Summary of C++ Data Structures

Wayne Goddard
School of Computing, Clemson University, 2018

Part 0: Review

Basics of C++

1.1 Summary

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

C++ has for, while and do for loops, if and switch for conditionals. The standard output is accessed by cout. The standard input is accessed by cin. These require inclusion of iostream library. The language is case-sensitive.

1.2 Data Types

C++ has several data types that can be used to store integers; we will mainly use int. We will use char for characters. Note that, one can treat a char as an integer for arithmetic: for example

```
char myChar = 'D';
int pos = myChar - 'A' + 1;
cout << "the char " << myChar << " is in position " << pos
     << " in the alphabet" << endl;
```

These integer-types also come in unsigned versions. We will not use these much. But do note that arithmetic with unsigned data types is different. For example the code

```
for( unsigned int X=10; X>=0; X--) cout << X;
```

is an infinite loop, since decrementing 0 produces a large number.

C++ has several data types that can be used to store floating-point numbers; we will almost always use double. There is also bool for boolean (that is, true or false); sometimes integers are substituted, where 0 means false and anything non-zero means true.

1.3 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.

1.4 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 at the start of a program tells the compiler of the existence of such a function: it specifies the name, arguments, and type of the function. The actual names of the arguments are optional, but recommended.

1.5 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. To indicate that a pointer points to nothing, it is set equal to nullptr.

1.6 Strings

There are two options to store strings in C++. The first is the way done in C, now called C-strings. A C-string is stored as a sequence of chars, terminated by the null character (which is denoted `'\0'` and has value 0 as an int). The user must ensure 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, cin-ed and cout-ed, 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.

We will mostly use the object from the string class provided in the string library. These can be compared, cin-ed and cout-ed, assigned C-string, appended, etc.

Sample Code

The first example code prints out the prime numbers less than 100. We will explain the use of namespace's later.

In the second example code, the binarySearch function searches a ***sorted*** array for a specific value. It returns the index if it finds the value, and `-1` otherwise.

primality.cpp
BinarySearch.cpp

Basics of Classes

2.1 Objects

An ***object*** is a particular instance of a ***class*** and there may be multiple instances of a class. An object has

- data, called attributes, fields or ***data members***, and
- functions, called methods or ***member functions***.

A member function is called with the `.` notation. For example:

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

The code

```
MyString puppet;
```

during compilation declares variable `puppet` to be an object of class `MyString`, and during execution causes the program to create the variable `puppet` by reserving memory for it and initializing it by executing the no-argument constructor. (See constructors below.)

2.2 Data Members and Member Functions

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.

Member functions execute on an object of that class. Most classes have two types of member functions:

- ***accessor*** functions: these allow the user to ***get*** data from the object.
- ***mutator*** functions: these allow the user to ***set*** data in the object.

2.3 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 no-argument constructor if none is coded. Some constructor is always executed when an object is created.

2.4 Why 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 (which allows partial code reuse)
- ***Polymorphism***: there are multiple implementations of methods and the correct one is executed

Sample Code

Below is a sample class and a main function. But note that there are several style problems with it, some of which we will fix later. The output is

```
GI Joe can drink
Barbie can't drink
```

Citizen.cpp

Program Development

3.1 Testing

One needs to test extensively. Start by trying some standard simple data. 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.

3.2 Avoiding Problems

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. However, the programmer should try to avoid exceptions: consider error recovery and avoidance.

3.3 Literate Programming

Good programming requires extensive comments and documentation. At 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.

3.4 Algorithms

An algorithm for a problem is a recipe that:
(a) is correct,
(b) is concrete,
(c) is unambiguous,
(d) has a finite description, and
(e) terminates.

Having found an algorithm, one should look for an efficient algorithm. As Shaffer writes: "First tune the algorithm, then tune the code."

Summary of C++ Data Structures

Wayne Goddard
School of Computing, Clemson University, 2018

Part 1: Fundamentals

More about Classes, Files and I/O

4.1 Scope, Lifetime and Static

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 be using the ***scope resolution*** operator: prefixing it with the classname followed by ::. Note that global ***static*** variables are created before main is started.

4.2 Object Storage, Allocation and Destructors

Some objects are created and destroyed ***automatically***: local variables in a function or block, member variables in a object, and temporary variables during expression evaluation.

Some objects are created ***dynamically***: this uses the new command and the result is usually assigned to a pointer. The user must deallocate these and release them to the system to avoid memory leaks. The delete command takes a pointer and recycles what the pointer points to. Note that an array that is new'ed needs to be deallocated with delete[] .

A class should often have a ***destructor***; this has the name of the class preceded by a tilde, and is called to properly release memory. The destuctor method is not usually invoked by name, but is automatically called by the delete command. Note that, like constructors, if you do not provide code for a destructor the compile will create a primitive one.

For example:

```
Tiger T;   // user should not run delete on this
Unicorn *U = new Unicorn( ); // dynamic allocation
Vole *V = new Vole[10]; // create array of Voles; default constr. run on each
...
delete U; // invokes destructor and releases space
delete[] V;  // invokes destructors on each Vole and space released
```

4.3 Passing an Object to a Function: 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 in C++. For example, we saw that arrays are implicitly passed this way.

C++ introduced the idea of references or aliases. This allows a method direct access to an object ("sharing") but uses a different syntax. An ampersand `&` indicates an object passed by sharing; inside the function it is treated as if it were a local variable.

4.4 Returning an Object from a Function

In C++, a function can return an object or a pointer to it. It is also possible to return a reference to an object. However, note that an object created with a declaration 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: see the code for the `<<` operator in the next chapter.

4.5 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, with the name 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 practice 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
```

4.6 Const's

The const modifier has several different uses in C++. One can indicate that a variable does not change by putting a const before it:

```
const double myPi = 3.14;
```

One can indicate that an argument is not changed by a function, by putting a const before it:

```
void myMethod( const Foo & bar ) { ... }
```

One can indicate that the method does not change the object on which it is invoked by putting a const after the parameter list:

```
void myMethod( Foo & bar ) const { ... };
```

One can of course use both. The compiler will try to check that the claims are correct. But it cannot guarantee them, since one can create a pointer and get it to point to a variable.

4.7 Initializer Lists

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')   // no-argument constructor
      { }
   private:
      Iso Bar;
      char ging;
 };
```

4.8 Libraries

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

A ***namespace*** is a context for a set of identifiers. By writing `std::string`, we say to use the `string` from that namespace. A way to avoid writing the namespace every time is to use the using expression. But, the user can create a class with the same name as one in `std`, and then the compiler doesn't know which to use.

4.9 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.

4.10 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 sample 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();
}
```

Sample Code

Consider a revision to our Citizen class. This is compiled by

```
g++ CitizenToo.cpp TestCitizenToo.cpp
```

CitizenToo.cpp
CitizenToo.h
TestCitizenToo.cpp

Standard Class Methods

5.1 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 for a member function that returns a fraction that is the sum of the current fraction and another fraction:

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

If the user has some code where two fractions are added, e.g. A+B, then this member function is called on A, with B as the argument. That is, the compiler changes A+B to A.operator+(B) ; In the actual code for the function, the data members of the first fraction are accessed directly; those of the second are accessed with other. notation.

5.2 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)

5.3 Inputting or Outputting a Class

Output of a class can be achieved by overloading the stream insertion operator <<. This is usually a separate ***global function*** (that is, not a member function). In order to access the private variables of your class, you usually need to make it a ***friend*** of your class (by adding its prototype inside the class).

```
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 the operator works with successive <<).

One can use the same approach to read an object from the user. Usually the user data is read into a string and then parsed internally. This is to handle malformed data without crashing.

5.4 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. This provides a ***shallow copy***—only the declared variables are copied. For example, if the class contains the header pointer to a linked list, the pointer will be copied, but both the header in the new object and the old object will point to the same Node in memory. This is usually not what you want. Instead a ***deep copy*** produces a *completely separate object*.

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

```
      barPtr = new Bar( *(other.barPtr) );
    }
};
```

You should assume the deep copy is required unless otherwise specified.

Note that the code `A=B` uses the assignment operator. There is a fundamental trio:

either the default copy constuctor, assignment operator and destructor are all okay, or you need to provide all three.

We see how to create an assignment operator next.

5.5 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, only one object exists.

If we write:

```
Foo bar1, bar2;
// changes to the two objects
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 a head pointer to a linked list, the pointer will be copied, but both the head 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 object
   return *this;
}
```

The this pointer always refers to the object on which the member function is being invoked. (The function has to return the object, so that A=B=C works.) Now, one should first deallocate the old stuff in the object using the delete command. However, if the user writes bar=bar, assigning an object to itself, this can cause a problem. Thus, one adds a test to avoid any changes occurring in this case:

```
if( this != &other ) {
```

Sample Code

We create a class called Fraction. Note that the fraction is stored in simplest form.

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.

Fraction.h
Fraction.cpp
TestFraction.cpp

Algorithmic Analysis

6.1 Algorithm Analysis

The goal of algorithmic 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 time proportional to 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 ***simple*** to count but mirrors the true running time.

6.2 Order Notation

We define big-O:

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

So $5n$ is $O(n^2)$ but n^2 is not $O(5n)$. Note that constants do not matter; saying f is $O(\sqrt{n})$ is the same thing as saying f is $O(\sqrt{22n})$.

The ***order*** (or growth rate) of a function is the simplest smallest function that it is O of. It ignores coefficients and everything except the dominant term.

> E*xample. Some would say* $f(n) = 2n^2 + 3n + 1$ *is* $O(n^3)$ *and* $O(n^2)$*. But its order is* n^2*.*

Terminology: The notation $O(1)$ means constant-time. ***Linear*** means proportional to n. Quadratic means $O(n^2)$. ***Sublinear*** means that the ratio $f(n)/n$ tends to 0 as $n \to \infty$ (somtimes written $o(n)$).

> L*ong Arithmetic. Long addition of two n-digit numbers is linear. Long multiplication of two n-digit numbers is quadratic.*

(Check!)

6.3 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*

6.4 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 the ***number of bits***. The function $\log n$ grows forever, but it grows (much) slower than any power of n.

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

6.5 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 algorithm is*
>
> ```
> for(int y=2; y<N; y++)
>       if( N%y==0 )
>           return false;
>   return true;
> ```
>
> *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?)*

Note that array access is assumed to take constant time.

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 integers between them; etc. Here is a test of this: this method is* ***quadratic****.*

```
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;
```

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.

7.1 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 ;
  }
}
```

7.2 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;
}
```

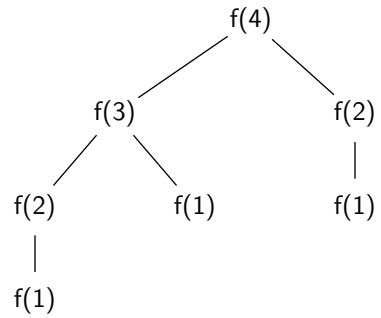
It is not hard to see that, for example, g(3) prints out the numbers from 3 down to 1. But, you have to be a bit more careful. The recursive call occurs before the value 3 is printed out. This means that the output is from smallest to biggest.

```
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 (((((.

7.3 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).)

7.4 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 placeQueen(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
          placeQueen(row+1);
          // if reach here, have failed and need to backtrack
```

```
        erase columns and diagonals this queen attacks;
      }
    }
  }
```

7.5 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.

Sample Code

StepsByRecursion.cpp

Summary of C++ Data Structures

Wayne Goddard
School of Computing, Clemson University, 2018

Part 2: Linear Data Types

Collections, Data Structures, Bags, Sets, and Lists

8.1 ADT

An ADT or ***abstract data type*** defines a way of interacting with data: it specifies only how the ADT can be used and says nothing about the implementation of the structure. An ADT is conceptually more abstract than a Java interface specification or C++ list of class member function prototypes, and should be expressed in some formal language (such as mathematics).

A ***data structure*** is a way of storing data that implements certain operations. When choosing a data structure for your ADT, you might consider many issues such as whether the data is static or dynamic, whether the deletion operation is important, and whether the data is ordered. In general

> *A data structure should take ownership of its data.*

In particular, it is responsible for recycling the storage of the data.

8.2 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.

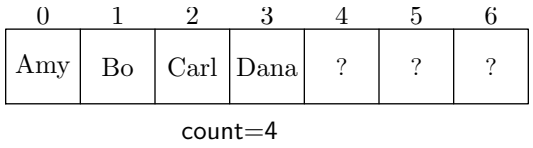
3. A collection where there is an ordering is often called a ***list***. Specific examples include an ***array***, a ***vector*** and a ***sequence***. These have the same idea, but vary as to the methods they provide and the efficiency of those methods.

The Bag ADT might have:

- accessors 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.



Sample Code

An array-based implementation of a set of strings.

StringSet.h
StringSet.cpp
TestStringSet.cpp

Linked Lists

9.1 Links and Pointers

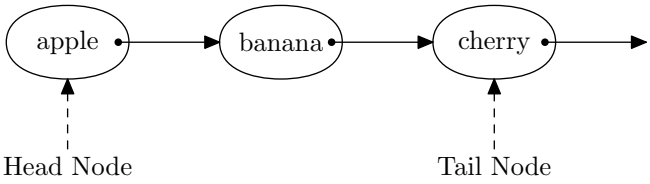
The linked list is not an ADT in its own right; rather it is a way of implementing many data structures. It is designed to *replace* an array.

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 here a struct. Note that in C++ a struct is identical to a class except that its members are public by default.

```
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:

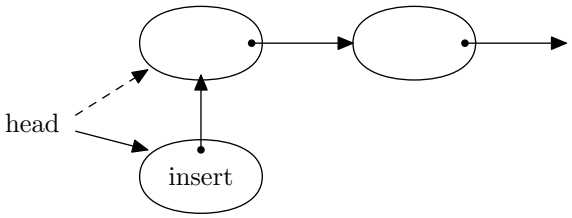
```
Node *head;
```

9.2 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. nullptr 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!=nullptr; 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.

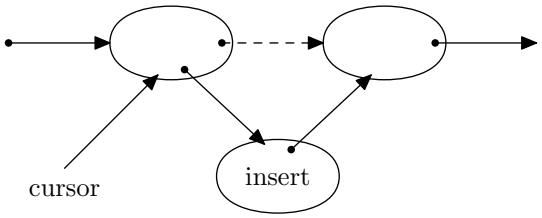


In code:

```
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 need 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;
```

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

9.3 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.

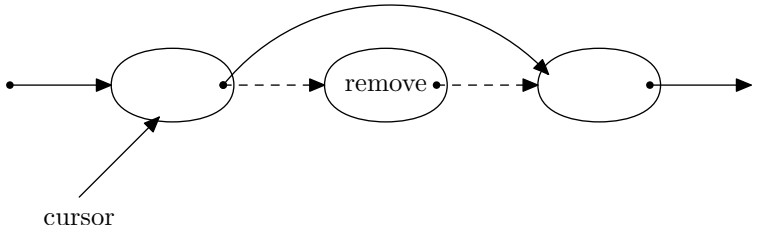
2. Draw a diagram: you have to get the picture right, and you have to get the order right.

9.4 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;
```

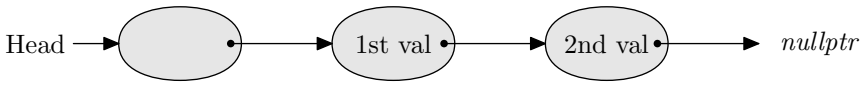
The problem is to organize cursor to be in the correct place. In theory, one would like to traverse the list, find the node to be deleted, and then back up one: but that's not possible. Instead, one has to look one node ahead. And then beware nullptr pointers. See sample code.

9.5 And Beyond

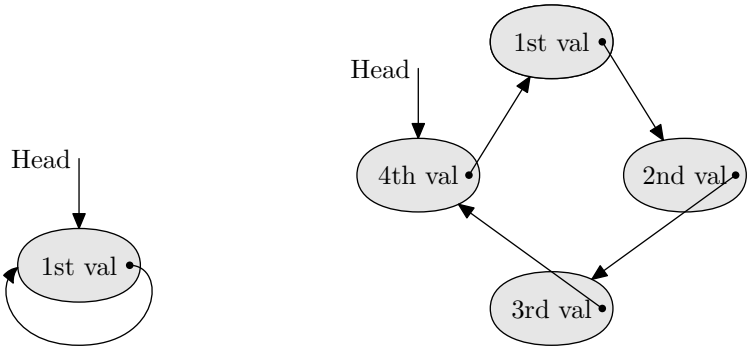
Arrays are better at ***random access***: they can provide an element, given a position, in constant time. Linked lists are better at additions/removals at the cursor: done in constant time. Resizing arrays can be inefficient (but is "on average" constant time).

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.



One can also have ***circularly linked lists*** where the last node points to the first.



Sample Code

MyLinkedBag.h
MyLinkedBag.cpp

Stacks and Queues

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

10.1 Stacks Basics

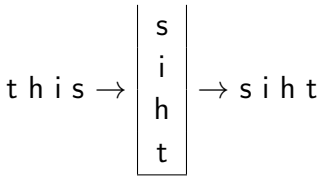
A ***stack*** is a data structure of ordered items such that items can be inserted and removed only at one end (called the ***top***). It is also called a ***LIFO*** structure: last-in, first-out.

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.



10.2 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.

10.3 Application: Balanced Brackets

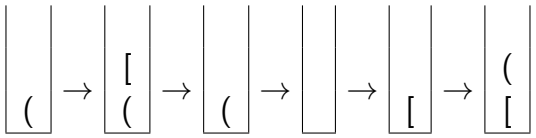
A common application of stacks is the parsing and evaluation of arithmetic expressions. Indeed, compilers use a stack in ***parsing*** (checking the syntax of) programs.

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.

10.4 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

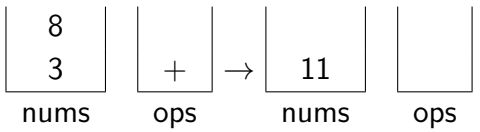
`(((3+8)-5)*(8/4))` → `((11-5)*(8/4))` → `(6*(8/4))` → `(6*2)` → `12`

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

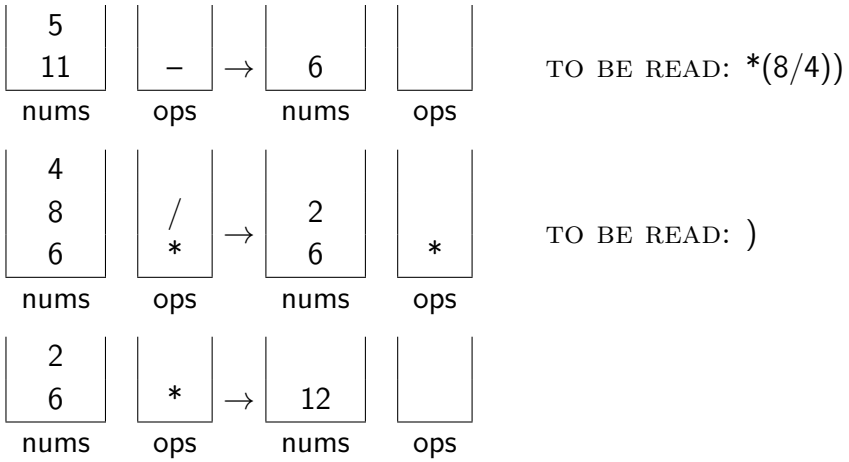
Algorithm (assuming brackets are correct!) is as follows:

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

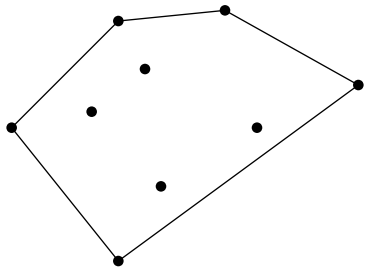


TO BE READ: –5)*(8/4))



10.5 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 elastic 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.



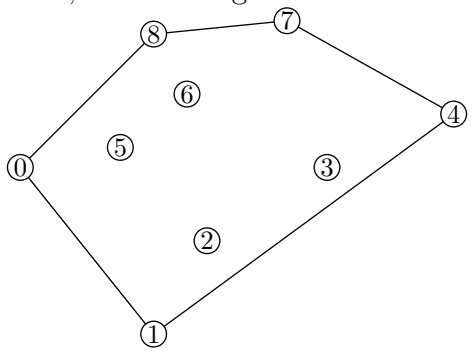
One algorithm to compute the convex hull is Graham's scan. It is an application of a stack. Let 0 be the leftmost point (which is guaranteed to be in the convex hull). Then number the remaining points by angle from 0 going counterclockwise: $1, 2, \ldots, n-1$. Let n^{th} be 0 again.

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 $O(n \log n)$.

For the points given earlier, the labeling is as follows:



The algorithm proceeds:

push(0)
push(1)
push(2)
pop(2), push(3)
pop(3), push (4)
push(5)
pop(5), push(6)
pop(6), push(7)
push(8)
push(0)

10.6 Queue Basics

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.

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.

10.7 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 rear≤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.

10.8 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?).

10.9 Application: Discrete Event 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, is to look ahead whenever possible. The overall mechanism is an infinite loop:

```
while(simulation continuing) do {
  dequeue nextEvent;
  update status;
  collect statistics
  precompute associated nextEvent(s) and add to queue;
}
```

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 allows us to discard the Customer: the only pertinent information is that the Washer is busy.

Sample Code

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

ArrayStack.h
ArrayStack.cpp
brackets.cpp

Standard Template Library

11.1 Overview

The standard template library (STL) provides templates for data structures and algorithms. Each data structure is in its own file. For example, there is vector, stack, and set. These are implemented as templates (which we will discuss more later). For now, it suffices to know that things like vector and stack are created to store a specific data type. This data type is specified in angle brackets at declaration:

```
vector<int> A;
```

Thereafter we can just treat A as before. For example, the push_back method adds an item at the end of the vector. Another useful method is emplace(val); this adds an object to the structure treating val as the input to the constructor for that object.

11.2 Iterators and Range-for Loops

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 (which we skip), but 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 Standard Template Library (STL), 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 it can be 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. Or even replace it with auto: this "typename" can be used in places to help the reader where the compiler can infer the type. 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;
}
```

A ***range-for*** loop can be used to process all the entries in some data structure. E.g.

```
int addup ( vector<int> & A ) {
  int sum = 0;
  for(int val : A )
     sum += val;
  return sum;
}
```

11.3 Adding Templates

Certain code needs the data-type to support certain operations. For example, a set needs a way to test for equality. Actually, the set from the STL assumes there is an equivalent to the $<$ command. (Two objects A and B are considered equal if both $A < B$ and $B < A$ are false.)

Note that if you need to create your own method for use in an STL data structure, you should use a two-argument version as friend, not the one-argument method described earlier:

```
class Foo {
   friend bool operator<(Foo & A, Foo & B);
};
bool operator<(Foo & A, Foo &B) { ... }
```

Sample Code

MyInteger.h
TestMyInteger.cpp

Summary of C++ Data Structures

Wayne Goddard
School of Computing, Clemson University, 2018

Part 3: Trees

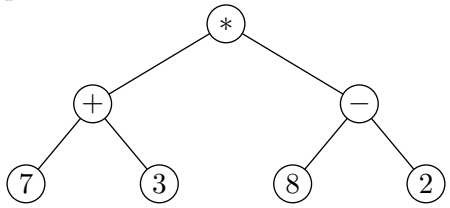
Trees

12.1 Binary Trees

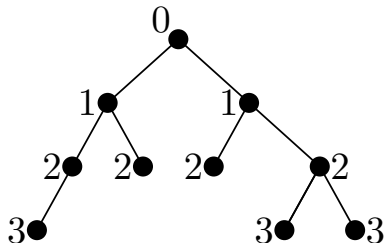
A ***tree*** is a container of positions arranged in child–parent relationship. A tree consists of ***nodes***: we speak of ***parent*** and ***child*** nodes. In a ***binary*** tree, each node has two possible children: a ***left*** and ***right*** child. A ***leaf*** node is one without children; otherwise 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 $(7 + 3) * (8 - 2)$:

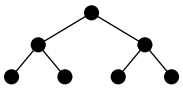


The ***descendants*** of a node are its children, their children etc. A node and its descendants form a ***subtree***. A node u is ***ancestor*** of v if and only if v is descendant of u. The ***depth*** of a node is the number of ancestors (excluding itself); that is, how many steps away from the root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

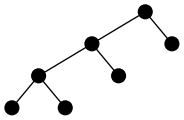


Special trees: A binary tree is ***proper/full*** if every internal node has two children. A binary tree is ***complete*** if it is full and every leaf has the same depth. (NOTE: different books have different definitions.)

Complete



Full but not complete



Note that:

- A complete tree of depth d has 2^d leaves and $2^{d+1} - 1$ nodes in total.
- A full tree has one more leaf than internal node.

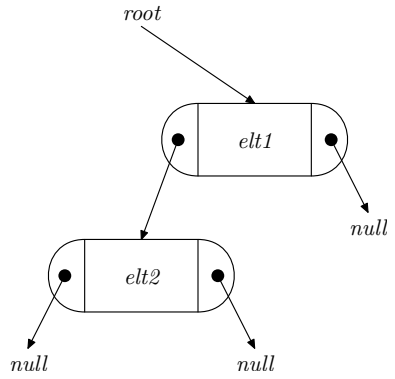
(Exercise to reader: prove these by induction.).

12.2 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 BTNode {
  <type> data;
  BTNode *left;
  BTNode *right;
};
```

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



Methods might include:
- get's and set's (data and children)
- isLeaf
- modification methods: add or remove nodes

For a general tree, there are two standard approaches:

- each node contains a collection of references to children, or
- each node contains references to *firstChild* and *nextSibling*.

12.3 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 unsuccesful 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.

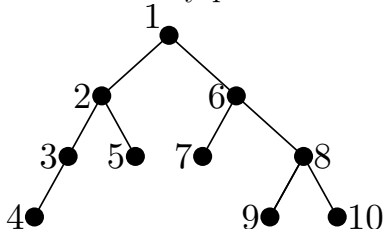
12.4 Tree Traversals

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 )
}
```

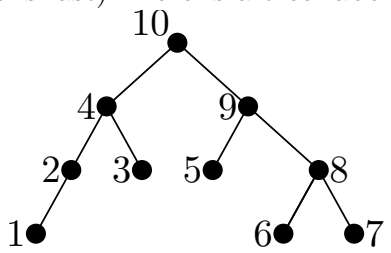
Here is a tree with the nodes labeled by preorder:



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

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

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



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 leftDepth=0, rightDepth=0;
     if( v->left )
        leftDepth = maxDepth (v->left) ;
     if( v->right )
        rightDepth = maxDepth (v->right) ;
     return 1 + max( leftDepth, rightDepth );
  }
}
```

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

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.

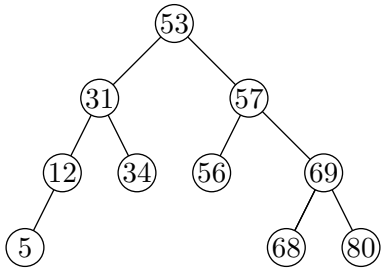
13.1 Binary Search Trees

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

> *left descendants are smaller, right descendants are bigger. (One can adapt this to allow repeated values.)*

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 object are sorted with respect to a ***key***.

Here is an example:



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.*

13.2 Insertion in BST

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

```
Node *find(key x) {
   Node *t=root;
   while( t!=nullptr && 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:

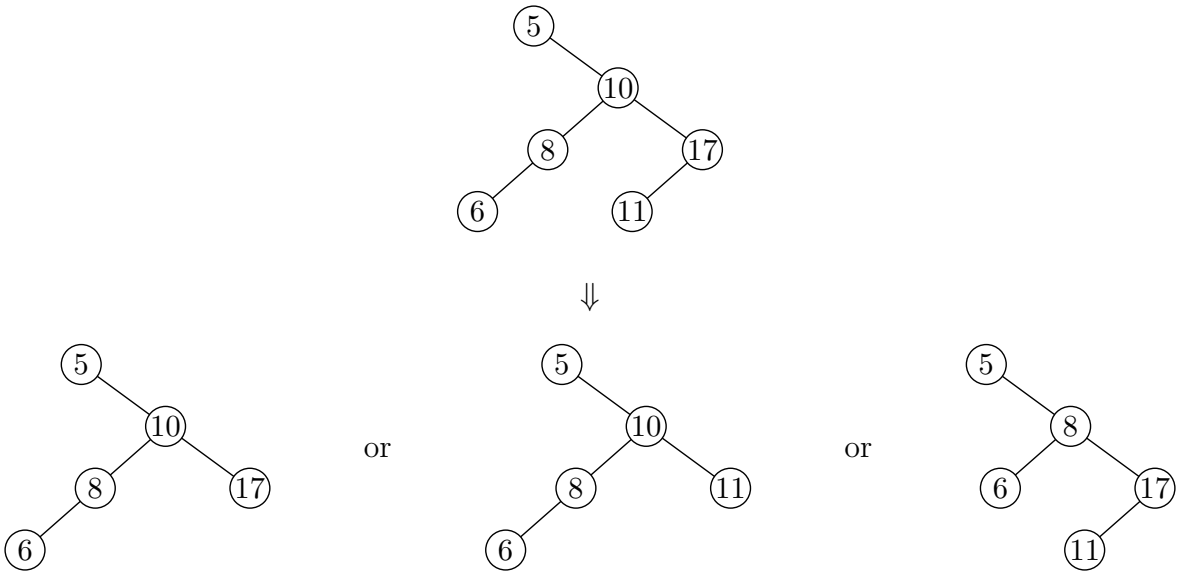
```
Node *insert(ItemType &elem, Node *t) {
   if( t==nullptr )
      return new Node( elem );
   else {
     if( elem.key<t->key )
       t->left = insert(elem,t->left);
     else if( elem.key>t->key )
       t->right = insert(elem,t->right);
     return t;
   }
}
```

13.3 Removal from BST

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

- *Node x is a leaf.* Then just delete.
- *Node x has only one child.* Then delete the node and do "***adoption by grandparent***" (get old parent of x to point to old child of x).
- *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 a right child. So swap the values of nodes x and y, and delete the node y using one of the two 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 $O(n)$.

13.4 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 the size of its subtree (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.)

Sample Code

Here is code for a binary search tree.

BSTNode.h
BinarySearchTree.h
BinarySearchTree.cpp

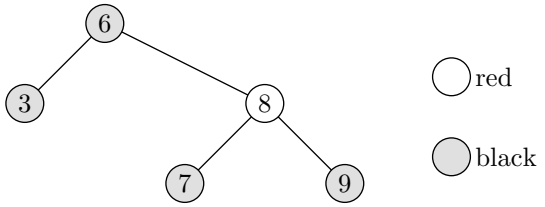
More Search Trees

14.1 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 nullptr contains the same number of black nodes.

Here is an example red-black tree:

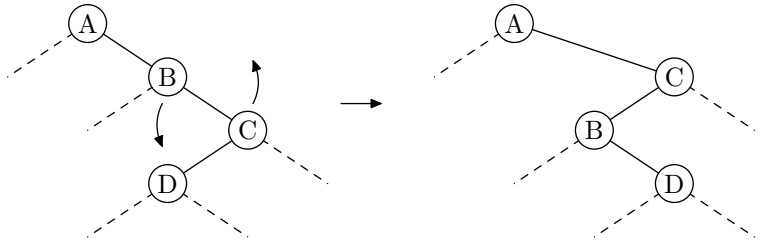


Theorem (proof omitted): The height of a Red-black tree storing n items is at most $2\log(n+1)$

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

14.2 Bottom-Up Insertion in Red-Black Trees

The idea for insertion in a red-black tree is to insert like in a binary search tree and then reestablish the color properties through a sequence of ***recoloring*** and ***rotations***. A rotation can be thought of as taking a parent-child link and swapping the roles. Here is a picture of a rotation of B with C:



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 nullptr 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 swop their colors.

   b) if opposite side to parent, then rotate self with parent, and then proceed as in case a).

We omit the details. Deletion is even more complex. Highlights of the code for red-black tree are included later.

14.3 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.

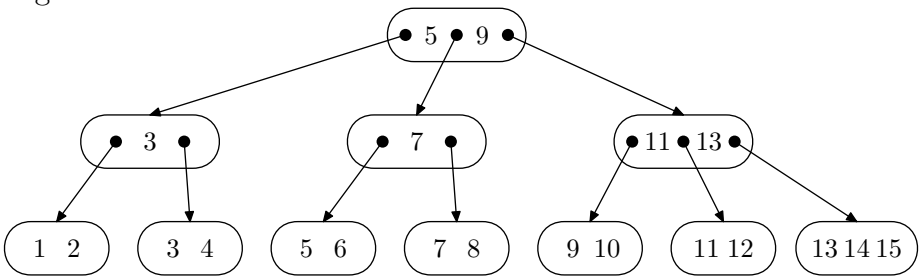
It is straight-forward to search a B-tree. The search moves down the tree. At a node with k keys, the input value is compared with the k keys and based on that, one of the $k+1$ pointers is taken. The time used for a search is proportional to the height of the tree.

14.4 Insertion into B-trees

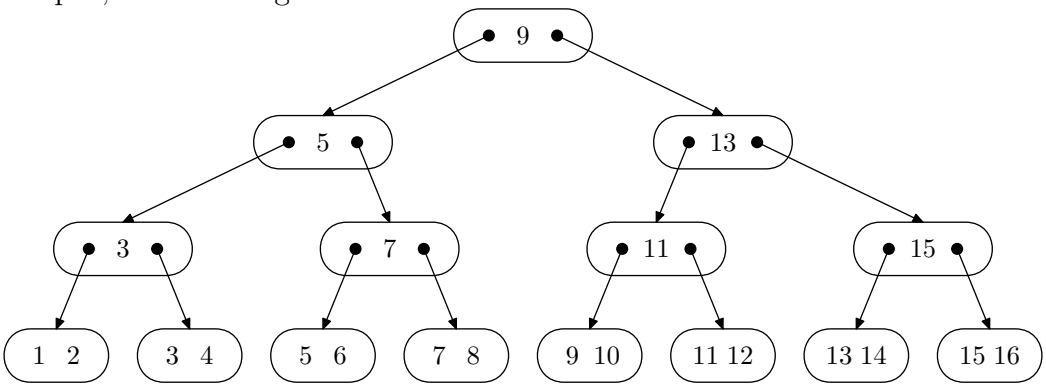
A fundamental operation used in manipulating a B-tree is ***splitting*** an ***overfull*** node. An internal node is overfull if it has M keys; a leaf is overfull if it has $M+1$ values. In the splitting operation, the node is replaced by two nodes, with the smaller and larger halves, and the middle 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.

For example, if we set $M = 3$ and insert the values 1 thru 15 into the tree, we get the following B-tree:



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



Deletion from B-trees is similar but harder. Some code for a B-tree implementation is included in the chapter on inheritance.

Sample Code

Here is code for red-black tree. Note that we have adapted the code for binary search trees given in the previous chapter. An alternative would have been to use inheritance, where RBNode extends BSTNode and RedBlackTree extends BinarySearchTree.

RBNode.h
RedBlackTree.h
RedBlackTree.cpp

Heaps and Priority Queues

15.1 Priority Queue

The (min)-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 is also a version focusing on the maximum.

There are several ***inefficient*** implementations:

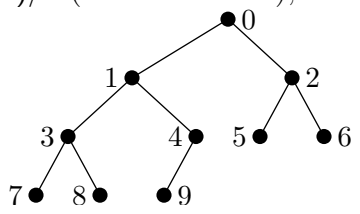
	insert	removeMin
unsorted linked list	$O(1)$	$O(n)$
sorted linked list or array	$O(n)$	$O(1)$
binary search tree	$O(n)$; average $O(\log n)$	

15.2 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 (rounded down), and the root is 0.



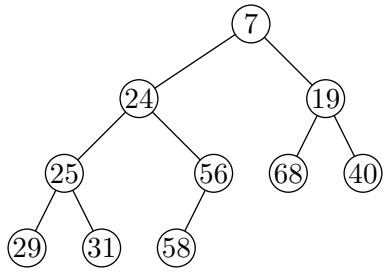
One can store a binary tree in an array/vector by storing each value at the position given by level numbering. But this is wasteful storage, unless nearly balanced.

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 ***min-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. Here is an example:



A ***max-heap*** can be defined similarly.

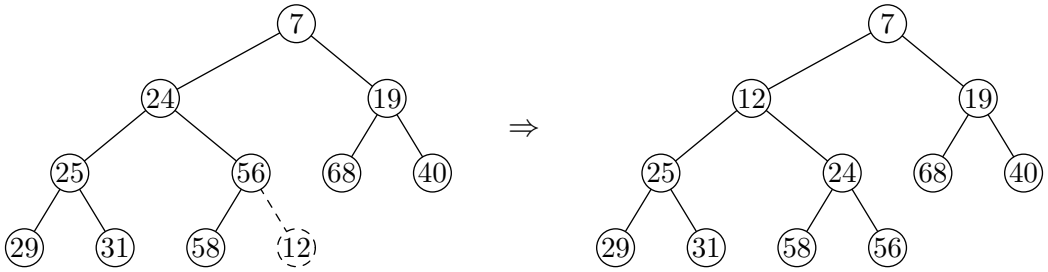
15.3 Min-Heap Operations

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 a "hole" to reduce data movement.

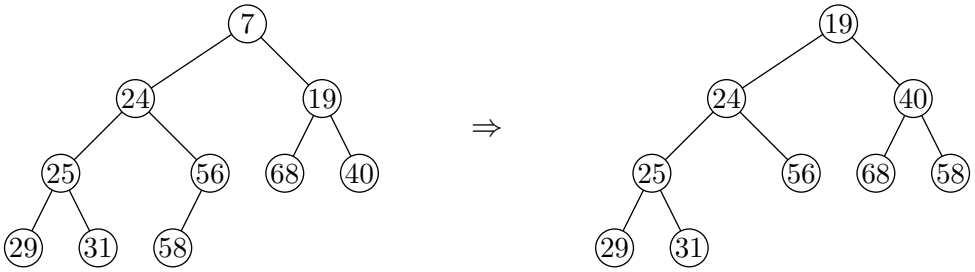
Here is an example of Insertion: inserting value 12:



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:



Variations of heaps include

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

15.4 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 in stored: removeMin moves the minimum to end, and so repeated application produces sorted the 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

15.5 Application: Huffman Coding

The standard binary encoding of a set of C characters takes $\lceil \log_2 C \rceil$ bits for a character. 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:

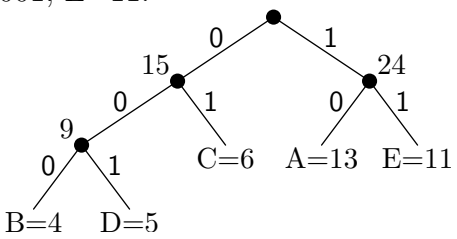
```
Repeat
    merge two (of the) rarest characters into a mega-character
    whose occurrence is the combined
Until only one mega-character left

Assign mega-character the code EmptyString
Repeat
```

split a mega-character into its two parts assigning each of these
the mega-character's 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.

Sample Code

PriorityQ.h
Heap.h
Heap.cpp

Summary of C++ Data Structures

Wayne Goddard
School of Computing, Clemson University, 2018

Part 4: More C++

Inheritance

16.1 Inheritance and Derived Classes

In C++ you can make one class an ***extension*** of another. This is called ***inheritance***. The classes form an ***is-a*** hierarchy. For example, to implement a B-tree, one might want a class for the internal nodes and a class for the leaf nodes, but want pointers to be able to point to either type. One solution is to create a BTreeNode class with two derived classes.

The advantage of inheritance is that it avoids code duplication, promotes code reuse, and improves maintenance and extendibility. For example, one might have general code to handle graphics such as Shape's, with specific code specialized to each graphics component such as Rectangle's.

Inheritance allows one to create multiple ***derived classes*** from a ***base class*** without disturbing the implementation of the base class. Using public inheritance (the only version we study), the class is defined as follows:

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

The derived class automatically has the methods of the base class as member functions, unless declared as private in the base class. They may be declared as protected in the base class to allow direct access only to extensions. Similarly, the derived class can access the instance variables of the base class, provided not private.

16.2 Polymorphism, Static Types, and Dynamic Types

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*** that 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
```

16.3 Overriding Functions

An important facet of inheritance is that the derived class can replace a general function of the base class with a tailored function. ***Overriding*** is providing a function with the same signature as the superclass but different body. The base class labels a function as virtual if it expects that function to be overridden by its derived classes. Note that a function declared as virtual in the base class is automatically virtual in the derived class.

But then a key question is: Which version of the function gets used? If we declare an object directly with code like Rectangle R; then the functions of the Rectangle class are used.

But if we reference an object (that is, use a pointer), then there are two options: if we declare the function as 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 the function as ***nonvirtual***, then which version is determined by the compiler at compile time, based on the static type of the ***reference***.

```
class Shape {
  virtual void foo(){ cout << "base foo"; }
  void bar() { cout << "base bar"; }
};
class Rectangle : public Shape {
  void foo( ) { cout << "derived foo"; }
  void bar( ) { cout << "derived bar";}
};
Shape *X;
Rectangle *Y;
X = Y = new Rectangle();
Y -> foo( );   // prints derived foo
```

```
Y -> bar( );   // prints derived bar
X -> foo( );   // prints derived foo
X -> bar( );   // prints base bar
```

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

16.4 Constructors in Derived Classes

The default constructor for Shape is automatically called before execution of the constructor for Rectangle, unless you specify otherwise. To specify otherwise, put the base class name as the first entry in the initializer list:

```
Derived(int alpha, string beta) : Base(alpha,beta) { }
```

16.5 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 none is 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 when declaring; 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();
```

Sample Code

A (somewhat artificial) example of using inheritance to create a 3-dimensional point given code for a 2-dimensional point.

TwoDPoint.h
TwoDPoint.cpp
ThreeDPoint.h
ThreeDPoint.cpp
TestPoint.cpp

Here is code for a primitive implementation of a B-tree.

BTreeNode.h
BTreeInternal.cpp
BTreeLeaf.cpp
BTree.h
BTree.cpp

Templates & Exceptions

We briefly consider exceptions and templates.

17.1 Exceptions

An ***exception*** is an unexpected event that occurs when the program is running. For example, if new cannot allocate enough space, this causes an exception. An exception is explicitly thrown using a throw statement. A throw statement must specify an exception object to be thrown. There are exceptions already defined; it is also possible to create new ones. (Or one can, for example, throw an int.)

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. 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.

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.

17.2 Templates

Thus far in our code 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. We saw already that if we want a set from the STL 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. The parameter class can be called a typename or a class: the two words are identical in this context. For example, a Node class might be written:

```
template <typename T>    //  <class T> would have same meaning
struct Node
{
      Node(T initData, Node *initNext) :
                  data(initData), next(initNext)
```

```
        { }
        T data;
        Node *next;
  } ;
```

This class is then used in the linked-list class with

```
template <typename U>
class List
{
  Node<U> *head;
```

It is standard to break the class heading over two lines. To a large extent, one can treat 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 instantiated parameter choice separately.*

Consequently, the implementation code must be in the template file: one can `#include` the cpp-file at the end of the header file.

As example, iterators allow one to write generic code. For example, rather than having a built-in boolean contains function, one does:

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

The algorithm library has multiple templates for common tasks in containers.

17.3 Providing External Function for STL

To do sorting, one could assume $<$ is provided. We saw earlier how to create such a function for our own class. But sometimes we are using an existing class such as string.

Some sorting templates allow the user to provide a function to use to compare the elements. For the case of sorting this is sometimes called a comparator. See the code for the Sorting chapter.

In some cases, the template writers provide several options. One option for the user is then to provide a specific function (or functor) within the std namespace. We do not discuss this here.

Sample Code

Note that this code is compiled with g++ TestSimpleList.cpp only. (SimpleList.cpp is #included by its header file.)

SimpleList.h
SimpleList.cpp
TestSimpleList.cpp

Summary of C++ Data Structures

Wayne Goddard
School of Computing, Clemson University, 2018

Part 5: More Data Structures and Algorithms

Hash Tables and Dictionaries

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 by allowing the dictionary to be unsorted.

18.2 Components

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

- an array of fixed size (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 sum

$$\sum_i a_i \, 37^i$$

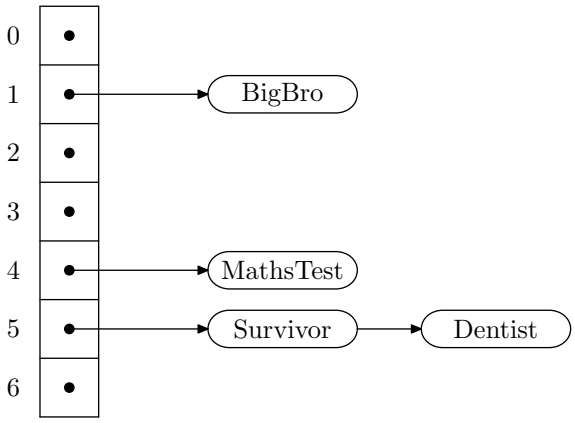
where a_i are the codes for the individual letters.

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 $\rightarrow$ 1
> Survivor = 1921182209221518 $\rightarrow$ 5
> MathsTest = 130120081920051920 $\rightarrow$ 4
> Dentist = 04051420091920 $\rightarrow$ 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 $h, h+1, h+2, h+3, \ldots$
- ***quadratic probing***: move down array in increasing increments: $h, h+1, h+4, h+9, h+16, \ldots$ (again, wrap around if needed)

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***: when the entry in a bucket is deleted, the bucket must be marked as "previously used" rather than "empty". 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.

One can show (a process called ***amortized analysis***) that this does not significantly affect the average running time.

Sorting

We have already seen one sorting algorithm: Heap Sort. This has running time $O(n \log n)$. Below are four more comparison-based sorts; that is, they only compare entries. (An example of an alternative sort is ***radix*** sort of integers, which directly uses the bit pattern of the elements.)

19.1 Insertion Sort

Insertion Sort is the algorithm that:

> *adds elements one at a time, maintaining a sorted list at each stage.*

Say the input is an array. Then the natural implementation is such that the sorted portion is on the left and the yet-to-be-examined elements are on the right.

In the worst case, the running time of Insertion Sort is $O(n^2)$; there are n additions each taking $O(n)$ time. For example, this running time is achieved if the list starts in exactly reverse order. On the other hand, if the list is already sorted, then the sort takes $O(n)$ time. (Why?)

Insertion Sort is an example of an ***in situ*** sort; it does not need extra temporary storage for the data. It is also an example of a ***stable sort***: if there are duplicate values, then these values remain in the same relative order.

19.2 Shell Sort

Shell Sort was invented by D.L. Shell. The general version is:

> *0. Let $h_1, h_2, \ldots, h_k = 1$ be a decreasing sequence of integers.*
> *1. For $i = 1, \ldots, k$: do Insertion Sort on each of the h_i subarrays created by splitting the array into every ${h_i}^{th}$ element.*

Since in phase k we end with a single Insertion Sort, the process is guaranteed to sort.

Why then the earlier phases? Well, in those phases, elements can move farther in one step. Thus, there is a potential speed up. The most natural choice of sequence is $h_i = n/2^i$. On average this choice does well; but it is possible to concoct data where this still takes $O(n^2)$ time. Nevertheless, there are choices of the h_i that guarantee Shell Sort takes better that $O(n^2)$ time.

19.3 Merge Sort

Merge Sort was designed for computers with external tape storage. It is a recursive divide-and-conquer algorithm:

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

The only step that actually does anything is the merging. The question is: how to merge two sorted lists to form one sorted list. The algorithm is:

> *repeatedly: compare the two elements at the tops of both lists, removing the smaller.*

The running time of Merge Sort is $O(n \log n)$. The reason for this is that there are $\log_2 n$ levels of the recursion. At each level, the total work is linear, since the merge takes time proportional to the number of elements.

Note that a disadvantage of Merge Sort is that extra space is needed (this is not an ***in situ*** sort). However, an advantage is that sequential access to the data suffices.

19.4 QuickSort

A famous recursive divide-and-conquer algorithm is QuickSort.

1. *Pick a pivot*
2. *Partition the array into those elements smaller and those elements bigger than the pivot*
3. *Call QuickSort on each piece*

The most obvious method to picking a pivot is just to take the first element. This turns out to be a very bad choice if, for example, the data is already sorted. Ideally one wants a pivot that splits the data into two like-sized pieces. A common method to pick a pivot is called ***middle-of-three***: look at the three elements at the start, middle and end of the array, and use the median value of these three. The "average" running time of QuickSort is $O(n \log n)$. But one can concoct data where QuickSort takes $O(n^2)$ time.

There is a standard implementation. Assume the pivot is in the first position. One creates two "pointers" initialized to the start and end of the array. The pivot is removed to create a hole. The pointers move towards each other, one always pointing to the hole. This is done such that: the elements before the first pointer are smaller than the pivot and the elements after the second are larger than the pivot, while the elements between the pointers have not been examined. When the pointers meet, the hole is refilled with the pivot, and the recursive calls begin.

19.5 Lower Bound for Sorting

Any comparison-based sorting algorithm has running time at least $O(n \log n)$.

Here is the idea behind this lower bound. First we claim that there are essentially $n!$ possible answers to the question: what does the sorted list look like. One way to see this, is that sorting entails determining the rank (1 to n) of every element. And there are $n!$ possibilities for the list of ranks.

Now, 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 guarantee having narrowed down the list to one possibility. (The code can be thought of as a binary decision tree.) A mathematical fact (using Stirling's formula) is that $\log_2(n!)$ is $O(n \log n)$.

Sample Code

Here is template code for Insertion Sort. We also introduce the idea of a ***comparator***, where the user can specify how the elements are to be compared.

Sorting.cpp

Algorithmic Techniques

There are three main algorithmic techniques: Divide and conquer, greedy algorithms, and dynamic programming.

1. *Divide and Conquer.* In this approach, you find a way to divide the problem into pieces such that: if you recursively solve each piece, you can stitch together the solutions to each piece to form the overall solution. Both Merge Sort and QuickSort are classic examples of divide-and-conquer algorithms. Another famous example is modular exponentiation (used in cryptography).

2. *Greedy Algorithms.* In a greedy algorithm, the optimal solution is built up one piece at a time. At each stage the best feasible candidate is chosen as the next piece of the solution. There is no back-tracking. An example of a greedy algorithm is Huffman coding. Another famous example is several algorithms for finding a minimum spanning tree of a graph.

3. *Dynamic Programming.* If you find a way to break the problem into pieces, but the number of pieces seems to explode, then you probably need the technique known as dynamic programming. We do not study this.

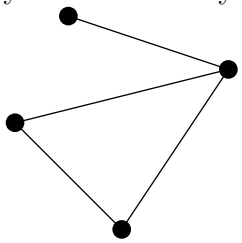
Graphs

21.1 Graphs

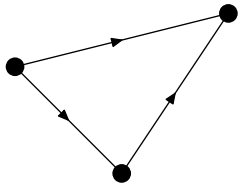
A graph has two parts: ***vertices*** (one vertex) also called ***nodes***. An ***undirected graph*** has undirected ***edges***. Two vertices joined by edge are ***neighbors***. A ***directed graph*** has directed ***edges/arcs***; each arc goes from ***in-neighbor*** to ***out-neighbor***. Examples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



connected



not strongly connected

21.2 Graph Representation

There are two standard approaches to storing a graph:

Adjacency Matrix

1) container of numbered vertices, and

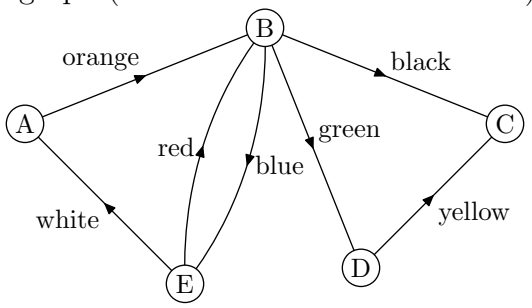
2) array where each entry has info about the corresponding edge.

Adjacency List

1) container of vertices, and

2) for each vertex an unsorted bag of out-neighbors.

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



Adjacency array:

	A	B	C	D	E
A	—	*orange*	—	—	—
B	—	—	*black*	*green*	*blue*
C	—	—	—	—	—
D	—	—	*yellow*	—	—
E	*white*	*red*	—	—	—

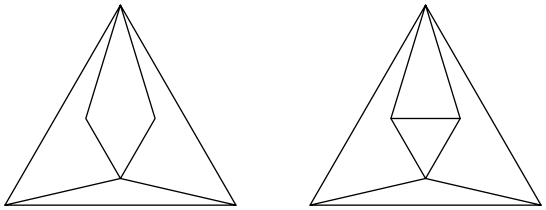
Adjacency list:

A	*orange*, B		
B	*black*, C	*green*, D	*blue*, E
C			
D	*yellow*, C		
E	*red*, B	*white*, A	

The advantage of the adjacency matrix is that determining 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.

21.3 Aside

Practice. Draw each of the following without lifting your pen or going over the same line twice.

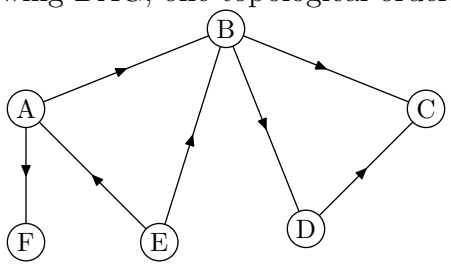


21.4 Topological Sort

A ***DAG***, directed acyclic graph, is a directed graph without directed cycles. The classic application 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.

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



A ***source*** is a vertex with no in-arcs and a ***sink*** is one with no out-arcs.

Theorem:
a) If a directed graph has a cycle, then there is no topological ordering.
b) A DAG has at least one source and one sink.
c) A DAG has a topological ordering.

Consider the proof of (a). If there is a cycle, then we have an insoluble constraint: if, say the cycle is $A \to B \to C \to A$, then that means A must occur before B, B before C, and C before A, which cannot be done.

Consider the proof of (b). We prove the contrapositive. Consider a directed graph without a sink. Then consider walking around the graph. Every time we visit a vertex we can still leave, because it is not a sink. Because the graph is finite, we must eventually revisit a vertex we've been to before. This means that the graph has a cycle. The proof for the existence of a source is similar.

The proof of (c) is given by the algorithm below.

21.5 Algorithm for Topological Ordering

Here is an algorithm for finding a topological ordering:

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

For efficiency, use the Adjacency List representation of the graph. Also:

1. maintain a counter in-degree at each vertex v; this counts the arcs into the vertex from "nondeleted" vertices, and decrement every time the current source has an arc to v (no actual deletions).

2. every time a decrement creates a source, add it to a container of sources.

There is even an efficient way to initially calculate the in-degrees at all vertices simultaneously. (How?)

Sample Code

Here is an abstract base class DAG, an implementation of topological sort for that class, and an adjacency-list implementation of the class

Dag.h
GraphAlgorithms.cpp
AListDAG.h
AListDAG.cpp

Paths & Searches

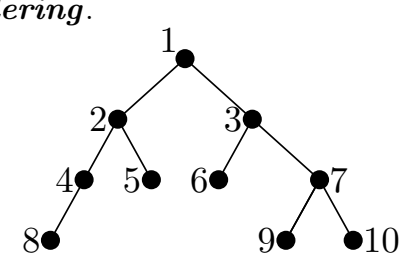
22.1 Breadth-first Search

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, which both run in linear time..

The idea behind ***breadth-first search*** is to:

> *Visit the source; then all its neighbors; then all their neighbors; and so on.*

If the graph is a tree and one starts at the root, then one visits the root, then the root's children, then the nodes at depth 2, and so on. That is, one level at a time. This is sometimes called ***level ordering***.



BFS uses a ***queue***: each time a node is visited, one adds its (not yet visited) out-neighbors to the queue of nodes to be visited. The next node to be visited is extracted from the front of the queue.

```
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
            }
    }
```

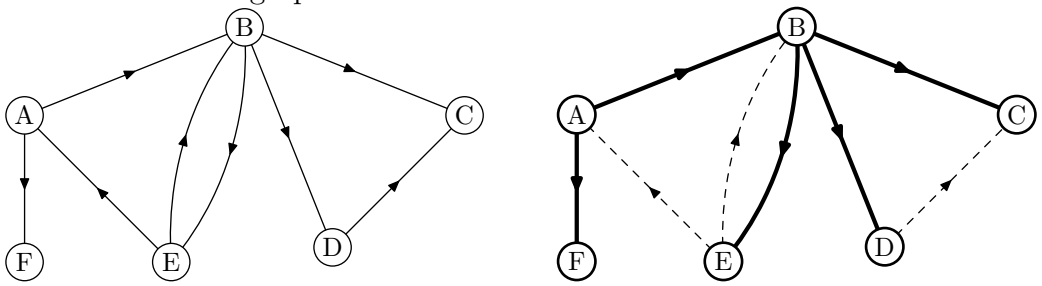
22.2 Depth-First Search

The idea for ***depth-first search*** (DFS) is "labyrinth wandering":

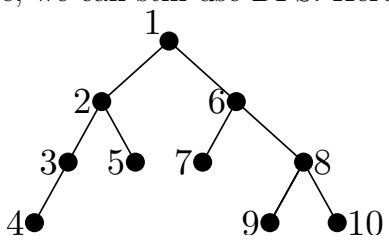
> *keep exploring new vertex from current vertex; when get stuck, backtrack to most recent vertex with unexplored neighbors*

In DFS, the seach continues going deeper into the graph whenever possible. When the search reaches a dead end, it backtracks to the last (visited) node that has unvisited neighbors, and continues searching from there. A DFS uses a ***stack***: each time a node is visited, its unvisited neighbors are pushed onto the stack for later use, while one of its children is explored next. When one reaches a dead end, one pops off the stack. The edges/arcs used to discover new vertices form a tree.

EXAMPLE. Here is graph and a DFS-tree from vertex A:



If the graph is itself a tree, we can still use DFS. Here is an example:




```
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)
        }
```

Note:

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

22.3 Test for Strong Connectivity

Recall that a directed graph is strongly connected if one can get from every vertex to every other vertex. Here is an algorithm to test whether a directed graph is strongly connected or not:

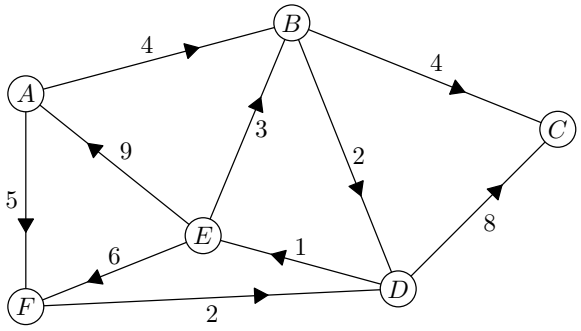
Algorithm: 1. Do a DFS from arbitrary vertex v & check that
all vertices are reached
2. Reverse all arcs and repeat

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

22.4 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. For example, in a BFS in an unweighted graph, vertices are visited in order of their distance from the start.

EXAMPLE. In the example graph below, the distance from A to E is 7 (via vertices B and D):



22.5 Dijkstra's Algorithm

Dijkstra's algorithm determines the distance from a 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 a vertex of minimum dist, finalize it and update all dist values.

```
Algorithm: Dijkstra (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)
    }
```

If doing this by hand, one can set in out in a table. Each round, one circles the smallest value in an unfinalized column, and then updates the values in all other unfinalized columns.

Example. Here are the steps of Dijkstra's algorithm on the graph of the previous page, starting at A.

A	B	C	D	E	F
⓪	∞	∞	∞	∞	∞
	④	∞	∞	∞	5
		8	6	∞	⑤
		8	⑥	∞	
		8		⑦	
		⑧			

Comments:

- Why Dijkstra works? Exercise.
- Implementation: store boolean array known. To get the actual shortest path, store Vertex array prev.
- The running time: simplest implementation gives a running time of $O(n^2)$. To speed up, use a priority queue that supports decreaseKey.