteful storage, unless nearly balanced.

We can change the definition of *complete binary tree* as a binary tree where each level except the last is complete, and in the last level nodes are added left to right. With this definition, a *heap* is a complete binary tree, normally stored as a vector, with values stored at nodes such that:

heap-order property: for each node, its value is smaller than or equal to its children's

So the minimum is on top. A *heap* is the standard implementation of a priority queue.



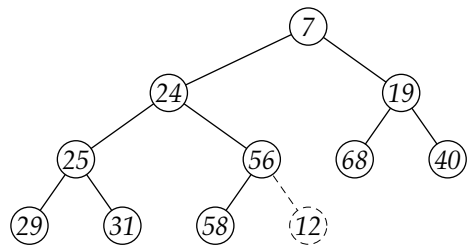
17.3 Heap Operations

The idea for *insertion* is to *Add as last leaf, then bubble up value until heap-order property re-established.*

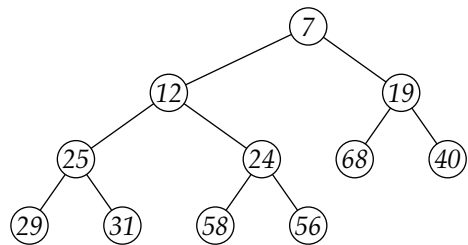
```
Algorithm: Insert(v)
  add v as next leaf
  while v < parent(v) {
    swapElements(v, parent(v))
    v = parent(v)
  }
```

Use “hole” to reduce data movement.

Here is an example of Insertion: inserting value 12:



Before



After

The idea for **removeMin** is to *Replace with value from last leaf, delete last leaf, and bubble down value until heap-order property re-established.*

Algorithm: RemoveMin()

temp = value of root

swap root value with last leaf

delete last leaf

v = root

while v > any child(v) {

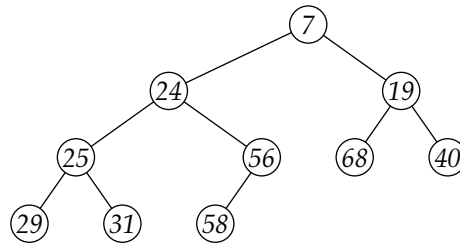
 swapElements(v, smaller child(v))

 v = smaller child(v)

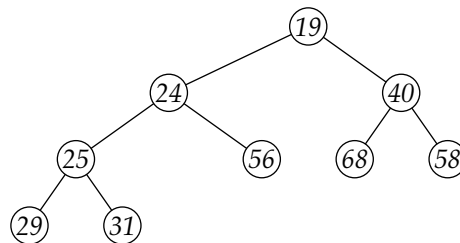
}

return temp

Here is an example of RemoveMin:



Before



After

Variations of heaps include

- d -heaps; each node has d children
- support of merge operation: leftist heaps, skew heaps, binomial queues

17.4 Heap Sort

Any priority queue can be used to sort:

Insert all values into priority queue
Repeatedly removeMin()

It is clear that inserting n values into a heap takes at most $O(n \log n)$ time. Possibly surprising is that we can create a heap in linear time. Here is one approach: work up the tree level by level, correcting as you go. That is, at each level, you push the value down until it is correct, swapping with the smaller child.

Analysis: Suppose the tree has depth k and $n = 2^{k+1} - 1$ nodes. An item that starts at depth j percolates down at most $k - j$ steps. So the total data movement is at most

$$\sum_{j=0}^k 2^j (k - j)$$

which is $O(n)$, it turns out.

Thus we get **Heap-Sort**. Note that one can *re-use* the array/vector in which heap is stored: `removeMin` moves minimum to end, and so repeated application produces sorted list in the vector

A Heap-Sort Example is:

heap					
1	3	2	6	4	5
heap					
2	3	5	6	4	1
heap					
3	4	5	6	2	1
heap					
4	6	5	3	2	1
heap					
5	6	4	3	2	1
heap					
6	5	4	3	2	1

17.5 Application: Huffman Coding

The standard binary encoding of C characters takes $\lceil \log_2 C \rceil$ bits. In a variable-length code, the most frequent characters have the shortest representation. However, now we have to decode the encoded phrase: it is not clear where one character finishes and the next one starts. In a *prefix-free code*, no code is the prefix of another code. This guarantees unambiguous decoding: indeed, the *greedy* decoding algorithm works:

traverse the string until the part you have covered so far is a valid code; cut it off and continue.

Huffman's algorithm constructs an optimal prefix-free code. The algorithm assumes we know the occurrence of each character:

Repeat

 merge the two (of the) rarest characters into a mega-character
 whose occurrence is the combined

Until only one mega-character left

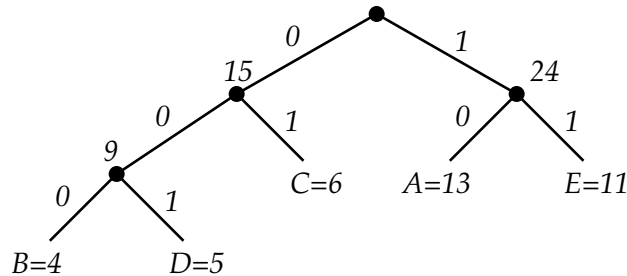
Assign mega-character the code EmptyString

Repeat

 split a mega-character into its two parts assigning each of these
 the mega-characters code with either 0 or 1

The information can be organized in a trie: this is a special type of tree in which the links are labeled and the leaf corresponds to the sequence of labels one follows to get there.

For example if 39 chars are A=13, B=4, C=6, D=5 and E=11 we get the coding A=10, B=000, C=01, D=001, E=11.



Note that a priority queue is used to keep track of the frequencies of the letters.

17.6 Sample Code: Heap

```
// PriorityQueue.h - wdg 2008
#ifndef PRIORITY_QUEUE_H
#define PRIORITY_QUEUE_H

typedef double ItemType;
const ItemType ERROR=-1;

class PriorityQueue
{
public:
    virtual void insertItem(ItemType)=0;
    virtual ItemType removeMin()=0;
    virtual bool isEmpty() const =0;
    virtual int getCount() const =0;
};

#endif

// Heap.h - wdg 2008
#ifndef HEAP_H
#define HEAP_H
#include "PriorityQueue.h"

class Heap : public PriorityQueue
```

```

{
public:
    Heap();
    void insertItem(ItemType);
    ItemType removeMin();
    bool isEmpty() const;
    int getCount() const;

private:
    int count;
    ItemType A[100];
};
#endif

```

```

// Heap.cpp - wdg 2008
#include "Heap.h"

```

```

Heap::Heap( ) : count(0)
{
}

void Heap::insertItem(ItemType newVal)
{
    int bubble = count;
    int parent = (bubble-1)/2; // round down
    while( bubble>0 && newVal<A[parent] ) {
        A[bubble] = A[parent];
        bubble = parent;
        parent = (bubble-1)/2;
    }
    A[bubble] = newVal;
    count++;
}

ItemType Heap::removeMin( )
{
    if (count==0)
        return ERROR;
    else {
        ItemType toReturn = A[0];

```

```

count--;
ItemType bubbleVal = A[count];
int bubble = 0;
int left = 1;
int right = left+1;
while( ( left < count && bubbleVal>A[left]) ||
        ( right < count && bubbleVal>A[right] ) )
    {
    int smallerChild = left;
    if( right < count && A[right]<A[left] )
        smallerChild = right;
    A[bubble] = A[smallerChild];
    bubble = smallerChild;
    left = 2*smallerChild+1;
    right = left+1;
    } // endwhile
A[bubble] = bubbleVal;
return toReturn;
}
}

bool Heap::isEmpty( ) const
{
    return (count==0);
}

int Heap::getCount( ) const
{
    return count;
}

```

Hash Tables

18.1 Dictionary

The dictionary ADT supports:

- `insertItem(e)`: Insert new item `e`
- `lookup(e)`: Look up item based on key; return access/boolean

Applications include counting how many times each word appears in a book, or the symbol table of a compiler. There are several implementations: for example, red-black trees do both operations in $O(\log n)$ time. But we can do better...

18.2 Components

The *hash table* is designed to do the unsorted dictionary ADT. A hash table consists of:

- a fixed-size array (normally prime) of *buckets*
- a *hash function* that assigns an element to a particular bucket

There will be *collisions*: multiple elements in same bucket. There are several choices for the hash function, and several choices for handling collisions.

18.3 Hash Functions

Ideally, a hash function should appear “random”! A hash function has two steps:

- convert the object to `int`.
- convert the `int` to the required range by taking it mod the table-size

A natural method of obtaining a hash code for a string is to convert each char to an `int` (e.g. ASCII) and then combine these. While concatenation is possibly the most obvious, a simpler combination is to use the sum of the individual char’s integer values. But it is much better to use a function that causes strings differing in a single bit to have wildly different hash codes.

For example, compute the expression

$$\sum_i a_i 37^i$$

where a_i are the codes for the individual letters.

18.4 Collision-Resolution

The simplest method of dealing with collisions is to put all the items with the same hash-function value into a common bucket implemented as an unsorted linked list: this is called *chaining*.

The *load factor* of a table is the ratio of the number of elements to the table size. Chaining can handle load factor near 1

EXAMPLE Suppose hashcode for a string is the string of 2-digit numbers giving letters (A=01, B=02 etc.) Hash table is size 7.

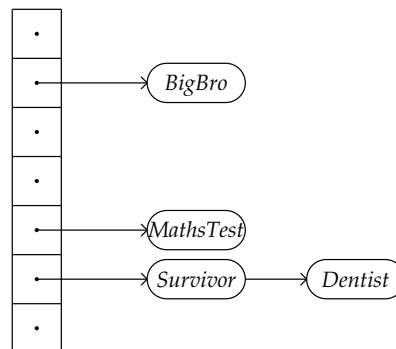
Suppose store:

BigBro = 020907021815 → 1

Survivor = 1921182209221518 → 5

MathsTest = 130120081920051920 → 4

Dentist = 04051420091920 → 5



An alternative approach to chaining is called *open addressing*. In this collision-resolution method: if intended bucket h is occupied, then try another nearby. And if that is occupied, try another one.

There are two simple strategies for searching for a nearby vacant bucket:

- *linear probing*: move down array until find vacant (and wrap-around if needed): look at $h, h + 1, h + 2, h + 3, \dots$
- *quadratic probing*: move down array in increasing increments: $h, h + 1, h + 4, h + 9, h + 16, \dots$

Linear probing causes chunking in the table, and open addressing likes load factor below 0.5.

Operations of search and delete become more complex. For example, how do we determine if string is already in table? And deletion must be done by *lazy deletion*: why?

18.5 Rehashing

If the table becomes too full, the obvious idea is to replace the array with one double the size. However, we cannot just copy the contents over, because the hash value is different. Rather, we have to go through the array and re-insert each entry.

Advance C++ Topics

We briefly consider class assignment, exceptions, templates and iterators.

19.1 Class Assignment

Suppose we have defined a class `Foo`. If we write:

```
Foo *bar1 = new Foo();
Foo *bar2 = bar1;
```

then the pointer `bar2` points to the same instance of `Foo` that `bar1` does. In particular, there is only one instance floating around.

If we write:

```
Foo bar1, bar2;
// changes to the two
bar2 = bar1;
```

then `bar2` is now a copy of `bar1`. Unless you specify otherwise, this is done by the default assignment operator, which is a *shallow copy*—only the declared variables are copied. For example, if `Foo` contains the header pointer to a linked list, the pointer will be copied, but both the header in `bar1` and the one in `bar2` will point to the same `Node` in memory. This is usually not what you want.

To create your own assignment operator, start with

```
Foo & operator= (const Foo &other) {
    // make copies of other's members and assign them to this
    return *this;
}
```

The `this` pointer always refers to the object on which the member function is being invoked. (It has to return the object, so that `A=B=C` works.) Now, one should deallocate the old stuff in the object using the `delete` command first—however, if the user writes `bar=bar`, assigning an object to itself, there is a potential problem.

Note that all languages have the shallow/deep copy issue.

19.2 Exceptions

An *exception* is an unexpected event that occurs when the program is running. In C++, an exception is treated as an object. There are exceptions already defined; it is also possible to create new ones.

An exception is explicitly thrown using a `throw` statement. A `throw` statement must specify an exception object to be thrown. All exceptions that are thrown must be eventually caught. A method might not handle an exception but instead propagate it for another method to handle.

A `try` clause is used to delimit a block of code in which a method call or operation might cause an exception. If an exception occurs within a `try` block, then C++ aborts the `try` block, executes the corresponding `catch` block and then continues with the statements that follow the `catch` block. If there is no exception, the `catch` block is ignored.

Good practice says that one should state which functions throw exceptions. This is achieved by having a `throw` clause that lists the exceptions that can be thrown by a method. Write the exception handlers for these exceptions in the program that uses the methods.

19.3 Templates

Thus far we have defined a special type for each collection. Templates let the user of a collection tell the compiler what kind of thing to store in a particular instance of a collection. If we want an set that stores strings, we say

```
set<string> S;
```

After that, the collection is used just the same as before.

The C++ code then needs templates. It is common to use a single letter for the parameter class. For example, the `Node` class might be written:

```
template <class T>
class Node
{
    public:
        Node(T initData, Node *initNext) {
            data = initData;
            next = initNext;
        }
        T data;
        Node *next;
}
```

and this is invoked with

```
template <class U>
class List
```

```
{
    Node<U> *head;
```

It is standard to break the class heading over two lines. To a large extent, you can think of `Node<>` as just a new class type.

Note that one can write code assuming that the parameter (`T` or `U`) implements various operations such as assignment, comparison, or stream insertion. These assumptions should be documented! Note that the template code is not compiled abstractly; rather it is compiled for each parameter choice separately, and the `cpp` file is included at the end of the header file.

19.4 Iterators

The idea of iterators is simply wonderful. They allow one to do the same operation on all the elements of a collection. Creating your own requires learning more C++ syntax. Using iterators is standardized. This is especially useful in avoiding working out how many elements there are: the basic idea is element access and element traversal.

In the STL (standard template) library, many structures have iterators. While the paradigm is the same for each, each iterator is a different type. A C++ iterator behaves like a pointer; it can be incremented and tested for completion by comparing with a companion iterator. The actual value is obtained by dereferencing.

Note that `begin()` returns the first element in the collection, while `end()` returns a value beyond the last element: it must not be dereferenced.

```
int addup ( vector<int> & A ) {
    int sum = 0;
    vector<int>::const_iterator start = A.begin();
    vector<int>::const_iterator stop = A.end();
    for( ; start!=stop; ++start )
        sum += *start;
    return sum;
}
```

You can leave out the `const_` part. In the above case, the `vector` template class also implements subscripting; so one could write:

```
int addup ( vector<int> & A ) {
    int sum = 0;
    for(int i=0; i<A.size(); i++ )
        sum += A[i];
    return sum;
}
```

Iterators also come up if testing whether a structure contains a particular element. Rather than having a built-in boolean `contains` function, one does:

```
template < class E >
bool contains( set<E> & B , E target ) {
    return B.find ( target ) != B.end();
}
```

We do not discuss how to create our own iterators.

19.5 Sample Code: SimpleList

```
// SimpleList.h - wdg - 2008
// header for primitive List
#ifndef SIMPLE_LIST_H
#define SIMPLE_LIST_H

template<class T>
class SimpleList
{
public:
    SimpleList(int cap);
    void add(T item);
    T getItem(int position) const;
    SimpleList<T> & operator=(const SimpleList<T> & rhs);

private:
    int count;
    int capacity;
    T *A;    // will store array
};

#include "SimpleList.cpp"

#endif



---


// SimpleList.cpp - wdg - 2008
// implementation code for a list of doubles
// uses partially filled array
```

```

#include <stdexcept>
using namespace std;

template <class T>
SimpleList<T>::SimpleList(int cap) : count(0), capacity(cap), A(new T[cap])
{
}

template <class T>
void SimpleList<T>::add(T item)
{
    if( count == capacity )
        throw invalid_argument("No room in list");
    A[count]=item;
    count++;
}

template <class T>
T SimpleList<T>::getItem(int position) const
{
    if( position<0 || position>=count)
        throw invalid_argument("Invalid position");
    return A[position];
}

template <class T>
SimpleList<T> & SimpleList<T>::operator=(const SimpleList<T> & rhs)
{
    if ( this!= &rhs ) { // uses addressof operator so that compare pointers
        capacity = rhs.capacity;
        count = rhs.count;
        delete [] A;
        A = new T[capacity];
        for(int i=0; i<count; i++)
            A[i] = rhs.A[i];
    }
    return *this;
}

```

```

// TestSimpleList.cpp - an inadequate test program - wdg 2009
#include <iostream>
using namespace std;
#include "SimpleList.h"

int main( )
{
    SimpleList<double> B(5); // calls constructor
    B.add(0);
    B.add(3.1415);
    B.add(1.414);
    cout << B.getItem(0) << " " << B.getItem(1) << " " << B.getItem(2) << endl;
    SimpleList<double> C(10);
    for(int i=0; i<10; i++)
        C.add(i);
    B=C;
    cout << B.getItem(0) << " " << B.getItem(1) << " " << B.getItem(2) << endl;
    cout << B.getItem(-99); // causes exception
    return 0;
}

```

Sorting

20.1 Five Sorting Algorithms

Here is a too brief summary of 5 sorting algorithms. (Needs to be extended!)

Insertion Sort. Add elements one at a time from right, maintaining sorted list. (E.g. using an array.) Running time: $O(n)$ – $O(n^2)$

Shellsort. Given increment sequence $h_i = n/2^i$: in phase i , do insertion sort on array split into every h_i th element. Running time: $O(n \log n)$ – $O(n^2)$

Heap Sort. buildHeap, then repeatedly deleteMin. Running time: $\Theta(n \log n)$.

MergeSort.

1. Arbitrarily split
2. Call MergeSort on each half
3. Merge two sorted halves

Running time: $O(n \log n)$. Extra space needed, but sequential access enough.

QuickSort.

1. Pick pivot (e.g. first element or middle-of-three)
2. Partition array w.r.t. pivot
3. Call QuickSort on each piece

Running time: $O(n \log n)$ – $O(n^2)$

20.2 Lower Bound for Sorting

This is the idea behind the lower bound for time needed for sorting. Given an array, sorting entails determining the rank (1 to n) of every element. There are $n!$ possibilities for the list of ranks. Each operation (such as a comparison) reduces the number of possibilities by at best a factor of 2. So we need at least $\log_2(n!)$ steps to produce the one correct possibility. (The code can be thought of as a binary decision tree.) A mathematical fact is that $\log_2(n!)$ is approximately $n \log_2 n$.

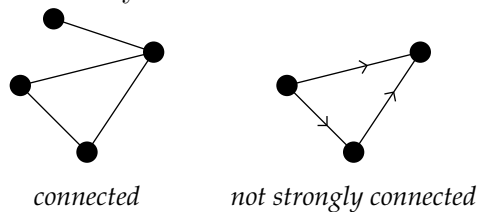
Graphs

21.1 Graphs

A graph has two parts: *vertices* (one vertex) also called *nodes*. An *undirected graph* has undirected *edges*. Two vertices joined by edge are *neighbors*. A *directed graph* has directed *edges/arcs*; each arc goes from *in-neighbor* to *out-neighbor*. Examples include:

- city map
- circuit diagram
- chemical molecule
- family tree

A *path* is sequence of vertices with successive vertices joined by edge/arc. A *cycle* is a sequence of vertices ending up where started such that successive vertices are joined by edge/arc. A graph is *connected* (a directed graph is *strongly connected*) if there is a path from every vertex to every other vertex.



21.2 Graph Representation

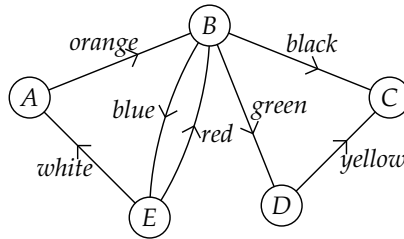
ADJACENCY MATRIX

- 1) container of vertices, numbered
- 2) array where each entry has info about the corresponding edge

ADJACENCY LIST

- 1) container of vertices
- 2) for each vertex a list of out-neighbors

An example directed graph (with labeled vertices and arcs):



Adjacency array:

	A	B	C	D	E
A	—	<i>orange</i>	—	—	—
B	—	—	<i>black</i>	<i>green</i>	<i>blue</i>
C	—	—	—	—	—
D	—	—	<i>yellow</i>	—	—
E	<i>white</i>	<i>red</i>	—	—	—

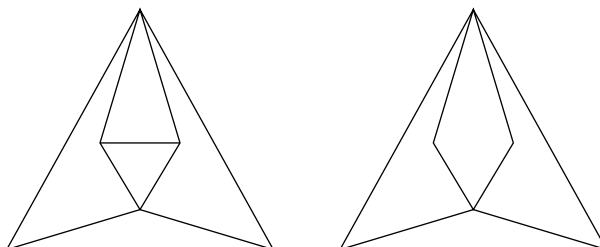
Adjacency list:

A	<i>orange, B</i>		
B	<i>black, C</i>	<i>green, D</i>	<i>blue, E</i>
C			
D	<i>yellow, C</i>		
E	<i>red, B</i>	<i>white, A</i>	

The advantage of the adjacency matrix is that determining $\text{isAdjacent}(u,v)$ is $O(1)$. The disadvantage of adjacency matrix is that it can be space-inefficient, and enumerating outNeighbors etc. can be slow in sparse graphs.

21.3 Aside

Practice. Draw each of the following without lifting your pen.



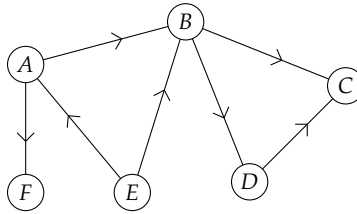
21.4 Topological Sort

A **DAG**, directed acyclic graph, is a directed graph without directed cycles. The classic example is scheduling constraints between tasks of a project.

A **topological ordering** is an ordering of the vertices such that every arc goes from lower number to higher number vertex

Fact: there is a topological ordering iff directed graph is acyclic

EXAMPLE. In the following DAG, one topological ordering is: E A F B D C



Here is an algorithm for topological sort. A **source** is a vertex with no in-arcs.

Algorithm: TopologicalOrdering()
Repeatedly
Find source, output and remove

For efficiency:

1. maintain counter **indegree** at each vertex **v**, and decrement every time the current source has arc to **v** (no deletions).
2. every time a decrement creates a source, add to container of sources.

21.5 Sample Code

Here is an abstract base class **Dag**, an implementation of topological sort for that class, an adjacency-list implementation of the class, and a primitive test program. (The class **IntQueue** provides a queue of ints.)

```
// Dag.h - wdg - 2008
#ifndef DAG_H
#define DAG_H

//DAG is Directed Acyclic Graph
#include<string>
using namespace std;
```

```

class DAG
{
    public:
        virtual void addEdge (int u, int v) = 0;
        virtual void removeEdge(int u, int v) = 0;
        virtual bool isAdjacent (int u, int v) const = 0;
        virtual int outDegree(int u) const = 0;
        virtual int inDegree(int u) const = 0;
        virtual int numberEdges() const = 0;
        virtual int numberVertices() const = 0;
        virtual string toString() const = 0;
};

#endif

```

```

// GraphAlgorithms.cpp - wdg - 2008
// Using Java style!
#include "Dag.h"
#include "IntQueue.h"

#include <iostream>
using namespace std;

class GraphAlgorithms
{
    public:
        static string topologicalSort(const DAG & dag)
        {
            int N = dag.numberVertices();
            int inDeg[N];
            for(int i=0; i<N; i++)
                inDeg[i] = dag.inDegree(i);

            IntQueue Q(N);
            for(int j=0; j<N; j++)
                if( inDeg[j]==0 )
                    Q.enqueue( j );

            string S;
            while ( !Q.isEmpty() ) {

```

```

        int curr = Q.dequeue();
        S += curr+'0';
        S += " ";
        for(int j=0; j<N; j++)
            if( dag.isAdjacent(curr,j) ) {
                inDeg[j]--;
                if( inDeg[j]==0 )
                    Q.enqueue(j);
            }
    }

    return S;
}
};

```

```

// AListDAG.h - wdg - 2008
// Adjacency list implementation of directed graph
#include "Dag.h"
#include <cstdlib>
using namespace std;

class GNode
{
    int neighbor;
    GNode *next;
    GNode( int nb, GNode *nx ) : neighbor(nb), next(nx) {
    }
    friend class AListDAG;
};

class AListDAG : public DAG
{
private:
    int N;
    GNode **outNeigh;    // used as array of pointers

public:
    // Constructor
    AListDAG(int vertices) : N(vertices), outNeigh(new GNode *[vertices])

```

```

{
    for(int i=0; i<N; i++)
        outNeigh[i]=NULL;
}

virtual ~AListDAG( )
{
    for(int i=0; i<N; i++)
        while( outNeigh[i] ) {
            GNode * hold = outNeigh[i]->next;
            delete outNeigh[i];
            outNeigh[i]=hold;
        }
    delete [] outNeigh;
}

void addEdge(int u, int v)
{
    outNeigh[u] = new GNode(v, outNeigh[u]);
}

void removeEdge(int u, int v)
{
    if( outNeigh[u] && v==outNeigh[u]->neighbor )
        outNeigh[u] = outNeigh[u]->next;
    else {
        GNode *curr = outNeigh[u];
        while ( curr->next && v!=curr->next->neighbor )
            curr = curr->next;
        if( curr->next && v==curr->next->neighbor )
            curr->next = curr->next->next;
    }
}

int numberVertices( ) const
{
    return N;
}

```

```

int inDegree(int v) const
{
    int count = 0 ;
    for(int i=0; i<N; i++)
        for( GNode *curr = outNeigh[i]; curr; curr=curr->next )
            if( v==curr->neighbor )
                count++;
    return count;
}

```

```

int outDegree(int v) const
{
    int count = 0 ;
    for( GNode *curr = outNeigh[v]; curr; curr=curr->next )
        count++;
    return count;
}

```

```

bool isAdjacent (int u, int v) const
{
    for( GNode *curr = outNeigh[u]; curr; curr=curr->next )
        if( v==curr->neighbor )
            return true;
    return false;
}

```

```

int numberEdges ( ) const
{
    int count = 0;
    for( int i=0; i<N; i++)
        count += outDegree(i);
    return count;
}

```

```

string toString() const
{
    string S;
    for(int i=0; i<N; i++) {
        S += i+'0';
    }
}

```

```

        S += ": ";
        for( GNode *curr = outNeigh[i]; curr; curr=curr->next) {
            S += curr->neighbor+'0';
            S += "," ;
        }
        S+= "\n";
    }
    return S;
}

};

```

```

// SortTest.cpp - wdg - 2008
// one exercising of topological sort
#include "AListDAG.cpp"
#include "GraphAlgorithms.cpp"
#include<iostream>
using namespace std;

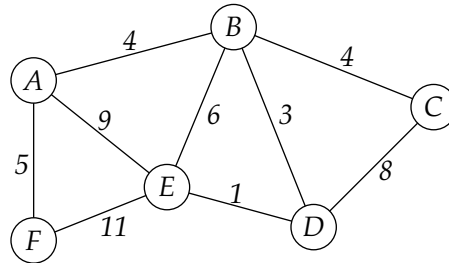
int main( )
{
    AListDAG d(10);
    d.addEdge(4,0); d.addEdge(1,8);
    d.addEdge(4,1); d.addEdge(4,9);
    d.addEdge(2,8); d.addEdge(6,7);
    d.addEdge(6,4); d.addEdge(5,4);
    d.addEdge(3,4); d.addEdge(1,0);
    d.addEdge(0,8); d.addEdge(7,8);
    d.addEdge(4,8); d.addEdge(9,1);
    d.addEdge(9,2); d.addEdge(5,9);
    d.addEdge(3,6); d.addEdge(1,2);
    cout << d.toString() << endl;
    cout << GraphAlgorithms::topologicalSort(d) << endl;
    cout << "Should be: 3 5 6 4 7 9 1 0 2 8" << endl;
    return 0;
}

```

Paths & Searches

22.1 Distance

The *distance* between two vertices is the minimum number of arcs/edges on path between them. In a weighted graph, the *weight* of a path is the sum of weights of arcs/edges. The distance between two vertices is the minimum weight of a path between them.



In the example, distance A–E is 8 (via B and D)

22.2 Breadth-first Search

The idea is to *Visit source; then all its neighbors; then all their neighbors; and so on.* Thus, vertices are visited in order of their *distance* from the start. If the graph is a tree, this is *level ordering*.

```

Algorithm: BFS (start):
  enqueue start
  while queue not empty {
    v = dequeue
    for all out-neighbors w of v
      if ( w not visited ) {
        visit w
        enqueue w
      }
  }

```

22.3 Dijkstra's Algorithm

Dijkstra's algorithm determines the distance from the start vertex to all other vertices. The idea is to *Determine distances in increasing distance from the start*. For each vertex, maintain *dist* giving minimum weight of path to it found so far. Each iteration, choose vertex of minimum *dist*, finalize it and update *dist*.

```

Algorithm: dijkstraOutline (start):
  initialise dist for each vertex
  while some vertex un-finalized {
    v = un-finalized with minimum dist
    finalize v
    for all out-neighbors w of v
      dist(w) = min(dist(w), dist(v) + cost v-w)
  }
  
```

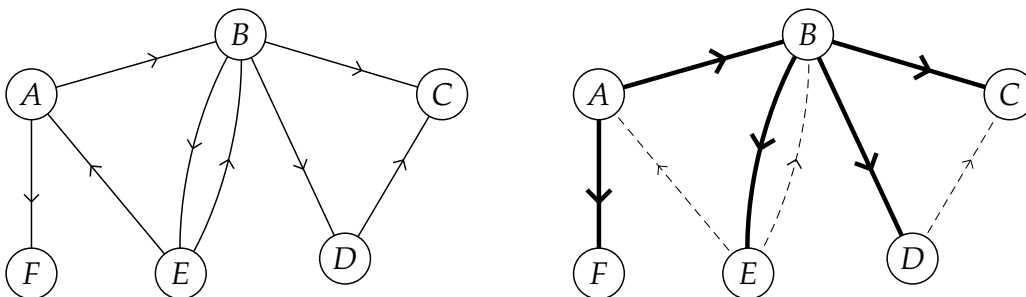
Comments:

- why works? Exercise.
- implementation, store boolean known
- to get path, store Vertex prev
- running time: naïve is $O(n^2)$
- to speed up: use priority queue that supports decreaseKey

22.4 Depth-First Search

The idea is *labyrinth wandering*: keep exploring new vertex from current vertex; when get stuck, backtrack to most recent vertex with unexplored neighbors. The edges/arcs used to discover new vertices are called *tree edges*.

EXAMPLE



Digraph and DFS tree from A

```
Algorithm: DFS(v):
  for all edges e outgoing from v
    w = other end of e
    if w unvisited then {
      label e as tree-edge
      recursively call DFS(w)
    }
```

Notes on DFS

- the search visits all vertices that are reachable
- fastest if uses adjacency list
- to keep track of whether visited a vertex, must add field to vertex (the decorator pattern)

22.5 Test for Strong Connectivity

Algorithm: 1. Do a DFS from arbitrary vertex v & check if all vertices
2. Reverse all arcs and repeat

Why does this work? Think of vertex v as the hub.