IUSB CSCI

C243 Data Structures

Class Notes

Dr. Vrajitoru

Fall 2014

Table of Contents

0. REVIEW: Pointers, Dynamic Allocation, Linked Lists

Recall that a *pointer variable* is a variable that is intended to hold the memory address in RAM at which some kind of data is stored. When a pointer variable has been given a valid address, then that address is called a *pointer*. We can create a pointer variable with a declaration such as

```
int * p;
```

This creates a variable `p` that's intended to hold a pointer to an integer. The data type of the variable `p` is `int *` which can be read as "pointer to `int`". The declaration above does not place a valid memory address in the variable `p`. We could do that with the following two expressions:

```
int n = 7;
p = &n;
```

The first of these expressions declares an integer variable `n` and initializes it to the value `7`. The second expression takes the RAM address of the variable `n` and places it in the variable `p`. (Recall that the operator `&`, when placed just before a variable, takes the RAM address of that variable.) Now we say that `p` "points to" the contents of the variable `n`. This gives us two ways of accessing the memory location that holds the integer `7`. They are illustrated by the following two expressions, which produce the same output:

```
cout << n << endl;     // output the value of  n
cout << *p << endl;    // output the value pointed to by  p
```

The expression `*p` denotes "the data object pointed to by `p`", which in this case is an integer. We say that the asterisk operator `*` "dereferences" the pointer `p`. Here is another example that illustrates the two ways of accessing the data object `7`: either of the following expressions will change the value stored in `n` from `7` to `100`:

```
n = 100;
*p = 100;
```

We can change the value of `p` (instead of the value it points to) by assigning it a different memory address. For example, if we write

```
int k = 24;
p = &k;
```

then `p` will no longer point to the place in memory where the value of `n` is stored; instead, it will now point to the place where the value of `k` is stored. That is, the value of `p` will now be the memory address of the variable `k`.

0.1. Arrays and Pointers

When you are reading and writing C++ programs, it is important to understand that when an array is declared in an expression such as

```
int data[50];
```

the identifier "data" is a *pointer constant*, and the declaration above creates that constant and makes it point to the first cell in the array of 50 integer cells. (In the declaration above, the cells are not initialized, and so they may contain garbage bits left over from whatever program previously used those 50 memory locations in the RAM.) A pointer variable can be assigned the value of a pointer constant. For example, using the pointer variable `p` declared earlier, we could now write

```
p = data;
```

After this statement is executed, `p` will point to whatever garbage integer value occupies the first cell of the array `data`. We could now assign the value `44` to that cell in any of the following four ways:

```
data[0] = 44;
*p = 44;
p[0] = 44;
*data = 0;
```

If we wished to do so, we could now "advance" the pointer in `p` so that it pointed to the second cell of the array `data`. The simplest way to do that is just to write

```
++p;
```

although it is equally correct to write

```
p = p + 1;
```

At this point, we could place the integer 99 in the second cell of the `data` array in any of the following ways:

```
data[1] = 99;
*p = 99;
p[0] = 99;
*(data + 1) = 99;
```

Incidentally, a different way to make `p` point to the first cell (cell 0) of the array `data` is by writing the more complicated expression

```
p = &(data[0]);
```

but writing an expression such as this marks the programmer as ignorant of the connection between pointers and arrays in C++.

0.2. Dynamic Allocation of Arrays

The example above, which involves making a pointer point to an existing array, is not the most common way for pointers to be used in connection with arrays. Far more often we perform the following steps instead:

- create (declare) a pointer variable,
- *dynamically allocate* a (nameless) array,
- and make the pointer variable point to the array.

For example, we could write the following:

```
float * fdata;            // Declare "fdata" to be a "pointer to
float".
fdata = new float[50]; // Dynamically allocate array of 50 floating
                       // point numbers; make fdata point to array.
```

We could now initialize all 50 cells of this array to the real number −1.0 as follows:

```
for (int i = 0; i < 50; ++i)
   fdata[i] = -1.0;
```

What happens during the "dynamic allocation" of memory space in the example above? The answer is that the library function `new` goes to the *run-time heap*, which is a section of RAM that has been set aside by the operating system for use by your program. (The run-time heap will be part of your program's "memory segment" whenever you execute the program.) The library function searches for some "unallocated memory", in this case, 50 contiguous floating point cells. If it is able to find that much RAM space, it marks those 50 cells as "allocated" (using some "red tape" accounting entries in a couple of cells just before the 50 floating point cells), and then it returns a pointer to the first of the 50 cells. If it is unable to find the requested amount of unallocated memory, then it returns the special pointer value NULL . It is always important to check whether dynamic allocation has succeeded or failed, so the example above involving the `fdata` pointer is not complete. We might decide to make it read this way:

```
float * fdata;
fdata = new float[50]; // Dynamically allocate array of 50 floating
                       // point numbers; make fdata point to array.
if (fdata == NULL)
{
    cout << "Dynamic allocation failure for fdata." << endl;
    exit(1);  // Terminate the program.
}
else .....
```

When the library function `exit()` is called, it takes the drastic action of terminating execution of the program. The argument you give to the `exit()` function is "returned" to the operating system, or to the environment from which the program was run. For now, just use the argument `1`.

Let's look at a situation in which dynamic allocation is quite important. Suppose we are writing a C++ program that needs to obtain a set of integers from an interactive user. The integers will be placed in an array, and then our program will perform one or more operations on the array. Suppose we decide, for purposes of program modularization, to write a separate "data-getting" function. The function will ask the user how many integers she will enter, read her response, and then prompt her for each of the numbers separately. The data-getting function will return a pointer to the storage array and also will return the number of integers that were entered. It will return these values by means of two *reference* parameters:

`data` whose data type will be `int *` (pointer to `int`);
`capacity` whose data type will be `int`.

Here is the code for such a function.

```
void getData (int * &data, int &capacity)
{
    cout << "How many integers do you want to enter? ";
    cin  >> capacity;
    data = new int[capacity];  // Dynamically allocate an array and
                               // make parameter "data" point to it.
    if (data == NULL)
    {
        cout << "Dynamic allocation failure for data array.\n";
        cout << "The program will now be terminated."  << endl;
        exit(1);               // Terminate execution.
    }
    else
    for (int i = 0; i < capacity; ++i)
    {
        cout << "Enter integer number " << i + 1 << ": ";
        cin  >> data[i];
    }
}
```

0.3. Deallocation of Dynamically Allocated Space

Let's discuss what happens when we finish using the data array that was dynamically allocated. If our program performs no other dynamic allocation, then there is no harm in allowing the array to exist until the program terminates, at which point the operating system will reclaim the entire program segment in RAM for some other program. If, however, there is any possibility that our program might make need to perform dynamic allocation *after* the point at which our data array is no longer needed, then we should ***deallocate*** the space occupied by the array so that it will be free for later dynamic allocations. The deallocation operation can be performed by any function that has a pointer to the first cell of the dynamically allocated array. Suppose the function is using the identifier "`score`" for that pointer. Then the statement

```
delete [] score;
```

will cause a run-time heap management routine to mark the array cells as belonging again to the "free" section of the run-time heap. How does the heap management routine know how many cells to deallocate? The answer is easy: it consults the "red tape" data area before the first cell, where the cell size and number of cells were recorded at the time the array was allocated.

Why are empty square brackets included in the "`delete`" statement above? They are used to indicate to the compiler that an array is to be deallocated, and that the destructor (if any) for the data objects in the array is to be called on every cell of the array. There is a simple rule that you should follow concerning square brackets in a "`delete`" statement on a pointer `p`:
if the space pointed to by `p` was allocated using square brackets (e.g., `p = new float[n];`), then write `delete [] p;` if the space pointed to by `p` was *not* allocated using square brackets (e.g., `p = new Node; )` , then write `delete p;` . Failure to follow this rule can sometimes produce programs that crash for no apparent reason.

What would happen if you called "`delete`" on a pointer that was not pointing to the first byte of a dynamically allocated memory space? The answer depends on the heap management routine supplied by the compiler. Some routines blindly proceed, which means that they consult the bytes just to the left of the byte pointed to, and then modify those bytes. This can cause astonishing "bugs" in programs. Other management routines are slightly more sophisticated, and so they may (or may not) detect that an error has occurred. In any case, you want to take pains to avoid such an error.

The "`delete [] score;`" statement will not change the value contained in the pointer variable "`score`", nor will the statement "erase" all the data in the deallocated array. Thus it would be possible (though reckless and foolhardy) to have the program continue to access the data in the array "`score`" even after the array has been deallocated. The principal danger here is that if the program later executes a statement that calls for dynamic allocation, the management routine is likely to allocate part or all of the space occupied by the deallocated array. Then new data will probably be written into the freshly allocated area, overwriting the old array data. But other things can go wrong, even if no further dynamic allocation takes place anywhere in the program For example, if a large array is marked as "free", then the operating system -- which often keeps part of your executing program in blocks on the hard disk -- may discover that the block containing your array is no longer

in use, and when you attempt to access the array, the operating system may refuse to allow the access. (Note: this is a point you will understand better when you have taken our Operating Systems class.) So TAKE NO CHANCES, because when critical programs crash, they can kill people. Never deliberately make your program access deallocated memory space.

A bug that's particularly hard to find occurs when a programmer makes two or more pointers point to a dynamically allocated array and then later uses one of those pointers to deallocate the array. If the programmer is not aware of the fact that the other pointers are now accessing array cells that have been deallocated, then the program is likely to work correctly up to the point at which a fresh allocation occurs and new data is written. Bugs like this are extremely difficult to track down.

0.4. Using Multiple Files

Real life programs are usually made of thousands and even millions of instructions. It is not manageable to have them all in one source file, this is why all modern IDEs provide means to split the source code of a program into several source file. Some of the advantages and reasons for it:

- It makes the code easier to read and facilitates the search for specific information in the program.
- Easier to maintain and upgrade.
- Better suited for team work.
- Increases the reusability of the code.

General Principles

Global variables are visible only in the file where they are declared. They should only be declared in one source file, and any other file where they are needed should declare them as *extern*.

Each class should in general be defined in their own header file (with a similar name) and implemented in their own source file, with the same name as the header file.

Any other data structures and type definitions should also be placed in header files to make them easier to use in more than one source file.

Functions that are not part of any class should be grouped by category in a sensible way.

The prototypes of all the function contained in a source file should be placed in a header file with the same name. Any other source file that needs to call those functions can then include the header file containing the prototype.

Compiling Projects with Several Source Files

Each source file is compiled separately into an object containing the machine language code for all the functions and classes in that source file. The result is a file with the same name and the extension **.o**. Technically an object contains some extra information beside machine language code that allows the link editor to link functions, create function tables, and relocate the code as needed.

The compiler under Linux is **g++**. The option **–c** allows us to compile a file and generate the object file but not the executable. For example, the command

```
g++ MyArray.cc –c
```

generates (if no errors are found) a file named **`MyArray.o`** .

For all the functions called from a source file, the compiler must know the name of the function, the type of the returned value, and the type of each parameter along with how many parameters the function takes. This information is provided by the function prototype. Library header files like <time.h> contain prototypes for functions defined in the library.

A separate operation called *link editing* or simply *linking*, creates a link from every function call in all the object files to the actual machine language code of the function which can be found either in the same object file or in a different one.

The same procedure is applied to library functions which can be found in libraries, like for example **libm.a** containing the math functions. A file with this extension is an archive of several object files that have been compiled separately. Other versions of the same library could have the extension **.so**, meaning shared object, which are dynamically linked at runtime, making it easier to upgrade these libraries without having to recompile the programs. They are the equivalent of DLLs (Dynamically Linked Library) under Windows.

The link editor under Linux is **g++**. A compiling command that creates an executable (without the option –c) must specify all the objects that need to be linked for the program to be completely defined. The user-generated objects are featured as arguments of the command, while other libraries are linked with the **–l** option. For example, the command

```
g++ main.cc –o array MyArray.o –lm
```

compiles the source file **main.cc** and create an executable named array (option **–o**). For this purpose it links into the executable the object **MyArray.o** which has been created with a previous command with the extension –c, and the library **libm.a**.

Header Files

Header files should not be included more than once in a source file because that would generate compiler errors as the prototypes and type declarations they contain would be declared twice. It is not always easy to keep track of all the header files included in a source file because some header files can include other header files. It is even recommended for a header file to include all header files it needs, for example, those containing the declaration of types used in that header file.

To avoid including a header file more than once, a common procedure is used in C++ based on some precompiler commands.

```
#define constName value
```

This is a primitive way to define a constant, inherited from c. It is not recommended for general use, but it is often used for defining the compilation context. The value in the command is optional.

```
#ifdef constName
// first set of instructions
#else
// second set of instructions
#endif
```

This is called conditional compilation. If the constant with the given name has been defined with the precompiler command #define, the first set of instructions is compiled and the second one discarded. If the constant is not yet defined, then the first set is discarded and the second set is compiled. The else branch is optional.

```
#ifndef constName
// first set of instructions
#else
// second set of instructions
#endif
```

This is the opposite of the previous command. In this case the first set of instructions is compiled only of the constant has not yet been defined, and discarded otherwise.

We can use these precompiler commands to make sure that a header file is not compiled more than once in any source file, no matter how many times it is actually included. The usual syntax is the following::

```
#ifndef HEADER_CONST_H
#define HEADER_CONST_H
// contents of the header file
#endif
```

The constant should be an identifier that is not used anywhere else in the program. The first time that the precompiler encounters an instruction #include for this header file, the constant HEADER_CONST_H has not yet been defined, meaning that the content of the header file is compiled. By the same operation, the constant HEADER_CONST_H is being defined. This means that the second time the same header file is included in the same source file, the precompiler finds the constant to be already defined, and discards the entire content of the header file, insuring that it is not compiled again.

All the system header files and those from any publicly distributed library already contain such statements so the user doesn't have to take care of this aspect except for user-defined header files.

Notation. All the user-defined header files are included with the name in quotes, as for example,

```
#include "MyArray.h"
```

while all the system header files are included with the name in angle brackets, as in

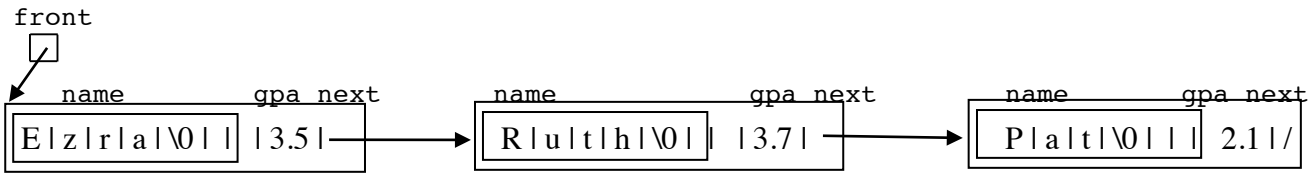
```
#include <time.h>
```

0.5. Linked Lists

In many programs, the amount of data that the program will have to hold and manipulate is not determined at any particular point in the life of the program (until the program ends, of course). When this is the case, experienced programmers will often create a "dynamic (storage) structure" to hold the data. A "dynamic structure" is one that can grow or shrink during the life of the program. The simplest kind of dynamic storage structure is known as a ***linked list.*** A linked list consists of ***nodes***, each of which has one or more data members and a pointer member whose job is to point to the next node in the list. A pointer is maintained to the node at the front of the list. When a new node must be set up to hold a new data object (or several data values), the node is dynamically allocated. Then data values are copied into the appropriate members of the node, and the new node is added to the list in some way. When a node must be removed from the list, it is tracked down in the list, cut out of the list, and then deallocated. The creation of a linked list must be preceded by the definition of a *node* data type. Here is an example.

```
const int STRING_MAX = 6;
struct Node                    // Defines a "node" data type.
{
  char   name[STRING_MAX + 1];
  float  gpa;
  Node * next;
};
typedef Node * NodePtr;   // Gives a convenient name to the  Node *
                           // data type.
```

The program in which the definition above occurs could then begin creating nodes, putting data into them, and linking them together to form a list that we can imagine looks like this:



This is a "logical view" of the linked list. The positions of the dynamically allocated nodes in the run-time heap may not match the picture above. The pointers may, therefore, be pointing here, there, and yon in the actual RAM.

Suppose we want to print out the name and gpa contained in the first node of this linked list. Here is one way to do it:

```
cout << (*front).name << "\t" << (*front).gpa << endl;
```

An alternative notation that most people prefer looks like this:

```
cout << front->name << "\t" << front->gpa << endl;
```

Typically we place a NULL pointer value in the `next` member of the last node in the list. The NULL acts as a sentinel value that can be used during searches of the list. When the list is empty, we customarily signal this by giving the value NULL to the pointer for the list (the pointer `"front"` in the example above). In all the problems we are about to solve, we shall assume these two conventions about lists. When drawing a picture of a NULL-terminated linked list, we'll use a forward slash (`/`) to indicate the NULL value in the last node of the list.

We're now going to write several functions that can operate on linked list data structures of the kind we are considering. We'll begin by writing a function that can count the **number of nodes** in a linked list. We'll pass to the function a pointer to the first node of the list. We'll make the function return the number of nodes it found in the list.

Next let's write a function that **prints out the data** in the linked list, one line per node. For example, if the function is given the list shown on the preceding page, then the function should print

```
Ezra   3.5
Ruth   3.7
Pat    2.1
```

We'll pass to the function a pointer to the first node of the list. The function need not return a value.

Let's write a function that **locates the last node** in a linked list and returns a pointer to that node. If the list is empty when the function is called, the function will return the NULL value.

Let's write a function that creates a node, assigns a string and a floating point number to the data members of the node, and **inserts the new node at the front of a list**. We'll pass to the function a pointer to the first node of the list, a pointer to a name, and a grade point average. The function will modify the node pointer it has been given so that the argument that's passed to the function will be made to point to the new node at the front of the list. If the list is empty when the function is called, the new node will become the first and only node of the list. We'll make the function return the logical value `true` ("succeeded") if it succeeds in dynamically allocating a node and putting the data into it. If dynamic allocation fails, we'll return `false` ("failed"). (That is, we are using the built-in data type `bool` in the newer versions of C++. See page 1-11 in these class notes.)

Let's write a function that **removes and deallocates the first node** in a linked list. We'll pass to the function a pointer to the first node of the list and two reference parameters in which the data in the node can be carried back to the calling function. If the list is empty, the function will perform no useful work and will return `false` . If the list is non-empty, the function will return `false` .

Let's write a function that **removes and deallocates the last node** from a linked list. We'll pass to the function a pointer to the first node of the list. If the list is empty, the function will perform no useful work. We'll make the function return the logical value `false` if the list is empty, and `true` if the list is non-empty. The data in the deallocated node will be discarded.

Let's write a function that **searches a linked list for a prescribed name**. We'll pass to the function a pointer to the first node of the list and a pointer to a name (string). The function will search the names in the list for a copy of that name. We'll make the function return a pointer to the first node in which it finds the prescribed name, or NULL if the name cannot be found.

Let's write a function that searches a linked list and returns a pointer to the node containing the **largest GPA**. If the list is empty when the function is called, the function will return NULL.

Let's write a function that can **copy a linked list**. The function will be passed a pointer to the list to be copied. It will return a pointer to the newly constructed list. If it cannot construct a copy because of dynamic allocation failure, it will terminate the entire program with an error message.

0.6. Containers and Iterators

Container classes

Def. A *container* is a class whose instances are collections of other objects.

Variants of containers can be found in many libraries and toolkits, especially in computer graphics. Their goal is to organize the storage of objects of a particular type and make their access easier for applications. An important point for containers is the safe implementation of their methods which should make provisions for unreliable calls from the client applications. Examples include stacks, queues, lists, and arrays.

In general, a container class is expected to implement *methods* for the following purposes:

- tell the number of objects they contain (size),
- insert new objects into the container and remove objects from it.

More methods could be required depending on the type of container or to the type of the stored objects.

Containers can be of two types: value based containers and reference based containers.

Value based containers will store copies of the objects, just like value parameters. Accessing an object will also give you back a copy of it. It is typical when the stored objects are simple types: int, char, etc.

Reference based containers only store pointers to the objects. This is typically the case when the stored objects are of a complex class and the duplication of such objects is costly.

Iterator classes

Def. *Iterators* are classes that can be seen as container browsers.

Iterators provide convenient and safe methods that facilitate the access to objects stored in a container. Typically, a container does not grant the client direct use to the stored objects for reasons of security and encapsulation. Some operations may require access to these stored objects beyond what is provided by the container class methods. In this case the client is granted more detailed access to the objects through an object oriented structure (the iterator) that makes sure that the operations will not result in memory violation even if the client performs operations that they shouldn't (like access an object of an array outside its range). The classical examples of iterators are indexes or subscripts to the elements of an array, or pointers to the objects stored in a linked list.

They can be seen as a general safe implementation of pointers to objects stored in the container.

They usually implement *methods* to go from an object in the collection to the next and to access the content of a stored object. In C++, these methods would typically be implemented as the operator **++** and ***** respectively.

0.7. Object-Oriented Implementation of Lists

The object oriented implementation of a list usually requires 3 classes: a list node, a list container, and a list iterator.

The *list node* class is a wrapper around one object to be stored in the list, contains the necessary pointers to build the structure of the linked list and a pointer or a copy of the stored object. This class implements functions to create and destroy a list node, as well as internal functions related to a node. Typically the only usage of this class is within the list container and list iterator classes.

The *list container* class is a container class storing an entire linked list, all the pointers needed for efficient operation on the list, as well as other related information that we might find useful, like the number of nodes. Typically a list container has data members for the head or front of the list, eventually the tail or back, the size. It implements methods that represent global operations defined on a list. It is also the interface between the client application and the list. For this reason, this is the class that is called *List*.

The *list iterator* class is a commodity class that allows us to browse a list more easily.
A list iterator object usually contains one data member which is a pointer to a list node object and allows the client to go to the next node by a simple operation (++) without actually giving them direct access to that pointer. A second function of this class is to retrieve the actual data stored in the list node that it points to, usually implemented as the dereference operator. This operator will not result in a memory violation or segmentation fault if the list node it points to is null. List iterators are also used to mark special positions inside a list, like where a particular item can be found in the list.

A list iterator class usually implements at least the following methods:

- advance (operator ++),
- dereference (operator *),
- check if the pointer to a list node is null (operator bool).

All these operations must be implemented in a safe way, such that if the client does something wrong the program doesn't crash.

```
template <class T>
class ListNode {
private:
    T datum;
    ListNode *next;
public:
    ListNode();
    ListNode(T value, ListNode &other = NULL);
    ListNode(ListNode &other);
    ~ListNode();
};

template <class T>
class ListIterator {
private:
    ListNode<T> *current;
public:
    ListIterator &operator++();
    T &operator*();
    operator bool();
    friend class ListNode;
};

template <class T>
ListIterator &ListIterator :: operator++()
{
    if (current)
        current = current->next;
    else
        cerr << "Error moving to the next list node." << endl;
    return *this;
};
```

```
template <class T>
class List {
private:
    ListNode<T> *head, *tail;
    int size;
public:
    List();
    ~List();
    void insertFront(T value);
    void insertEnd(T value);
    bool removeFront(T &value);
    bool removeEnd(T &value);
    // Iterators used to locate objects inside the list.
    ListIterator *begin();
    ListIterator *end();
    ListIterator *find(T target);
};

// Insert a new node with the specified content at the front of the
// list. It must make sure to update both the head and the tail of
// the list if need be.
<template class T>
void List::insertFront(T value)
{
    ListNode *newNode = new ListNode(value, head);
    size++;
    head = newNode;
    if (size == 1)
        tail = newNode;
}
```

Written Exercises

0-1. What output is produced by the following fragment of C++ code?

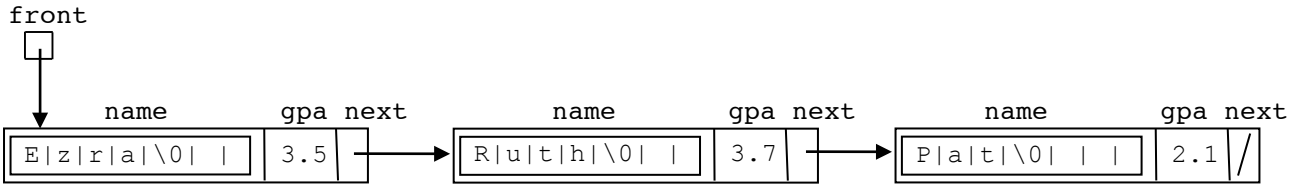
```
int i = 5, j = 7;
int * p = &i;
int * q = &j;
*p += *q;
cout << "i = " << i << endl;
```

0-2. What output is produced by the following fragment of C++ code?

```
int i = 5, j = 7, k;
int * p = &i;
int * q = &j;
k = *p;
*p = *q;
*q = k;
cout << "i - j = " << i - j << endl;
```

0-5. Suppose we have made the following definitions and constructed the linked list shown below.

```
const int STRING_MAX = 6;

struct Node                    // Defines a "node" data type.
{
    char   name[STRING_MAX + 1];
    float  gpa;
    Node * next;
};

typedef Node *NodePtr;   // Gives a convenient name to the  Node *
                          // data type.
NodePtr front;
```



What output is produced by the following fragment of C++ code?

```
NodePtr p = front->next;
NodePtr q = p->next;
p = q;
cout << (front->next)->name << endl;
cout << q->name << endl;
```

0-6. Assume the same definitions and linked list as in Exercise 0-5. What output is produced by the following fragment of C++ code?

```
NodePtr p = front->next;
NodePtr q = p->next;
front->next = q;
q->next = p;
p->next = NULL;
for (p = front; p != NULL; p = p->next)
    cout << p->name << endl;
```

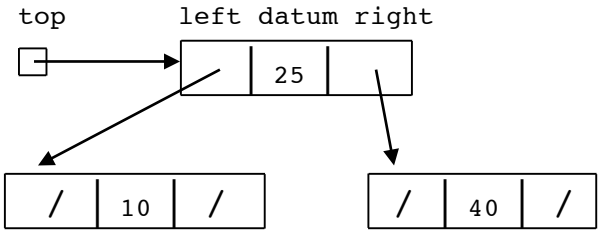
0-9. Assume the same definitions and linked list as in Exercise 0-5. What is the difference -- if any -- between the output produced by the following two C++ statements:

```
cout << (*front).gpa << endl;
cout << front->gpa << "\n";
```

0-12. Suppose we have made the following definitions and built the linked structure shown below.

```
struct Node                    // Defines a "node" data type.
{
    int  datum;
    Node * left, * right;
};

typedef Node *NodePtr;   // Gives a convenient name to the  node *
                          // data type.
NodePtr top;
```

structure:



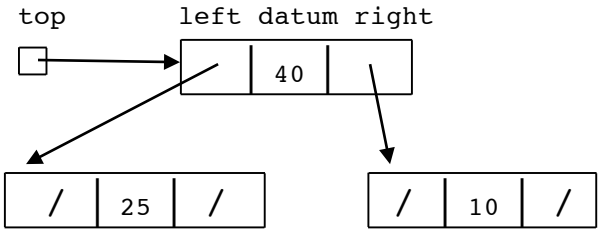
(By our convention, the cells marked with / hold the NULL pointer value.)

Now consider the following fragment of C++ code. Work through it carefully to figure out exactly what it does.

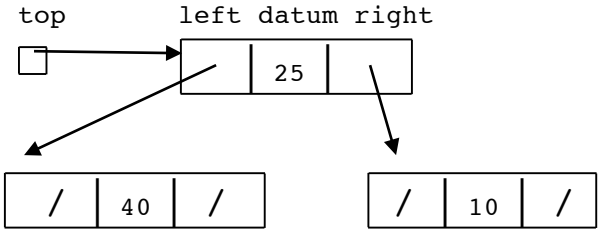
```
NodePtr temp = top;
top = temp->left;
temp->left = NULL;
top->left = temp->right;
temp->right = NULL;
top->right = temp;
```

Which -- if any -- of the diagrams below shows the correct logical view of the structure after the code at the bottom of the preceding page has been executed?

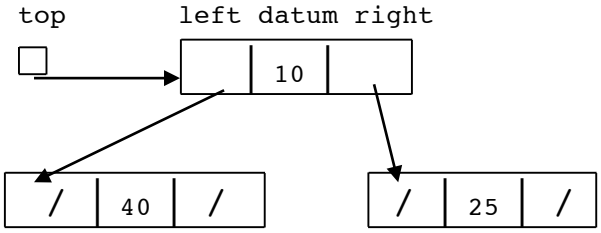
structure (1):



structure (2):



structure (3):



Solutions for Odd-Numbered Exercises

0-1. The second and third lines of the code make `p` and `q` point to the RAM addresses where the values of `i` and `j` are stored. Then fourth line of the code assigns the value pointed to by `q` to the memory cell pointed to by `p`, which means that the value `7` pointed to by `q` is added to the value `5` pointed to by `p`. This changes the value of `i` to `12`. Thus the output is

```
i = 12
```

0-5. The first line of the code makes `p` point to the node containing the name `Ruth`. The second line makes `q` point to the node containing `Pat`. The third line makes `p` point to the node containing `Pat`. The fifth line prints the name `Ruth`. The sixth line prints the name `Pat`. (The third line really did nothing useful, but some students get confused and think that the first line makes `p` an "alias" for the pointer in the node containing `Ezra`, so when `p` is assigned the value of `q`, they think that this changes the pointer in the `Ezra` node. That is an error, however.)

0-9. There is no difference. The expression `"front->gpa"` is just an alternative way of writing the more complicated and hard-to-read expression `(*front).gpa`.

Solutions to Problems Posed on Pages 0-7 and 0-8

```
/***************  N U M B E R   O F   N O D E S  ****************
This function counts the number of nodes in the NULL-terminated linked
list pointed to by the parameter "p" and it returns that number.
Documented by W. Knight.  Coded by W. Knight. */

int numberOfNodes (NodePtr p)
{
    int count = 0;

    while (p != NULL)    // can also be written simply as "while (p)"
    {
        ++count;
        p = p->next;
    }

    return count;
}


/*********************  P R I N T   L I S T  **********************
This function prints all the data in the NULL-terminated linked list
pointed to by "p".  If the list is empty, a message to that effect is
printed.  Documented by W. Knight.  Coded by W. Knight. */

void printList (NodePtr p)
{
    if (p == NULL)      // Can also be written simply as "if (!p)"
        cout << "The list is empty."  << endl;

    else

        while (p)        // Same as "while (p != NULL)"
        {
            cout << p->name << "\t" << p->gpa  << endl;
            p = p->next;
        }
}


/************  P O I N T E R   T O   L A S T   N O D E  *************
This function traverses the NULL-terminated linked list pointed to by "p"
and returns a pointer to the last node, provided the list is not empty.
If the list is empty, the function returns the pointer value NULL.
Documented by W. Knight.  Coded by W. Knight.  */

NodePtr pointerToLastNode (NodePtr p)
{
    if (!p)
        return NULL;

    else
    {
        while (p->next)    // same as "while (p->next != NULL)"
            p = p->next;

        return p;
    }
}
```

```
/*****************  I N S E R T   A T   F R O N T  ******************
This function dynamically allocates a new node, copies the parameter
string "s" into the name member of the new node, and copies the parameter
"grade" into the gpa member of the new node.  Then it inserts the node at
the front of the NULL-terminated linked list pointed to by "p".  Thus, if
the list was empty, the new node becomes the first and only node in the
list.  The function returns true provided it was able to allocate the new
node.  If the dynamic allocation fails, it returns false.  Documented by
W. Knight.  Coded by W. Knight. */

bool insertAtFront (NodePtr & p, char s[], float grade)
{
    NodePtr temp = new Node; // Declare a pointer variable and make
                             // it point to a newly allocated node.

    if (!temp)                // If allocation fails, leave function.
        return false;
    else
    {
        strcpy (temp->name, s);   // The algorithm expressed in these
        temp->gpa = grade;        // four lines is a very important
        temp->next = p;           // one.  Note that no test needs to
        p = temp;                 // be made to see whether p is NULL.
        return true;
    }
}

/**************  R E M O V E   F R O N T   N O D E  ***************
This function removes the first node from the NULL-terminated linked list
pointed to by "p" and deallocates it, provided the list is not empty.  If
the list is empty, the function returns false (failure).  If the list is
not empty, the function copies the name and gpa from the node into the
parameters of the same names and then returns true ("success")  Documented
by W. Knight. Coded by W. Knight. */

bool removeFrontNode (NodePtr & p, char name[], float & gpa)
{
    if (!p)
        return false;

    else
    {
        strcpy (p->name, name);  // Copy the node data into the
        p->gpa = gpa;            // parameters.

        NodePtr temp = p;    // We need an auxiliary pointer to
        p = p->next;          // hold onto the node to be deallocated
        delete temp;          // while p is advanced to the next node.
        return true;
    }
}
```

QUESTION: why don't we need to have an ampersand (`&`) in front of the parameter `name` the way we do in front of the parameter `gpa` ?

```
/***************  R E M O V E   L A S T   N O D E  ****************
This function removes the last node from the NULL-terminated linked list
pointed to by "p" and deallocates it, provided the list is not empty.  If
the list is empty, no action is taken.  If the list was not empty, the
function returns true ("success"), while if the list was empty, the
function returns false ("failure").  Documented by W. Knight.  Coded by W.
Knight. */

bool removeLastNode (NodePtr & p)
{
    if (!p)  // if p == NULL;  the list is empty
        return false;

    else if (!p->next)  // if p->next == NULL;  only 1 node in the list
    {
        delete p;
        p = NULL;
        return true;
    }

    else  // There are at least 2 nodes in the list.
    {
        NodePtr runner = p->next; // will "run" along the list
        NodePtr prior = p;        // will be one step behind runner

        while (runner->next)  // while there's a node beyond *runner
        {
            prior = runner;
            runner = runner->next;
        }
        // When loop ends, runner points to last node of the list.

        delete runner;
        prior->next = NULL;
        return true;
    }
}


/******  L O C A T I O N   O F   S P E C I F I E D   N A M E  *******
This function searches the NULL-terminated linked list pointed to by "p"
for a name matching the string "s".  If it finds a node containing such a
name, it returns a pointer to that node.  If not, it returns NULL.
Documented by W. Knight.  Coded by W. Knight. */

NodePtr locationOfSpecifiedName (NodePtr p, char s[])
{
    while ( p != NULL  &&  strcmp (s, p->name) != 0 )
        p = p->next;

    return p;
}
```

Could write the loop condition this way:

```
while ( p  &&  strcmp (s, p->name) )
```

```
/********************  H I G H E S T   G P A  *********************

This function returns a pointer to the node containing the largest GPA in
the NULL-terminated linked list pointed to by "p".  If the list is empty,
the function returns the NULL value.  Documented by
W. Knight.  Coded by W. Knight. */

NodePtr nodeWithHighestGpa (NodePtr p)
{
    if (!p)
        return NULL;

    else
    {
        NodePtr highest = p; // First GPA is the "highest so far".

        p = p->next;
                                    // ALTERNATIVELY:
        while (p)                   // for (p = p->next; p; p = p->next)
        {                           //   if (p->gpa > highest->gpa)
            if ( p->gpa > highest->gpa ) //     highest = p;
                highest = p;

            p = p->next;
        }

        return highest;
    }
}

/********************  C O P Y   O F   L I S T  *******************

This function creates a copy of the NULL-terminated linked list pointed to
by the parameter "p".  It then returns a pointer to the newly created
list.  If the copy cannot be made because of a failure of dynamic
allocation, then the entire program is terminated with an error message.
Documentation by W. Knight.  Code by W. Knight. */

NodePtr copyOfList (NodePtr p)
{
    if (!p)
        return NULL;  // Signifies an empty list.

    else
    {
        NodePtr front = new Node;   // Get 1st node of the new list.
        if (!front)                  // If front == NULL
        {
            cout << "Dynamic allocation failure for front." << endl;
            exit(1);                    // Terminate the program.
        }
        strcpy (front->name, p->name); // Copy name from first node.
        front->gpa = p->gpa;           // Copy gpa  from first node.

        NodePtr back = front;         // Create a pointer to point to
        p = p->next;                   // the back of the new list.

        while (p)   // While not beyond the end of the old list
```

```
        {
            back->next = new Node;

            if (!back->next)                // if back->next == NULL
            {
                cout << "Dynamic allocation failure: back->next.\n";
                exit(1);                    // Terminate the program.
            }

            back = back->next;      // Advance "back" to the new node.

            strcpy (back->name, p->name); // Copy name and gpa from
            back->gpa = p->gpa;           // the node in the old list

            p = p->next;                  // Advance p to the next
        }                                // node in the old list.

        back->next` = NULL;

        return front;
    }
}
```

Important Formatting Remark

The code for the function above is "broken" over two pages. AND VERY BADLY. The page break occurs just after an `"while"` line. Breaking the code that way is very inconsiderate of the prospective readers of the code, because the readers will have to flip back and forth between these two pages to read the code with full comprehension.

Let's not do this to each other! In these notes, I will try to keep all the code for a function on a single page. When that proves impossible, I will try to insert a page break at the end of some "logical block", so that when you are reading my code, you will not have to flip back and forth between two pages any more than is necessary. I request that you show me the same courtesy.

When using the Emacs editor, you can arrange for a page break at various lines in the file by placing the character Ctrl-L ("form feed") at the beginning of an empty line. This requires a special pair of keystrokes, however, because if you simply type Ctrl-L , the editor will reposition the window so that the blinking cursor is in the center of the window. The way to place Ctrl-L into your file as a character in the file is to type Ctrl-Q Ctrl-L (two keystrokes) . The Ctrl-Q character is treated as an "escape character" by the editor, so that the next character typed is understood as that actual character and not just a command to the editor.

1. The "Stack" ADT (Abstract Data Type)

Definition: A **stack** (also called a **pushdown stack**) is an abstract data structure into which we can insert objects of some specified type in any order whatsoever, but the objects can be removed only in reverse order from the order in which they have been inserted. (For this reason, a stack is called a Last-In-First-Out, or LIFO, data structure.) The operation of inserting an object into a stack is called "pushing the object onto the stack". The operation of removed an object from a stack is called "popping the top object off the stack". Several other operations are usually available on stacks:

(1) create an empty stack; this operation is sometimes called Make-Stack.
(2) force a stack to be empty; i.e., discard all the objects in the stack; this operation is sometimes called Make-Empty.
(3) examine the top object on the stack without popping it off the stack; this operation is sometimes called simply Top.
(4) query a stack to determine whether it is empy; this operation is sometimes called Is-Empty.

Example: Suppose we have a stack of integers, and we perform the following operations:

push 3;
push 5;
pop the stack and print the removed integer;
push 7;
push 9;
pop the stack and print the removed integer;
pop the stack and print the removed integer;
push 11;
pop the stack and print the removed integer.
pop the stack and print the removed integer.

What will be printed?

Where in computer science do stacks occur? By way of answer, a knowledgeable person is tempted to ask, "Where don't they occur?" They are ubiquitous!

Every time a C/C++ program runs, the linker-loader program (i.e., the part of the operating system that chooses a segment of memory in which the program will be placed and that loads the program into that segment) sets up an "empty" portion of the program's memory segment to be used for the **run-time stack**. Whenever any function in the program is called (and this includes even the main function), a **stack frame** is placed on the run-time stack to hold all the variables associated with the function and also the "return address" for the moment when the function ceases execution. Let's take an example. Suppose we have a function that looks like this:

```
int greatest_common_divisor (int m, int n, bool & error)
{
    int quotient;
    int remainder;

    if (m == 0 && n == 0)
    {
        error = true;
        return 0;
    }

    else
    {
        a bunch more code
        return quotient;
    }
}
```

Now suppose another function contains the following two lines of code.

```
d = greatest_common_divisor (numerator, denominator, faulty);

if (!faulty)
{
    ....
```

What happens when the function call shown above is executed? A stack frame is set up on (pushed onto) the run-time stack. The following information is then copied into the stack frame:

- the machine address of the next instruction after the function call into a field that we can call "return address";
- the *value* of "`numerator`" is copied into the formal parameter space "`m`";
- the *value* of "`denominator`" is copied into the formal parameter space "`n`";
- a *pointer* to "`faulty`" is placed in the formal parameter space "`error`";
- a space is reserved for the local variable "`quotient`";
- a space is reserved for the local variable "`remainder`";
- a space is reserved for the return value of the function.

Then the code for the function begins to execute. Any time the parameters "`m`" and "`n`" or the local variables "`quotient`" and "`remainder`" are accessed, they are accessed in the stack frame; if there are variables with these names in the calling function, those variables will not be accessed. Any time the parameter "`error`" is accessed, the pointer to "`faulty`" in the calling function will be dereferenced, and changes that the `greatest_common_divisor` makes in the parameter "`error`" will actually be made in the variable "`faulty`". Thus "`error`" is really just an alias for "`faulty`". When the function executes a "`return`" instruction, the value being returned is copied into the space reserved for it, and then control is transferred back to the instruction pointed to by the "return address". The stack frame is considered to have been "popped" off the stack.

If at any time during its execution, the `greatest_common_divisor` function calls another function, a stack frame for that function call will be pushed onto the run-time stack and the code executed by that function will access the variables in that stack frame (except for reference parameters).

Here is another application of stacks. Next time you are using the Emacs editor, try typing the following lines:

```
{
    if ( (x + a[initial_index]) < f(u, v) )
        cout << a[initial_index];
}
```

If you keep your eye on the screen while you are typing, you will see that when you type the first right bracket, the cursor will jump back momentarily to the matching left bracket; when you type the first right parenthesis, the cursor will jump back momentarily to the matching left parenthesis; when you type the final brace, the cursor will jump back momentarily to the opening brace. What the Emacs editor is doing is maintaining (out of your sight) a stack of characters along with their text-positions. When you type a left brace, a left parenthesis, or a left bracket, the editor pushes a copy of that character, together with its text-position, onto the stack. Whenever you type a right brace, a right parenthesis, or a right bracket, the editor examines the top of the stack to make sure that your right grouping symbol is properly paired with a left grouping symbol; if it is, the editor pops the stack, discards the copy of the left symbol, and moves the cursor back to that position in the text for a little over a second.

1.1 Array Implementation of a Stack

General description of the class. The objects will be stored sequentially in the array. Adding new objects to the end of the array, removing them from the end also. The capacity is the maximum number of objects that can be stored in the stack, the size is the number of objects currently stored in the stack. The only reason this is inefficient is for the limitation in capacity.

```
template <class T>
class Stack<T> {
private:
    T *storage;
    int size, capacity;
public:
    void push(T);
    T pop();
    T top();
    void make_empty();
    bool is_empty();
    void increment_size();
};

// A prototype.
void error(char *message);

// Insert an object at the end of the array.
void Stack::push(T obj)
{
    if (size >= capacity)
        increment_size();
    storage[size]= obj;
    size++;
}

// Remove an object. If the stack is empty, we must exit with an
// error because there's no value we can return to warn the
// client.
T Stack:: pop()
{
    if (size)
    {
        --size;
        return storage[size];
    }
  else
      error("pop operation on an empty stack");
}
```

```
// Inspect the front object. If the stack is empty, we must exit
// with an error because there's no value we can return to warn the
// client.
T Stack:: top()
{
    if (size)
        return storage[size-1];
    else
        error("inspecting the top of an empty stack");
}

// This one is really simple: assert that we don't have anything in
// the stack. If the elements of the array are pointers to external
// objects, we should delete each of them.
void make_empty()
{
    size=0;
}

// Check the size, that's all.
bool is_empty()
{
    return (size == 0);
}

// Double the size each time we reach the upper limit.
void Stack::increment_size()
{
    T new_array = new T[2*capacity];
    for (int i=0; i<capacity; i++)
        new_array[i] = storage[i];
    delete [] storage;
    storage = new_array;
    capacity = 2*capacity;
}

// Not a class method. Display the error message and exit.
void error(char *message)
{
    cout << message << endl;
    exit(1);
}
```

1.2. An Implementation in C++ of the Stack ADT

On the following pages you will find C++ code for a class named "`stack_of`" that implements the stack ADT. The code is in a header file named "`stack.h`". We are going to examine the code in considerable detail. This will provide a kind of review of C++ classes with their data members and function members, including constructors, destructors, and overloaded operators.

Documenting for Two Audiences

The documentation at the top of the following header file is addressed to the "audience" of programmers of client code, i.e., code that will make use of the definitions in this header file. The documentation addressed to these programmers describes how to use the "stack interface" provided here. By the "stack interface" we mean the public member functions that can be invoked in client code to create and manipulate stacks of various kinds. The client programmers do not need to know anything about the actual implementation of these public functions, and so we do NOT describe for them the data members and algorithms that have been used to code the member functions.

There is other documentation in the following file. It occurs in many places beyond the point in the file where the definition of the "`stack_of`" class appears. This other documentation is addressed to a different "audience" consisting of programmers who want to understand -- or possibly modify -- the stack implementation given in this file. Each member function is carefully documented to describe exactly how it will behave when it is called.

If we divide the documentation in the file into documentation for client programmers and documentation for implementers, then it is possible for us to modify the implementation without changing even one word of the documentation for clients -- assuming we do not intend to change how the member functions will behave. The only documentation we will have to modify will be the implementer documentation, i.e., the documentation that explains the details of the implementation.

The "`bool`" (Logical) Data Type

This file uses the built-in simple C++ data type named "`bool`", which is short for "Boolean" (the reference is to George Boole, an English logician of the nineteenth century). The "`bool`" data type has only two values, namely `false` and `true`, which can be assigned to any variable that has been declared to be of type `bool`. If you force a program to print the values of `false` and `true`, the program will print `0` (zero) and `1` (one) respectively. If you assign values of other simple data types (e.g., int, float, char, pointer) to a `bool` variable, the value will be cast to `0` or `1`.

Examples:

```
bool p, q, r;
p = true;
cout << p << endl;         // will output 1
q = false;
cout << q << endl;         // will output 0
cout << !q << endl;        // will output 1
cout << p && q << endl;    // will output 0
r = 3.1416;                // will type cast 3.1416 to true
cout << r << endl;         // will output 1
r = NULL;                  // will type cast NULL to false
cout << r << endl;         // will output 0
```

Class Exercise on Stack Implementation (as given in the file `stack.h`)

```
#include <iostream>                                  // 1
#include "stack.h"                                   // 2

void busywork(int k);                                // 3
stack_of<float> float_stack(stack_of<int> t); // 4

main()                                               // 5
{
    busywork(99);                                    // 6

    return 0;                                        // 7
}

void busywork(int k)                                 // 8
{
    stack_of<int> v;                                 // 9
    v += k;                                          // 10

    for (int i = 1; i <= 10; ++i)                    // 11
        v += i * i;                                  // 12

    stack_of<int> w = v;                             // 13

    ____________________________;  // 14 Make stack v empty.

    if (_____________________)  // 15 If the stack w is not empty

        _____________________;  // 16 Change the top int on w to -1.

    stack_of<float> y;                               // 17

    y = float_stack(w);                              // 18

    return;                                               // 19
}

stack_of<float> float_stack(stack_of<int> t)
{
    stack_of<float> f;                               // 20

    // Pop the integers off the stack t, one by one, until the
    // stack is empty, and push each popped int onto the stack f.

    ____________________________                     // 21

    ____________________________                     // 22

    return f;                                        // 23
}
```

Written Exercises

1-1. Suppose you have created an empty stack, call it S , of integers. Suppose you then perform the following operations in the order shown.

Push 10 onto S .
Push 11 onto S .
Push 12 onto S .
Push 13 onto S .
Pop S and print the integer that was popped.
Push 14 onto S .
Pop S and print the integer that was popped.
Pop S and print the integer that was popped.
Pop S and print the integer that was popped.
Push 15 onto S .
Push 16 onto S .
Pop S and print the integer that was popped.
Push 17 onto S .
Pop S and print the integer that was popped.
Pop S and print the integer that was popped.
Pop S and print the integer that was popped.

Draw a picture of the stack as it evolves. Also list the order in which the integers are printed.

1-2. Suppose you have created an empty stack, call it S , of characters. Suppose you then perform the following operations in the order shown.

Push `'s'` onto S .
Push `'a'` onto S .
Push `'d'` onto S .
Pop S and print the character that was popped.
Push `'t'` onto S .
Push `'a'` onto S .
Pop S and print the character that was popped.
Pop S and print the character that was popped.
Pop S and print the character that was popped.
Push `'e'` onto S .
Push `'t'` onto S .
Push `'s'` onto S .
Pop S and print the character that was popped.
Pop S and print the character that was popped.
Push `'r'` onto S .
Pop S and print the character that was popped.
Push `'u'` onto S .
Push `'u'` onto S .
Pop S and print the character that was popped.
Push `'t'` onto S .
Push `'c'` onto S .
Pop S and print the character that was popped.
Pop S and print the character that was popped.

Pop S and print the character that was popped.
Push `'r'` onto S.
Pop S and print the character that was popped.
Pop S and print the character that was popped.
Pop S and print the character that was popped.

Draw a picture of the stack as it evolves. Also list the order in which the characters are printed.

1-4. Below are two versions of a C++ function named "`get_a_line`". When either of these versions is called, it returns a pointer to a character string that has been entered by an interactive user at the keyboard. *One* of these two versions has a serious flaw. If a program makes a call to this flawed version of the function, the user may find that the function seems to work, but then the string may mysteriously become corrupted after a time.

Identify the flawed version of the function, and explain fully, with the help of a diagram of computer memory, exactly what happens when the flawed function is called by another function, as in the following example:

```
char * user_info;
user_info = get_a_line();
```

and what might happen after this call is completed and further instructions are then executed (including calls to other functions).

```
/***************** G E T  A  L I N E  V E R 1 ***************
This function prompts an interactive user to enter a line of text
at
the keyboard.  The line is read into an array in the memory of the
computer, and a pointer to the first character of the array is
returned by the function.  Coded by A. Nonymous. */

char * get_a_line ()  // Version 1
{
    const int BUFFER_SIZE = 257;
    char buffer[BUFFER_SIZE];
    cout << "Enter some text on the line below and press Enter.\n";
    cin.getline (buffer, BUFFER_SIZE);
    return buffer;
}
```

```
/***************** G E T   A   L I N E   V E R 2 ***************
This function prompts an interactive user to enter a line of text
at
the keyboard.  The line is read into an array in the memory of the
computer, and a pointer to the first character of the array is
returned by the function.  Coded by A. Nonymous. */

char * get_a_line ()  // Version 2
{
    const int BUFFER_SIZE = 257;
    char * buffer = new char[BUFFER_SIZE];
    if (!buffer)
        exit(1);
    cout << "Enter some text on the line below and press Enter.\n";
    cin.getline (buffer, BUFFER_SIZE);
    return buffer;
}
```

1-8. Suppose we have written a non-template C++ class named "`student_record`" as follows:

```
class student_record
{
public:
    student_record (char s[], float grade_pt);   // Constructor
    ~student_record ();                          // Destructor
    student_record (const student_record & s);   // Copy
constructor
    student_record & operator= (const student_record &s);
        //Assignment
private:
    char * name; // Space for this array will be dyn'ly allocated
    float  gpa;
};

// The code for the member functions is not shown here.
```

(a) Suppose a client program defines a student record named "`worst_student`" as follows:

```
student_record worst_student("W. Knight", 1.9);
```

Which of the following is correct?

i. The copy constructor is called to initialize `worst_student`.

ii. The ordinary constructor is called to initialize `worst_student`.

iii. The assignment operator is called to initialize `worst_student`.

iv. First the ordinary constructor and then the copy constructor are called to initialize `worst_student.`

(b) Suppose the appropriate member function(s) is called in part (a). Which of the following is correct at the instant when the member function begins to execute?

i. That member function must first construct the character pointer `name` and the floating point number `gpa` before it can do its dynamic allocation and initialize the data members.

ii. The data members `name` and `gpa` already exist, but have garbage values; the job of the constructor is to give them appropriate values (taken from the parameters `s` and `grade_pt`).

iii. The data member `name` already exists and already points to a dynamically allocated array into which the parameter string `s` can be copied. Similarly, `gpa` already exists and is ready to have the value of the parameter `grade_pt` copied into it.

(c) Suppose the client program next introduces a student record named `"best"` as follows:

```
student_record best = worst_student;
```

Which of the following is correct?

i. The copy constructor is called to initialize `best.`

ii. The ordinary constructor is called to initialize `best.`

iii. The assignment operator is called to initialize `best.`

iv. First the ordinary constructor and then the copy constructor are called to initialize `best.`

(d) Suppose the student record `best` is passed by value to a function. Which of the following is correct?

i. The copy constructor is called to make a copy of the record being passed, and this copy is placed in the stack frame for the function call.

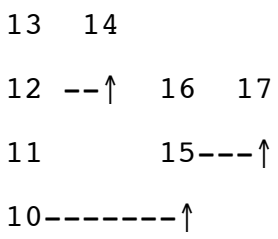
ii. The ordinary constructor is called to make a copy of the record being passed, and this copy is placed in the stack frame for the function call.

iii. The assignment operator is called to make a copy of the record being passed, and this copy is placed in the stack frame for the function call.

iv. First the ordinary constructor and then the copy constructor are called to make a copy of the record being passed, and this copy is placed in the stack frame for the function call.

Solutions for Odd Numbered Exercises

1. The following diagram shows how the stack evolves as the various operations are performed on it.



By placing the 14 on the same line as the 13 we indicate that the 14 has "taken the place of" the 13 in the stack. Similarly, 16 takes the place of 12 after 15 has taken the place of 11 . Etc.

Output: 13 14 12 11 16 17 15 10

2. The "Queue" ADT (Abstract Data Type)

Definition: A **queue** is an abstract data structure into which we can insert objects of some specified type in any order whatsoever, but the objects can be removed only in the same in which they have been inserted. (For this reason, a queue is called a First-In-First-Out, or **FIFO**, data structure.) The operation of inserting an object into a queue is called **enqueuing** the object into the queue The operation of removed an object from a queue is called **dequeuing** the object from the queue. The object that has been in the queue the longest (i.e., the object that will be removed at the next dequeue operation) is said to be at the **front** of the queue. The most recently inserted object in a queue is said to be at the **back** of the queue. In a queue that contains only one object, that object is both at the front and at the back of the queue. Several other operations are usually available on queues:

(1) create an empty queue; this operation is sometimes called Make-Queue.
(2) force a queue to be empty; i.e., discard all the objects in the queue; this operation is sometimes called Make-Empty.
(3) examine the front object in the queue without dequeuing it; this operation is sometimes called simply Front.
(4) query a queue to determine whether it is empty; this operation is sometimes called Is-Empty.
(5) concatenate two queues, i.e., place all the objects in one queue at the back of the other queue; this operation is (as you would expect) often called Concatenate.

Where in computer science do queues occur? One place is in the UNIX operating system, or for that matter, in any multi-tasking operating system. If a number of programs (more commonly referred to in operating systems theory as "processes") are running simultaneously under the direction of the operating, then they are actually running sequentially in little "time slices": one process will run for a short time, and the other processes will wait their turn in a *queue*. When the operating system's scheduler decides that the currently running process has had enough time, then that process is placed at the back of the scheduling queue and the process at the front of the queue is allowed to begin running.

Another place that a queue occurs in an operation system setting is in the message passing system. Under UNIX, one process can send a message to another process. The messages sent to a given process are enqueued and wait until the receiving process runs, at which time it can dequeue none, some, or all of the messages waiting for it.

Programs that perform simulations, i.e., computer modeling of the behavior of some system such as a collection of supermarket check-out lines, typically use queues to hold records representing the entities to be processed by the system.

Suppose we want to write a C++ class that implements the queue ADT. The queue class should be similar to the one we saw for stacks. Let's think of various ways in which it would be possible to design such an implementation.

Possible Queue Implementations

(1) A NULL-terminated linked list of nodes. Suppose each node has an "`datum`" member and a "`link`" member.

We will need a pointer, call it "`front`", to point to the first node in the list. We can set `front` to NULL to indicate that the queue is empty.

Since objects will be enqueued at the opposite end of the list from the front node, we will need to attach new nodes to the last node (if any) in the list. We have two choices here. One possibility is that each time we want to enqueue an item we can create a "search pointer", initialize it to point to the front node, and then make a linear search to the last node. An alternative (the one you should use in Program 4) is to maintain a pointer, call it "`back`", to the last node in the list whenever the list is non-empty (when the list is empty, `back` need not be set to NULL since we can check `front` to determine whether the queue is empty). The second alternative is far more time-efficient and so is almost always used. (Note that the choice between these two alternatives involves a SPACE-TIME TRADE-OFF.)

To enqueue a copy of an object x :

```
make an auxiliary pointer point to a newly allocated node;
copy  x  into the datum member of the new node;
place NULL in the link member of the new node;
if the list is not null
  make the link member of the last node point to the new node;
else
  make the front pointer point to the new node;
point "back" to the new node;
```

To dequeue the front object from the queue:

```
if the queue is empty,
  report an error and EXIT;
copy the front datum value into a temporary variable;
point an auxiliary pointer to the front node;
shift the front pointer so it points at the following node
   (or -- automatically -- to NULL if there are no more
nodes);
deallocate the node pointed to by the auxiliary pointer;
return the value stored in the temporary variable;
// No change should be necessary in the "back" variable; it
may
// point to the deallocated node, but we will not dereference
// "back" until we know the list is non-empty;
```

(2) Use a dynamically allocated array; expand it whenever it fills up and an object must be added. Keep track of the capacity of the array in a variable, say "`qcap`", and keep track of the number of objects in the queue with a variable, say "`qsize`". The front object in the queue is in cell `0`, in which case the last object will be in the cell with subscript `qsize - 1`. The queue is initialized by setting `qsize` to `0`.

To enqueue a copy of an object `x`:

```
if (qsize == qcap)
    enlarge the array;
copy  x  into cell  qsize  of the array;
++qsize;
```

To dequeue the front object:

```
if (qsize == 0)
    report an error and EXIT;
copy the front object to a temporary location;
shift all objects in the array left by one cell;
    // uses a loop!
--qsize;
return the object stored in the temporary location;
```

We can immediately sense that this is not a very efficient implementation. When the queue is long, each dequeue operation will require a lengthy loop to shift all the remaining objects. The next implementation seeks to reduce drastically the amount of shifting.

(3) Use an array but do "lazy shifting": shift only when the back of the queue bumps up against the right end of the array. We require an extra pointer (actually a subscript variable) to the front of the queue, say "`qfront`", and it now works better to have a pointer (subscript variable) to the back object in the queue, say "`qback`", instead of using "`qsize`". When `qfront == qback` this will mean that there is only one object in the queue. An empty queue will be signaled by having `qback` equal to `qfront - 1` (think about this!). We'll initialize the queue by setting `qfront` to 0 and `qback` to −1 .

To enqueue a copy of an object x :

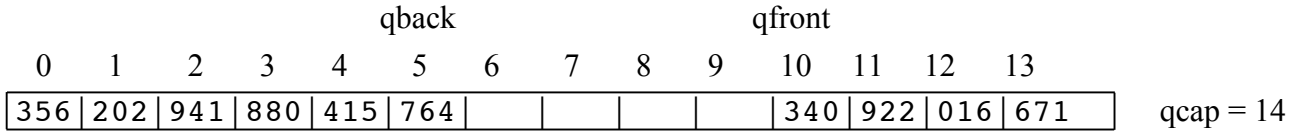
```
     if (qback == qcap - 1) // queue's back is at right end of
array
          if (qfront == 0)    // array is completely full
              enlarge the array;
          else
          {
              shift all objects to the left by  qfront  cells;
              reset qfront and qback properly;
          }
          ++qback;
          copy  x  into cell  qback  of the array;
```

To dequeue the front object:

```
     if (qback == qfront - 1)
          report an error and exit;
     else
     {
          ++qfront;
          return the object in cell  qfront - 1  of the array;
     }
```

This implementation will normally run much more efficiently than implementation (2), with only infrequent need to shift the objects in the storage array. A little thought reveals, however, that if the array becomes nearly full then the shifts *may* occur frequently and cause the queue to run inefficiently until the capacity is momentarily exceeded, at which point the storage array will be expanded.

(4) It is possible to get rid of the shifts entirely by using "wrap-around": we treat the array as if it were a ring, with cell 0 following cell `qcap-1`. Thus the *logical* back of the array may, after a time, lie to the left of the front of the array. That is, the picture might look like this:



In this way of doing things, cell 0 is *logically* just to the right of cell 13, and cell 13 is logically just to the left of cell 0.

One "bookkeeping" difficulty immediately presents itself: what happens if we fill the array so full that `qback` is equal to `qfront-1`? Remember that `qback == qfront-1` was used as the test for an *empty* queue. There are two ways out of this difficulty. One way is to make sure we keep the cell just to the logical left of `qfront` unfilled at all times; this "wastes" one cell equal to the size of the objects being stored. The other way is to re-introduce the `qsize` variable (see implementation (2)), and use it to keep track of the number of objects in the queue. Then when `qback` is just to the logical left of `qfront` we consult the `qsize` variable to determine whether the queue is empty or full. This requires one extra integer variable, which is less wasteful than the other method when the object size is large. Let's look at some pseudo-code for this second method. We initialize `qsize` to 0, `qfront` to 0, and `qback` to `qcap-1` (this is the subscript of the cell to the logical left of `qfront`).

To enqueue a copy of an object x :

```
     if (qsize == qcap)    // The array is completely full.
         enlarge the array; // Copying to the new array will
require                            // more thought than before
     if (qback == qcap-1)
         qback = 0;          // Wrap around.
     else
         ++qback;
     copy x into cell qback of the array;
     ++qsize;
```

To dequeue the front object: (see next page)

```
if (qsize == 0)
    report an error and EXIT;
else
{
    --qsize;
    if (qfront == qcap-1)
    {
        qfront == 0;
        return the object in cell  qcap - 1  of the array;
    }
    else
    {
        ++qfront;
        return the object in cell  qfront - 1  of the array;
    }
}
```

2.1 Queue Implemented in a Circular Array

```
template <class T>
class Queue<T> {
private:
    T *storage;
    int size, capacity, qfront, qback;
public:
    void enqueue(T &x);
    T dequeue();
    T front();
    void make_empty();
    bool is_empty();
    bool is_full();
    void increment_size();
};

// Insert an object in the queue.
void Queue::enqueue(T &x)
{
    if (is_full())
        increase_size();
    qback++;
    if (qback == capacity)
        qback = 0;
    size++;
    storage[qback] = x;
}

// Remove an object from the queue and return its value.
T Queue::dequeue()
{
    if (is_empty())
        error("dequeuing from an empty queue");
    T temp = storage[qfront];
    qfront++;
    if (qfront == capacity)
         qfront = 0;
    size--;
    return temp;
}

// Inspecting the value of the front object.
T Queue::front()
{
    if (is_empty())
        error("inspecting the front of an empty queue");
    return storage[qfront];
}
```

```
// Emptying the queue.
void Queue::make_empty()
{
    qfront = 0;
    qback = -1;
    size = 0;
}

// Check if the queue is empty.
bool Queue::is_empty()
{
    return (size == 0);
}

// Check if the queue is full.
bool Queue::is_full()
{
    return (size == catacity);
}

void Queue::increment_size()
{
    int i;
    T new_array = new T[2*capacity];
    for (i=qfront; i<capacity; i++)
        new_array[i-qfront] = storage[i];
    for (i=0; i<qback; i++)
        new_array[i+capacity-qfront] = storage[i];
    delete [] storage;
    storage = new_array;
}
```

Exercises

2-2. A *queue* is a FIFO structure. What does FIFO stand for?

The action of inserting an object into a queue is usually called_____.

The action of removing an object from a queue is usually called _____.

Suppose we perform the following operations, in the order listed, on an initially empty queue Q .

- Insert an object A into Q .
- Insert an object B into Q .
- Remove an object from Q and print it.
- Insert an object C into Q .
- Insert an object D into Q .
- Remove an object from Q and print it.
- Remove an object from Q and print it.
- Remove an object from Q and print it.

In what order will the objects be printed?

2-4. Suppose we implement a queue using an array of and the algorithms described on page 2-5 in the class notes. The objects in the queue will be integers. Assume that we keep track of how many objects are in the queue by maintaining a `qsize` variable (this avoids having to leave the cell just to the left of `qfront` empty). Suppose the array has just 7 cells (numbered 0 through 6 , of course.) Suppose we perform the following operations on this queue (assumed to be empty initially):

Enqueue 29.
Enqueue 48.
Enqueue 14.
Enqueue 55.
Dequeue an integer and print it.
Enqueue 81.
Enqueue 34.
Dequeue an integer and print it.
Enqueue 70.
Enqueue 62.
Enqueue 99.
Dequeue an integer and print it.
Enqueue 25.
Dequeue an integer and print it.
Dequeue an integer and print it.
Enqueue 47.
Enqueue 18.
Dequeue an integer and print it. **[continued on the next page]**

(a) Will all the operations above be possible? Or will the queue start to overflow at some point?

(To answer the remaining questions, assume that if an attempt was made to enqueue an integer when the array was filled, the integer was discarded; that is, assume that the array was not enlarged.)

(b) List the integers printed by the operations above.

(c) Draw a picture of the array after the operations above have been performed. Show the position of the `qfront` and `qback` variables.

3. Measuring Algorithm Performance

When you write a program as an assignment in a class, you normally test it for correctness on small amounts of data, which usually means that the program will run to completion quickly, even if the algorithm used in the program is an inefficient one. By contrast, programs written for commercial or research purposes typically are required to process huge amounts of data or carry out lengthy computations. For a program of that kind, the amount of time and the amount of memory space that the program will require in order to process the data or perform its calculations is of significant concern. Generally, if we have competing algorithms, we'll prefer the one that takes less time, if there is a difference (e.g. binary search is far better than linear search for large ordered arrays). In the old days, memory space was also a big factor. In fact, it was often the case that considerable ingenuity had to be expended to create *any* algorithm that could successfully process large amounts of data in the very small memory spaces available on early machines. Today, time requirements for competing algorithms are generally what we're interested in, although memory space requirements will sometimes also require consideration. In this section of the notes we will look at ways of measuring and comparing algorithm running times.

Example 1: Let's begin with a simple example. The following fragment of C/C++ code sums the values stored in the cells of an array named `"a"` containing `n` floating point numbers. The algorithm involves a simple linear pass from left to right over the array.

```
float sum = 0.0;

for (int i = 0; i < n; ++i)
    sum += a[i];
```

Let T(n) denote the running time for this fragment of code. Is there anything definite that we can say about T(n) ?

If the code is executed on a machine with a slow clock speed, T(n) will be larger than if the code is executed on a fast clock machine. So T(n) depends on the processor (i.e., CPU) clock speed of the computer on which the code is executed. This means that we cannot express T(n) by some formula in n that gives the exact number of seconds.

If the code is executed on a machine with a rich instruction set, T(n) will probably be smaller than if the code is executed on a machine with a primitive instruction set. So even if two machines have the same processor clock speeds, T(n) will be different for the two machines.

If the code is compiled by two different C/C++ compilers on the same computer, the machine code produced by one compiler may be different from the machine code produced by the other, and this means that one of the two executables will run faster, no matter what the size n of data set may be. So T(n) can be different for two different machine code versions of the algorithm on the same machine.

Here is something definite we can say with certainty:

$$T(n) = An + B ,$$

where A and B are constants that depend on such things as processor clock speed, instruction sets, and the particular machine code produced by the compiler. That is, T(n) will be a *linear function* of the size n of the data set to be processed. This is the fact that is of interest to us when we write the code, because it has the following important consequence: on any particular machine and with any particular executable version of the code, *if we double the size of the data set to be processed, we will double (roughly) the running time*.

$$T(2n) = A(2n) + B \approx 2(An + B) = 2T(n) .$$

For large values of n , the constant B is insignificant. More generally, when we increase the size of the data set to be processed by a factor of k , the running time will also be increased by about the same factor k .

We can generalize this example by saying that any loop that makes a single pass over n data objects and performs a constant-time operation or set of operations on each object will require an amount of time of the form An + B , where the constants A and B depend on such things as processor speed, the richness of the processor's instruction set, and the way the code was translated into machine instructions.

Example 2: Let n denote the number of floating point values stored in a file. How much time will it take for the following algorithm to read the n data values, identify the largest, and print it out?

```
// Assume that "infile" is a non-empty file of floating point numbers
// in random order, and the file has just been opened for reading.

float datum, largest;

infile >> largest;  // Read the first number into "largest".
infile >> datum;    // Attempt to read a second number.
while (!infile.eof())  // If read was successful, compare datum and
{                      //                                   largest.
    if (datum > largest)
        largest = datum;

    infile >> datum;   // Attempt to read another number.
}

cout << "Largest value in the input file was " << largest << endl;
```

Let T(n) denote the running time for this algorithm on the file of n floating point numbers. Then the algorithm makes a single pass over the data and processes each data item exactly once. You might assume from what we said in Example 1 that the running time would be of the form An + B for some constants A and B . Note, however, that in this algorithm the processing time on a data item is not constant. The first item is read into a variable without any test being made on it. Each subsequent item is read into a variable, a comparison is made between it and the largest item seen so far, and then an assignment statement *may or may not* be executed. Thus the amount of time required for processing each item after the first is one of two different constants: the amount of time C_1 required to make the comparison and then skip the assignment statement, and the amount of time C_2 required to make the

comparison and then carry out the assignment statement. The constant C_1 will, of course, be smaller than the constant C_2. The running time $T(n)$ will then have the form

$$T(n) = C_0 + kC_1 + (n - 1 - k)C_2 + an + b,$$

where C_0 denotes the time required to process the first data item in the file,

k denotes the number of times the assignment statement is skipped,

$an + b$ accounts for the "loop overhead" involved in testing whether the file is empty and reading from the file.

The value of k depends on the arrangement of the floating point numbers in the file. The maximum possible value of k is $n - 1$, which occurs when the floating point numbers are in decreasing order in the file, so that the first is the largest. In this case $T(n) = C_0 + (n - 1)C_1 + an + b$. The minimum possible value of k is 0, which occurs when the floating point numbers are in increasing order in the file, so that each one is larger than all those that preceded it. In this case,
$T(n) = C_0 + (n - 1)C_2 + an + b$. Since C_1 is smaller than C_2, we have shown the following:

$$\min T(n) = C_0 + (n - 1)C_1 + an + b,$$
$$\max T(n) = C_0 + (n - 1)C_2 + an + b,$$
$$C_0 + (n - 1)C_1 + an + b \leq T(n) \leq C_0 + (n - 1)C_2 + an + b.$$

Grouping like terms, we find that

$$\min T(n) = A_1n + B_1,$$
$$\max T(n) = A_2n + B_2,$$
$$A_1n + B_1 \leq T(n) \leq A_2n + B_2,$$

where $A_1 = (C_1 + a)$ and $B_1 = (C_0 - C_1 + b)$, and similarly for A_2. What we have discovered is that the running time $T(n)$ for the algorithm is bounded below and above by linear functions of n, and thus we can be justified in saying that the running time is -- roughly speaking -- a *linear function* of the size of the data set to be processed. Even less exactly, we can think of the running time as being proportional to n when n is large (when n is large we ignore the constants B_1 and B_2).

We can generalize this example by saying that any loop that makes a single pass over n data objects and performs a set of operations on each object that requires at least an amount of time C_1 and at most an amount of time C_2, where C_1 and C_2 are constants (they do not depend in any way on n), then the running time $T(n)$ for the loop satisfies these conditions:

$$\min T(n) = A_1n + B_1,$$
$$\max T(n) = A_2n + B_2,$$
$$A_1n + B_1 \leq T(n) \leq A_2n + B_2,$$

where the constants A_i and B_i depend on such things as processor speed, the richness of the processor's instruction set, and the way the code was translated into machine instructions. When n is large, the B constants are negligible and we can assume that doubling the amount of data to be processed will approximately double the running time of the algorithm.

Example 3 (Selection Sort): The following function implements an algorithm known as Selection Sort on a array of objects of some type "`otype`" on which the comparison operator > is defined. It works by making a pass over the array `a[0...n-1]` to find the location of the largest object, swapping that object into the last position (cell `a[n-1]`), and then repeating these two operations on the unsorted subarray `a[0...n-2]`, and then on the unsorted subarray `a[0...n-3]`, and so on until we have reduced the "unsorted" subarray to size 1.

```
template <class otype>   // "otype" is short for "object type"
void selectionSort (otype a[], int n) // n = #objects in array a
{
    // For each cell a[k] from the last to the next-to-first:
    for (int k = n - 1;  k > 0;  --k)
    {
        // Find the location of the largest item in a[0...k]
        int bestLocation = 0;

        for (int j = 1; j <= k; ++j)
            if (a[j] > a[bestLocation])
                bestLocation = j;

        // Swap the largest item of a[0...k] into cell a[k]
        swap (a[bestLocation], a[k]);  // Code not shown for "swap"
    }
} // selectionSort
```

Let T(n) denote the running time of this function on any given array. Let's ask what we can say about T(n). We'll break T(n) down into pieces: the time required by the body of the outer loop when k has the value n – 1, plus the time required by the body of the outer loop when k has the value n – 2, and so on.

	minimum time required by the body of the outer loop	maximum time required by the body of the outer loop
1-st execution of the body of the outer loop; k = n – 1:	$A_1(n-1) + B_1$	$A_2(n-1) + B_2$
2-nd execution of the body of the outer loop; k = n – 2:	$A_1(n-2) + B_1$	$A_2(n-2) + B_2$
i-th execution of the body of the outer loop; k = n – i :	$A_1(n-i) + B_1$	$A_2(n-i) + B_2$
(n–1)-st execution of the body of outer loop; k = 1:	$A_1 + B_1$	$A_2 + B_2$

The minimum amount of time required for the function to execute is

$$A_1(n-1) + B_1 + A_1(n-2) + B_1 + A_1(n-3) + B_1 + \ldots + A_1(1) + B_1 + (an + b)$$
$$= A_1[(n-1) + (n-2) + (n-3) + \ldots + 1] + B_1(n-1) + (an + b),$$

where the $(an + b)$ term is there to account for "loop overhead" of the outer loop, plus some constant times associated with setting up the function call and getting it started executing, plus the time required for the return to take place.

The computations in the preceding paragraph lead us to a sum of the form

$$1 + 2 + 3 + \ldots + L$$

where L is some positive integer. We can simplify the sum above by using an algebraic trick. Write

$$S(L) = 1 + 2 + 3 + \ldots + (L-2) + (L-1) + L .$$

Reordering the terms gives $S(L) = L + (L-1) + (L-2) + \ldots + 3 + 2 + 1$.

Adding the expressions gives $2S(L) = (L+1) + (L+1) + (L+1) + \ldots + (L+1) + (L+1) + (L+1)$.

There are L $(L+1)$'s in the sum above, so $2S(L) = L(L+1)$, which gives $S(L) = \dfrac{L(L+1)}{2}$.

That is,

$$\boxed{1 + 2 + 3 + \ldots + (L-2) + (L-1) + L = \frac{L(L+1)}{2}}.$$

Taking L to be $n - 1$ in the paragraph at the top of this page, we see that the minimum amount of time required for the nested loops in Example 3 to execute is

$$A_1 \frac{(n-1)n}{2} + B_1(n-1) + an + b,$$

which can be rewritten in the form

$$\min T(n) = A_1^* n^2 + B_1^* n + C_1^* .$$

A similar computation shows that

$$\max T(n) = A_2^* n^2 + B_2^* n + C_2^* .$$

It follows that

$$A_1^* n^2 + B_1^* n + C_1^* \le T(n) \le A_2^* n^2 + B_2^* n + C_2^* .$$

What we have discovered is that the running time $T(n)$ for the Selection Sort algorithm is bounded below and above by quadratic functions of n. When n is large, the terms involving n^2 will be the dominant ones; that is, their size will swamp the size of the first degree and constant terms, so when n is large we can think of the running time $T(n)$ is being roughly proportional to n^2. If we double the size of the array to be sorted, we can expect that the running time will be increased by a factor of 4. If we triple the size of the array to be sorted, we can expect that the running time will be increased by a factor of 9. We see that the running time for this algorithm grows very rapidly with the amount of data to be processed. Later we will see some faster sorting algorithms.

Note: you should memorize the formula in the box above and be able to use it to simplify sums such as $1 + 2 + 3 + \ldots + k + (k+1)$. Taking $L = k + 1$ we get $\frac{(k + 1)(k + 2)}{2}$.

We can generalize Example 3 as follows. Suppose we have any loop that executes n times (or perhaps $n - 1$ times, or maybe $n + 1$ times; the exact number should depend in a linear way on the size n of the data set to be processed by the loop). Suppose further that the body of the loop, on the i-th time it is executed, requires at least an amount of time that's a linear function of i and n and at most an amount of time that's a (different) linear function of i and n. Then the running time $T(n)$ for the loop satisfies these conditions:

$$\min T(n) = A_1n^2 + B_1n + C_1 ,$$

$$\max T(n) = A_2n^2 + B_2n + C_2 ,$$

$$A_1n^2 + B_1n + C_1 \leq T(n) \leq A_2n^2 + B_2n + C_2 ,$$

where the constants A_i, B_i, and C_i depend on such things as processor speed, the richness of the processor's instruction set, and the way the code was translated into machine instructions.

Here are some other examples of the kinds of loops to which the above generalization applies.

```
for (int k = 1; k <= n; ++k)
  {
    do some initialization for the next loop;
    for (int j = 1; j < k; ++j)
      {
        loop body requiring time bounded below and above by constants
      }
    perform some post-loop operations requiring constant time;
  }
```

```
for (int k = 1; k <= n; ++k)
  {
    do some initialization for the next loop;
    for (int j = n; j >= k; --j)
      {
        loop body requiring time bounded below and above by constants
      }
    perform some post-loop operations requiring constant time;
  }
```

```
for (int k = 1; k <= n; ++k)
{
    do some initialization for the next loop;
    for (int j = 0; j <= n; ++j)
    {
        loop body requiring time bounded below and above by constants
    }
    perform some post-loop operations requiring constant time;
}
```

Here is a situation to which the generalization given above does *not* apply.

```
for (int k = 1; k <= n; ++k)
    loop body requiring time bounded below and above by constants

for (int j = 0; j <= n; ++j)
    loop body requiring time bounded below and above by constants
```

Example 4 (Linear Search): Here is a function that performs a linear search on a subarray `a[first...last]` for a prescribed target object. The objects in the array are of some data type on which the non-equality operator `!=` is defined. No assumption is made about the way in which the objects are arranged. The function returns the subscript of the leftmost cell containing a copy of `target`. If no cell contains such a copy, the function returns the value `last + 1` as a sentinel value to indicate that the search was unsuccessful.

```
template <class otype>
int locationByLinearSearch (const otype a[], const otype & target,
                            int first, int last)
{
    int k = first;

    while (k <= last && a[k] != target)
        ++k;

    // Assert:  when the loop above terminates, one of two things is
    // true:  (1) the subscript  k  has reached the value  last + 1
    // or     (2) a[k] is equal to "target".

    return k;  // In either case, we return the current value of  k .
} // locationByLinearSearch
```

Let n denote the number of cells in the subarray `a[first...last]`. Let T(n) denote the running time of the linear search function shown above. What can we say about T(n) ?

To answer this question fully, it is best to consider two separate cases: a search may be successful or unsuccessful. (Of course, until a search is made we don't know which case will occur.) We can let S(n) denote the running time when the search is successful, and we can let U(n) denote the running time when the search is unsuccessful. Then T(n) will be S(n) whenever the search is successful, and T(n) will be U(n) whenever the search is unsuccessful.

Example 5 (Binary Search): Here is a function that performs a search on a subarray `a[first..last]` containing objects of some data type on which the comparison operators < and > are defined. The objects in the array are assumed to be in *increasing order*, and the search is a *binary search*. The function uses two reference parameters to return a boolean value telling whether the search was successful, and the subscript of a cell containing a copy of `target`, provided the search is successful. If the search fails, the function returns a subscript, call it `k`, such that `a[k-1]` ≤ `target` ≤ `a[k]`. That is, in this case, k gives the location of the cell into which we could insert `target`, provided we shifted each of the objects in the cells `a[k...last]` to the right by one cell; the expanded subarray would still be in increasing order.

```
template <class otype>
void binarySearch (const otype a[], const otype & target, int first,
                   int last, bool & found, int & subscript)
{
    int mid;

    found = false;                      // The target hasn't been found.

    while (first <= last && !found)     // The value parameters "first"
    {                                   // and "last" are modified
        mid = (first + last)/2;         // during loop execution.

        if (target < a[mid])
            last = mid - 1;

        else if (a[mid] < target)
            first = mid + 1;

        else  // only remaining possibility:  a[mid] matches target
            found = true;
    }

    if (found)
        subscript = mid;    // The location of "target".
    else
        subscript = first;  // This is the appropriate subscript to
} // binarySearch           // return if "target" is not present.
```

Binary search works because an array is a **random access structure**, by which we mean that (for all practical purposes) the amount of time required to access any particular cell is the same as the time required to access any other cell. Thus we can "jump" from one cell to a distant cell during the search without having to access all the cells in between. Compare this with the situation we have in a linked list, where we cannot get to the middle node without accesses all preceding nodes in order.

Let's analyze the execution time for binary search. Let n denote the number of objects in the subarray `a[first...last]`. As with the linear search, we can consider separately two cases: the search is successful, in which case we will let S(n) denote the running time, and the case where the search is unsuccessful, in which case we will let U(n) denote the running time. To find a formula for U(n), we must count the number of times that the inner and outer loops will be executed. Let's look at this for n = 1, 2, 3, 4, . . . (see table on the following page).

	minimum # of times the loop body will be executed during an unsuccessful search min (n)	maximum # of times the loop body will be executed during an unsuccessful search max (n)	$\log_2(n)$
n = 1:			0
n = 2:			1
n = 3:			1. 6 (appr.)
n = 4:			2
n = 5:			2.3 (appr.)
n = 6:			2.6 (appr.)
n = 7:			2.8 (appr.)
n = 8:			3

Our computations above and in Exercise 3-18 (page 3-18) suggest that

$$\log_2(n) - 1 \ < \ \min(n) \ \leq \ \max(n) \ \leq \ \log_2(n) + 1 \quad \text{for all positive integers } n .$$

In fact, it is possible to give a mathematical proof of the inequalitites above (the proof will be given in CSCI C455). It follows that

$$\min U(n) = A_1 \log_2(n) + B_1 ,$$

$$\max U(n) = A_2 \log_2(n) + B_2 .$$

Next let's look at $S(n)$. The maximum number of times the loop body will execute during a successful binary search is the same as the maximum number for an unsuccessful search: the only difference is that on the very last try the target is found. Thus $\max S(n) = A_2 \log_2(n) + B_2$. What is the minimum number of times the loop body can execute during a successful search?

If we simply disregard the question of whether the binary search is successful or not and ask for information about the execution time $T(n)$ of the binary search algorithm on a subarray of length n , then the following is as much as we can say:

$$C \ \leq \ T(n) \ \leq \ A \log_2(n) + B .$$

The following rule generalizes what we have learned from studying binary search.

> Suppose that each time the body of a loop is executed, the value of some integer quantity Q associated with the loop is cut in half (or approximately in half), and the loop terminates when this quantity Q is reduced to 1 or 0 . Let Q_0 denote the initial value of the quantity Q .
> Then the body of the loop will be executed approximately $\mathbf{\log_2(Q_0)}$ times.

3.1. Logarithm Review

$\log_{10}(x)$ is defined to be the exponent on 10 that produces x .

Examples: $\log_{10}(100) = 2$ because 2 is the exponent you put on 10 to produce 100 .

$\log_{10}(1{,}000{,}000) =$ _____ because ___ is the exponent you put on 10 to produce 1,000,000 .

$\log_{10}(10) =$ _____ because ___ is the exponent you put on 10 to produce 10 .

$\log_{10}(1) =$ _____ because ___ is the exponent you put on 10 to produce 1 .

$\log_{10}(5) =$ _____ because ___ is the exponent you put on 10 to produce 5 .

$\log_{10}(10^p) =$ _____ because ___ is the exponent you put on 10 to produce 10^p

$10^{\log_{10}(x)} =$ _____ because _____ is the exponent you put on 10 to produce ___

$\log_{10}(xy) =$ _____

$\log_{10}(x/y) =$ _____

$\log_{10}(x^p) =$ _____

$\log_2(x)$ is defined to be the exponent on 2 that produces x .

Examples: $\log_2(8) = 3$ because 3 is the exponent you put on 2 to produce 8 .

$\log_2(32) =$ _____ because ___ is the exponent you put on 2 to produce 32 .

$\log_2(1024) =$ _____ because ___ is the exponent you put on 2 to produce 1024 .

$\log_2(2) =$ _____ because ___ is the exponent you put on 2 to produce 2 .

$\log_2(1) =$ _____ because ___ is the exponent you put on 2 to produce 1 .

$\log_2(11) =$ _____ because ___ is the exponent you put on 2 to produce 11 .

$\log_2(2^p) =$ _____ because ___ is the exponent you put on 2 to produce 2^p .

$2^{\log_2(x)} =$ _____

$\log_2(xy) =$ _____

$\log_2(x/y) =$ _____

$\log_2(x^p) =$ _____

Fact: $\log_2(x) \approx 3.32 \log_{10}(x)$ for all $x > 0$

More generally, all logarithm functions are constant multiples of each other.

THE MOST IMPORTANT FACT: log(x) grows extremely slowly, much slower than any linear function of x; slower even than $\sqrt{x}$.

3.2. Comparisons of Growth Rates

Suppose you have a 150 MHz (megaherz) PC on which you want to test six different algorithms that solve a particular data processing problem. [The term "150 MHz" means is that the CPU (central processing unit, or "processor" for short) is being driven by an electronic "clock" that produces 150 million cycles per second]. Suppose that on this PC, each machine instruction executed by the processor requires, on the average, 10 clock cycles. Then it is easy to see that this PC can execute approximately 15 million machine instructions per second.

Algorithm #	Approximate number of machine instructions required to solve the data processing problem when N items must be processed	Approx. Time Required for Problem of Size (below)				Approximate problem size that can be solved in
		N = 10	N = 100	N = 1,000	N = 1,000,000	1 minute
1	850 $\log_2$ (N)	.00019 sec	.00038 sec	.00056 sec	.00113 sec	$2^{1058823}$
2	300 N	.00020 sec	.00200 sec	.02000 sec	20 sec	3,000,000
3	90 N $\log_2$ (N)	.00020 sec	.00399 sec	.05979 sec	1.99 min	500,000+
4	30 N^2	.00020 sec	.02000 sec	2 sec	23 days	5,477
5	6 N^3	.00040 sec	.40000 sec	7 min	2×10^{11} yrs	531
6	3 (2^N)	.00021 sec	8×10^{15} yrs	-----	-----	28

3.3. Notation: Big Oh, Big Theta, Big Omega, Little Oh

When we were computing formulas for the running times of various algorithms, we expressed them in such forms as $T(n) = A_1n^2 + B_1n + C_1$. All those constants A_1 , A_2 , B_1 , etc. are just a nuisance. We are now going to learn a notation for running times that

- ignores constant factors (i.e., coefficients), and
- ignores lower order terms, i.e., the terms that can be neglected when n is large.

Definition: Let T(n) and F(n) be non-negative functions defined for all positive integers n . Then we write

$$T(n) = O(F(n))$$ (read " T(n) is **big-oh** of F(n) ")

if and only if there exists a constant c such that $T(n) \leq c\,F(n)$ for all sufficiently large n . The = sign in the expression is not used in the literal sense of equality of two numeric expressions, but rather as an abbreviation for the word "is".

Examples:

If $T(n) \leq An + B$ for some positive constant A and any constant B , then $T(n) = O(\_\_\_\_\_)$.

If $T(n) \leq An^2 + Bn + C$ for some positive constant A and any constants B and C , then $T(n) = O(\_\_\_\_\_)$.

If $T(n) \leq An^3 + Bn^2 + Cn + D$ for some positive constant A and any constants B , C , and D, then $T(n) = O(\_\_\_\_\_)$.

If $T(n) \leq A \log_2(n) + B$ for some positive constant A and any constant B , then $T(n) = O(\_\_\_\_\_)$.

If $T(n) \leq A2^n + Bn^2$ for some positive constant A and any constant B , then $T(n) = O(\_\_\_\_\_)$.

If $T(n) \leq C$ for some constant C , then $T(n) = O(\_\_\_\_\_)$.

If $T(n) \leq A \log_2(n) + Bn$ for some positive constants A and B , then $T(n) = O(\_\_\_\_\_)$.

Common functions arranged in increasing order of growth rate:

$1 \quad \log(n) \quad [\log(n)]^2 \quad \sqrt{n} \quad n \quad n\log(n) \quad n\sqrt{n} \quad n^2 \quad n^2\log(n) \quad n^2\sqrt{n} \quad n^3 \quad n^4 \quad 2^n \quad n!$

Definition: Let T(n) and F(n) be non-negative functions defined for all positive integers n . Then we write

$$T(n) = \Omega(F(n))$$ (read " T(n) is **big-omega** of F(n) ")

if and only if there exists a constant c such that $T(n) \geq c\,F(n)$ for all sufficiently large n .

Examples:

If $T(n) \geq An + B$ for some positive constant A and any constant B, then $T(n) = \Omega(\_\_\_\_\_)$.

If $T(n) \geq An^2 + Bn + C$ for some positive constant A and any constants B and C, then $T(n) = \Omega(\_\_\_\_\_)$.

If $T(n) \geq An^3 + Bn^2 + Cn + D$ for some positive constant A and any constants B, C, and D, then $T(n) = \Omega(\_\_\_\_\_)$.

If $T(n) \geq A \log_2(n) + B$ for some positive constant A and any constant B, then $T(n) = \Omega(\_\_\_\_\_)$.

If $T(n) \geq A2^n + Bn^2$ for some positive constant A and any constant B, then $T(n) = \Omega(\_\_\_\_\_)$.

If $T(n) \geq C$ for some constant C, then $T(n) = \Omega(\_\_\_\_\_)$.

If $T(n) \geq A \log_2(n) + Bn$ for some positive constants A and B, then $T(n) = \Omega(\_\_\_\_\_)$.

Definition: Let $T(n)$ and $F(n)$ be non-negative functions defined for all positive integers n. Then we write

$$T(n) = \Theta(F(n))$$ (read " $T(n)$ is **big-theta** of $F(n)$ ")

if and only if there exist positive constants c_1 and c_2 such that $c_1 F(n) \leq T(n) \leq c_2 F(n)$ for all sufficiently large n. Equivalently, $T(n)$ is big-theta of $F(n)$ if and only if $T(n)$ is both big-oh of $F(n)$ and big-omega of $F(n)$.

Examples:

(1) In Example 2 involving finding the largest number in a file of n numbers, the running time $T(n)$ was found to satisfy the inequalities $A_1n + B_1 \leq T(n) \leq A_2n + B_2$. It follows that $T(n) = \Theta(n)$.

(2) In Example 3 where we analyzed Selection Sort on an array of size n, the running time $T(n)$ was found to satisfy the inequalities $A_1n^2 + B_1n + C_1 \leq T(n) \leq A_2n^2 + B_2 + C_2$. It follows that $T(n) = \Theta(n^2)$.

(3) In Example 4 where we analyzed linear search in an array of size n, the running time when the search is unsuccessful is $\Theta(n)$, while the running time for a successful search is $O(n)$ and $\Omega(1)$.

(4) In Example 5 where we analyzed binary search in an array of size n, the running time for an unsuccessful search is $\Theta(\log(n))$, while the running time for a successful search is $\Omega(1)$ and $O(\log(n))$.

There is one more definition that is usually given in connection with this material. Your text's version of the definition is incorrect. Here is the correct definition.

Definition: Let $T(n)$ and $F(n)$ be non-negative functions defined for all positive integers n. Then we write

$$T(n) = o(F(n))$$ (read " $T(n)$ is **little-oh** of $F(n)$ ")

if and only if $\lim_{n \to \infty} \frac{T(n)}{F(n)} = 0$. Equivalently, $T(n)$ is little-oh of $F(n)$ if and only if for every positive number ε there exists an integer n_ε such that for all $n \geq n_\varepsilon$ we have $T(n) \leq \varepsilon F(n)$.

Written Exercises

3-1. Consider the following code fragment in which the variables `j` and `k` are assumed to have been declared as integers. Also assume that when the code fragment begins execution, the variable `k` has a positive value (i.e., is 1 or greater).

```
queueOf<int> q;          // Create an empty queue named  q .
for (j = 0; j <= k; ++j)
    if (j % 7 == 0)        // If  j  is divisible by  7
        q += j;              // then enqueue  j  into  q .
```

Let T(k) denote the execution time that will be observed when this code fragment is executed. Which of the following statements ***best*** describes T(k) ?

(a) $T(k) = Ak + B$ for some constants A and B .

(b) $A_1k + B_1 \le T(k) \le A_2k + B_2$ for some constants A_1 , B_1 , A_2 , and B_2 .

(c) $T(k) = Ak^2 + Bk + C$ for some constants A, B, and C .

3-3. Consider the following code fragment in which the variables `j, k,` and `count` are assumed to have been declared as integers. Also assume that when the code fragment begins execution, the variable `n` has a positive value (i.e., is 1 or greater).

```
count = 0;
for (j = 0; j <= k; ++j)
    ++count;
```

Let T(k) denote the execution time that will be observed when this code fragment is executed. Which of the following statements ***best*** describes T(k) ?

(a) $T(k) = Ak + B$ for some constants A and B .

(b) $A_1k + B_1 \le T(k) \le A_2k + B_2$ for some constants A_1 , B_1 , A_2 , and B_2 .

(c) $T(k) = Ak^2 + Bk + C$ for some constants A, B, and C .

3-5. Answer the following questions about the code fragment in problem 3-3 above.

(a) How many times will the assignment `j = 0` be executed?

(b) How many times will the inequality `j <= k` be tested? (Express your answer in terms of the value of `k` , i.e., as some function of `k` .)

(c) How many times will the increment operation `++j` be executed? (Answer in terms of `k` .)

(d) How many times will the increment operation `++count` be executed?

3-6. Consider the following code fragment in which the variables `m` and `n` are assumed to have been declared as integers. Also assume that when the code fragment begins execution, the variable `n` has a positive value (i.e., is 1 or greater).

```
stackOf<int> s;          // Create an empty stack named  s .
for (m = 0; m < n; ++m)
    s += m * m;    // Push the square of  m  onto the stack  s
.
```

Let T(n) denote the execution time that will be observed when this code fragment is executed. Which of the following statements ***best*** describes T(n) ?

a. $T(n) = An + B$ for some constants A and B .

b. $A_1n + B_1 \le T(n) \le A_2n + B_2$ for some constants A_1 , B_1 , A_2 , and B_2 .

c. $T(n) = An^2 + Bn + C$ for some constants A, B, and C .

3-8. Consider the following code fragment in which the variables `m` and `n` are assumed to have been declared as integers, and the array `a` has at least `n` cells that have been given various integer values.

```
for (m = n-1; m >= 0; --m)
    if (a[m] < 0)         // If  a[m]  is negative, then change
        a[m] *= (-1);     // its sign by multiplying it by -1 .
```

Let T(n) denote the execution time that will be observed when this code fragment is executed. Which of the following statements ***best*** describes T(n) ?

a. $T(n) = An + B$ for some constants A and B .

b. $A_1n + B_1 \le T(n) \le A_2n + B_2$ for some constants A_1 , B_1 , A_2 , and B_2 .

c. $T(n) = An^2 + Bn + C$ for some constants A, B, and C .

3-10. Answer the following questions about the code fragment in problem 3-6.

a. How many times will the assignment `m = 0` be executed?

b. How many times will the inequality `m < n` be tested? (Express your answer in terms of the value of `n` , i.e., as some function of `n` .)

c. How many times will the increment operation `++m` be executed? (Answer in terms of `n` .)

d. How many times will the push operation `s += m * m` be executed?

3-13. Consider the following code fragment in which the variables `k`, `j`, `n`, and `count` can be assumed to have been declared as integers. Also assume that when the code fragment begins execution, the variable `n` has a positive value.

```
count = 0;
for (k = n; k >= 0; --k)
    for (j = 0; j <= k; ++j)
        ++count;
```

The "outer loop" is the "`for`" loop controlled by the loop index `k`. The "body of the outer loop" consists, in this example, of another loop, the "inner" (or "nested") loop controlled by the loop index `j`. The "body of the inner loop" in this example is just the statement "`++count;`".

(a) How many times will the body of the inner loop be executed on the first time that the body of the outer loop is executed (i.e., when `k` is `n`)? (Express your answer in terms of `n`.)

(b) How many times will the body of the inner loop be executed on the second time that the body of the outer loop is executed (when `k` is fixed at `n-1`)? (Express your answer in terms of `n`.)

(c) How many times will the body of the inner loop be executed on the last time that the body of the outer loop is executed (when `k` is fixed at `0`)?

(d) What will the value of `count` be when this code fragment has completed execution? (Express your answer as an exact polynomial in `n`.) Note, incidentally, that the final value of `count` is exactly the same as the total number of times the body of the inner loop is executed.

3-14. Consider the following code fragment in which the variables `i, j,` and `k` have been declared as integers, `duplicateFound` has been declared to be boolean, and `a` has been declared to be an integer array with at least `k+1` cells.

```
duplicateFound = false;
for (j = 1; j < k; ++j)
    for (i = k; i > j; --i)
        if (a[i] == a[j])
            duplicateFound = true;
```

The "outer loop" is the "`for`" loop controlled by the loop index `j`. The "body of the outer loop" consists, in this example, of another loop, the "inner" (or "nested") loop controlled by the loop index `i`. The "body of the inner loop" in this example is an "`if`" statement.

(a) How many times will the body of the inner loop be executed on the first time that the body of the outer loop is executed (i.e., when `j` has the value `1`)? (Express your answer in terms of `k`.)

(b) How many times will the body of the inner loop be executed on the second time that the body of the outer loop is executed (when `j` has the value `2`)? (Express your answer in terms of `k`.)

(c) How many times will the body of the inner loop be executed on the last time that the body of the outer loop is executed (when `j` has the value `k-1`)?

(d) How many times altogether will the test expression `a[i] == a[j]` be evaluated? Express your answer as an exact polynomial in `k`.

3-16. Why is the following version of binary search bad? (Note: this version differs from the version on page 3-9 only in the body of the while loop.)

```
template <class otype>
void binarySearch (const otype a[], const otype & target, int first,
                   int last, bool & found, int & subscript)
{
    int mid;

    found = false;                      // The target hasn't been found.

    while (first <= last && !found)     // The value parameters "first"
    {                                   // and "last" are modified
        mid = (first + last)/2;         // during loop execution.

        if (target < a[mid])
            last = mid - 1;

        if (a[mid] < target)
            first = mid + 1;

        if (a[mid] == target)
            found = true;
    }

    if (found)
        subscript = mid;    // The location of "target".
    else
        subscript = first;  // This is the appropriate subscript to
} // binarySearch           // return if "target" is not present.
```

3-18. Work out the eight numbers that should appear on page 3-10 under the heading `min(n)`. Do your computations confirm that $\log_2(n) - 1 < \min(n)$ for at least these eight integers n ?

3-20. How many times (approximately) will the body of the following loop be executed?

```
for (int k = n; k > 1; k = k/2)
    cout << k << endl;
```

Express your answer in terms of the variable `n`. Also, show what the output will be if `n` is 45 .

3-21. Without using a pocket calculator, find an integer n such that $\log_2(200)$ lies between n and n+1 . Explain how you obtained your answer.

3-22. Without using a pocket calculator, find an integer n such that $\log_2(5000)$ lies between n and n+1 . Explain how you obtained your answer.

3-23. Without using a calculator, state exactly the value of the expression $2^{\log_2(809)}$.
[HINT: if you fully understand the concept of "$\log_2$ ", then this is a *very* easy question.]

3-24. Without using a calculator, state exactly the value of the expression $2^{\log_2(95)}$.

3-27. Without using a calculator, state exactly the value of the expression $\log_2(2^{7.654})$.

3-28. Without using a calculator, state exactly the value of the expression $\log_2(2^{0.12})$.

3-33. Suppose $T_1(n)$ and $T_2(n)$ are non-negative functions such that $T_1(n) = O(n^2)$ and $T_2(n) = O(n^2)$. Then the quantity $T_1(n) + T_2(n)$ must also be big-oh of n^2. Here is a proof: by definition, we know that there exist positive constants C_1 and C_2 such that $T_1(n) \le C_1 n^2$ and $T_2(n) \le C_2 n^2$ for all large n. It follows that $T_1(n) + T_2(n) \le C_1 n^2 + C_2 n^2 = (C_1 + C_2)n^2$, for all large n. Let C denote $C_1 + C_2$. Then C must be positive, and we have shown that $T_1(n) + T_2(n) \le Cn^2$ for all large n. This proves that $T_1(n) + T_2(n) = O(n^2)$.

(a) Give a similar proof that the product quantity $T_1(n)T_2(n) = O(n^4)$.

(b) Is it necessarily true that $T_1(n) = O(n^3)$? Explain.

(c) Is it necessarily true that $T_1(n) = \Omega(n^3)$. Explain.

(d) Is it necessarily true that $T_1(n) = O(n)$? Explain.

(e) Is it necessarily true that $T_1(n) = \Omega(n)$. Explain.

3-34. Suppose $T_1(n)$ and $T_2(n)$ are non-negative functions such that $T_1(n) = \Omega(n)$ and $T_2(n) = \Omega(n \log(n))$.

(a) Prove that $T_1(n)\, T_2(n) = \Omega(n^2 \log(n))$.

(b) Is it necessarily true that $T_2(n) = O(n \log(n))$? Explain your answer.

(c) Is it necessarily true that $T_2(n) = \Omega(n)$? Explain your answer.

(d) Is it necessarily true that $T_2(n) = \Omega(n^2)$? Explain your answer.

3-37. Suppose $T(n)$ is a non-negative function defined for all positive integers n.

(a) If $T(n) \le 3\sqrt{n} + 5 \log_2(n)$, then $T(n) = O(\ ?\ )$.

(b) If $T(n) \le 7n^2 + 2^n + 100n^2 \log_2(n)$, then $T(n) = O(\ ?\ )$.

(c) If $T(n) \le 8n \log_2(n) + 50n\sqrt{n}$, then $T(n) = \Omega(\ ?\ )$.

(d) If $T(n) \le 3(n!) + 7(2^n) + 60n^3$, then $T(n) = \Omega(\ ?\ )$.

(e) If $T(n) = An \log_2(n) + Bn^2 + C(\log_2(n))^2$, where A, B, and C are positive constants, then $T(n) = \Theta(\ ?\ )$.

(f) If $T(n) = A2^n + Bn^4 + C\sqrt{n} \log_2(n)$, where A, B, and C are positive constants, then $T(n) = \Theta(\ ?\ )$.

3-38. Suppose $E(n)$ is a non-negative function defined for all positive integers n.

(a) If $E(n) \le 5n^2 + 9n^3$, then $E(n) = O(\ ?\ )$.

(b) If $E(n) \le 8n\sqrt{n} + 100n \log_2(n)$, then $E(n) = O(\ ?\ )$.

(c) If $E(n) \le 2 \log_2(n) + \sqrt{n}$, then $E(n) = \Omega(\ ?\ )$.

(d) If $E(n) \le 4(n \log(n)) + 6(2^n) + n^3$, then $E(n) = \Omega(\ ?\ )$.

(e) If $E(n) = A \log_2(n) + Bn + C\sqrt{n}$, where A, B, and C are positive constants, then $E(n) = \Theta(\ ?\ )$.

(f) If $E(n) = An + Bn^{-1} + C\sqrt{n} \log_2(n)$, where A, B, and C are positive constants, then

$E(n) = \Theta(\ ?\ )$.

3-41. Given a choice between two algorithms, one of which requires $\Theta(n^2)$ time to process n data objects and the other of which requires $\Theta(n\sqrt{n})$ time to process n data objects, which would you prefer to use on a large data set? Why? (Assume both algorithms require the same amount of space.)

3-42. Given a choice between two algorithms, one of which requires $\Theta(n^2 \log(n))$ time to process n data objects and the other of which requires $\Theta(n^3)$ time to process n data objects, which would you prefer to use on a large data set? Why? (Assume both algorithms require the same amount of space.)

3-45. (a) Suppose analysis of an algorithm A shows that its running time on a data set of size n is $O(n^3)$. Suppose analysis of an algorithm B shows that its running time on a data set of size n is $O(n^4)$. Is it necessarily true that algorithm B will require more time than algorithm A on all large data sets?

(b) Suppose analysis of an algorithm C shows that its running time on a data set of size n is $\Omega(n^3)$. Suppose analysis of an algorithm D shows that its running time on a data set of size n is $\Omega(n^4)$. Is it necessarily true that algorithm D will require more time than algorithm C on all large data sets?

(c) Suppose analysis of an algorithm X shows that its running time on a data set of size n is $\Theta(n^3)$. Suppose analysis of an algorithm Y shows that its running time on a data set of size n is $\Theta(n^4)$. Is it necessarily true that algorithm Y will require more time than algorithm X on all large data sets?

3-47. Suppose we have an array of length M named "`list`". Suppose also that each cell of this array contains a pointer to an *ordered array* of length N . That is, `list` is an array of ordered arrays. (The word "list" does not always mean "linked list". You can keep a list of things in an array.)

The following code fragment makes a separate *binary search* (as coded in the class notes) of each of the M arrays pointed to by the pointers in the array `list` .

```
for (int i = 0; i < M; ++i)
{
    int         location;
    bool        succeeded;
    thingamabob target;

    binarySearch (list[i], target, 0, N-1, succeeded, location);
    if (succeeded)
        cout << "target found in list " < i << endl;
}
```

What is the maximum possible execution time for this code fragment? Express your answer in big-theta notation involving the variables M and N .

3-48. For each of the program fragments below, give a big-theta analysis of the running time in terms of the integer variable n .

(a)
```
for (int i = 0, sum = 0; i <= n; ++i)
    ++sum;
```

(b)
```
for (int i = 0, sum = 0; i <= n; ++i)
    for (int j = 0; j < n; ++j)
        ++sum;
```

(c)
```
for (int i = 0, sum = 0; i <= n; ++i)
    ++sum;
for (int j = 0; j < n; ++j)
    ++sum;
```

(d)
```
for (int i = 0, sum = 0; i <= n; ++i)
    for (int j = 1; j < i; ++j)
        ++sum;
```

(e)
```
for (int i = 0, sum = 0; i <= n * n; ++i)
    ++sum;
```

(f)
```
for (int i = 0, sum = 0; i * i <= n; ++i)
    ++sum;
```

(g)
```
for (int i = 0, sum = 0; i <= n * n; ++i)
    for (int j = 1; j < i; ++j)
        ++sum;
```

(h)
```
for (int i = 0, sum = 0; i <= n; ++i)
    for (int j = 1; j <= n * n; ++j)
        for (int k = 1; k <= j; ++k)
            ++sum;
```

(i)
```
for (int i = n, sum = 0; i >= 1; i = i/2)
    ++sum;
```

(j)
```
for (int i = n, sum = 0; i >= 1; i = i/2)
    for (int j = 1; j <= n; ++j)
        ++sum;
```

Solutions for Odd-Numbered Exercises

3-1. In this code fragment, the assignment statement `q += j` is sometimes executed and sometimes not, depending on the value of `j`. Thus the body of the loop does not require constant time; instead, sometimes it requires little time (only the amount of time to "jump over" the assignment statement), and sometimes it requires the time to perform the assignment statement. The running time is therefore not a strictly linear function of `k`, so answer "(a)" is not exactly correct (though it's not far off). A better answer is "(b)".

3-3. In this code fragment, the body of the loop requires constant time, and so the execution time is a strict linear function of `k`. Thus the best answer is "(a)".

3-5. (a) Just once, when the loop is first encountered.
(b) It will be tested $k+2$ times. The first $k+1$ times that it is tested (when $j = 0, 1, 2, \ldots, k$), it will evaluate to true. The last time it is tested (when $j = k+1$) it will evaluate to false.
(c) This operation is performed just after the body of the loop is executed; the body of the loop is executed as many times as the control condition (`j <= k`) is true, which is $k+1$.
(d) This is the body of the loop. As explained in part (c), this will be executed $k+1$ times.

3-13. (a) $n+1$ times (see the solution for problem 3-5(c))
(b) n times
(c) just once
(d) The final value of `count` is equal to the total number of times the body of the inner loop (which is just `++count;`) is executed. This will be $(n + 1) + n + (n - 1) + \ldots + 2 + 1$.
We can find a compact formula for this sum in any of several ways.
(1) Go back to first principles and derive a formula by adding the terms in the opposite order (as we did in the lecture). This gives $2(\text{the sum}) = (n+2) + (n+2) + \ldots + (n+2)$, where there are $n+1$ terms in the sum on the right side of the equation. Thus the desired answer is $\boxed{\frac{(n+1)(n+2)}{2}}$.

(2) We can use the formula $(n - 1) + (n - 2) + \ldots + 1 = \frac{n(n-1)}{2}$ in the following way:

$$(n+1) + n + \underline{(n-1) + \ldots + 2 + 1} = (n + 1) + n + \frac{n(n-1)}{2} = \frac{2(n+1) + 2n + n(n-1)}{2} = \boxed{\frac{(n+1)(n+2)}{2}}.$$

(3) We can use the formula in the first line of (2) above with n replaced by $n+2$ to get the same answer: $\frac{(n+2-1)(n+2)}{2} = \boxed{\frac{(n+1)(n+2)}{2}}$.

3-21. $2^3 = 8$, $2^4 = 16$, $2^5 = 32$, $2^6 = 64$, $2^7 = 128$, $2^8 = 256$, so $\log_2(128) = 7$, $\log_2(256) = 8$; it follows that $\log_2(200)$ must lie somewhere between 7 and 8.

3-23. 809, because $2^{\log_2(x)} = x$ for all positive numbers x.

3-27. 7.654 , because $\log_2(2^x) = x$ for all real numbers x .

3-33. (a) By definition, we know that there exist positive constants C_1 and C_2 such that $T_1(n) \leq C_1n^2$ and $T_2(n) \leq C_2n^2$ for all large n . It follows that $T_1(n)\, T_2(n) \leq (C_1n^2)\,(C_2n^2) = (C_1C_2)n^4$, for all large n . Let C denote C_1C_2 . Then C must be positive and we have shown that $T_1(n)T_2(n) \leq Cn^4$ for all large n . This proves that $T_1(n)T_2(n) = O(n^4)$.
Give a similar proof that the product quantity $T_1(n)T_2(n) = O(n^4)$.
(b) Is it necessarily true that $T_1(n) = O(n^3)$? Yes. If $T_1(n) \leq C_1n^2$ for all large n , then certainly we have $T_1(n) \leq C_1n^3$ for all large n , and thus $T_1(n) = O(n^3)$.
(c) Is it necessarily true that $T_1(n) = \Omega(n^3)$. No. Since $T_1(n) \leq C_1n^2$ for all large n , it cannot possibly be the case that there is some constant C such that $T_1(n) \leq Cn^3$ for all large n .
(d) Is it necessarily true that $T_1(n) = O(n)$? No. This requires that there exist a positive constant C such that $T_1(n) \leq Cn$ for all large n . While this *might* be true (the information we are given does not preclude it), it is not *necessarily* true. It may be the case that $T_1(n) = 5n^2$, in which case $T_1(n)$ would not be $O(n)$.
(e) Is it necessarily true that $T_1(n) = \Omega(n)$. No. For example, it *might* be true that $T_1(n) \geq 10$ for all large n , in which case there could not exist a positive constant C such that $T_1(n) \geq Cn$ for all large n .

3-37. (a) If $T(n) \leq 3\sqrt{n} + 5\log_2(n)$, then $T(n) = O(\sqrt{n})$.
(b) If $T(n) \leq 7n^2 + 2^n + 100n^2\log_2(n)$, then $T(n) = O(2^n)$.
(c) If $T(n) \leq 8n\log_2(n) + 50n\sqrt{n}$, then $T(n) = \Omega(n\sqrt{n})$.
(d) If $T(n) \leq 3(n!) + 7(2^n) + 60n^3$, then $T(n) = \Omega(n!)$.
(e) If $T(n) = An\log_2(n) + Bn^2 + C(\log_2(n))^2$, where A , B , and C are positive constants, then $T(n) = \Theta(n^2)$.
(f) If $T(n) = A2^n + Bn^4 + C\sqrt{n}\log_2(n)$, where A , B , and C are positive constants, then $T(n) = \Theta(2^n)$.

3-41. The function $n\sqrt{n}$ grows more slowly than n^2 when n is large, so the algorithm that requires $\Theta(n\sqrt{n})$ time to process n data objects is better for large data sets because it will require less time.

3-45. (a) No. The running time for algorithm B *might* actually be proportional to n^2 , while the running time for algorithm A *might* be proportional to n^3 , in which case B would require less time than A on large data sets.
(b) No. The running time for algorithm C *might* actually be proportional to n^5 , while the running time for algorithm D *might* be proportional to n^4 , in which case D would require less time than C on large data sets.

(c) Yes. The running time of X is approximately proportional to n^3 and the time for Y is approximately proportional to n^4, so Y will require more time than algorithm X on all large data sets?

3-47. The code given in this problem is searching a data structure that looks like this:

The body of the outer loop is executed M times (when i = 0, 1, 2, . . . , M-1). The body of the outer loop consists mostly of statements that require time bounded by smaller and larger constants. However, there is one statement that can require more time. The binary search function is called M times (once each time through the body of the outer loop). In the worst case, the binary search requires execution time proportional to the logarithm of the length of the array being searched. In the problem we are considering, each of those arrays is of length N, so the maximum possible execution time for each binary search is proportional to log(N). Since there are M such searches, the binary search times altogether require time proportional to M log(N)) *in the worst case*. The other parts of the code require time proportional to AM + B. Thus the total maximum execution time has the form AM + B + CM log(N). The dominant term here is M log(N), so the maximum execution time is Θ(M log(N)).

4. The "Table" ADT

One of the most common problems in programming involves maintaining a collection of objects in such a way that they can be accessed efficiently by specifying some "key" value that uniquely identifies the object. The key is assumed to be part of the object, i.e. *contained within* the object as a member or the object, or sometimes as a combination of members of the object. The most common keys are integers or character strings, but complex keys consisting of several different members are sometimes used.

As an example, consider the problem of maintaining the academic records of all the students who have enrolled at IUSB. Each student record would consist of the student's name and ID number, address, phone number, birthday, name of the high school from which the student graduated, and all the courses taken by the student, the semester in which each course was taken, and the course grade. The key for such a student record would be the ID number since two or more students may have exactly the same name, while the ID number uniquely identifies each student.

When a collection of data objects (such as student) records is created and maintained on a computer system, we call such a collection a "table", or perhaps a "table of keyed objects". Other names for this ADT are "dictionary", or "map of keys", or "associative container class". No one name seems to have become universal in the data structures world. We will use the term "table" for this ADT.

The primary operations on a table are the following:

(1) **Insert**: objects may be inserted into the table at any time and in any order; the insertion will fail if the table is full or if the table already contains an object having the same key as the object being inserted. *Thus it is always necessary, during an insertion operation, to make sure the table does not already contain an object whose key is identical with the key of the object we are attempting to insert*.

(2) **Access**: an object may be accessed in the table by specifying its key value; if the table contains an object with the specified key, then the access operation returns a copy of that object (or, depending on how the table is implemented, perhaps a pointer to the object); the operation will fail if the table contains no object having the specified key; *a successful access operation does not alter the contents of the table*.

(3) **Remove** (delete): an object may be removed from the table by specifying its key value; if the table contains an object with the specified key, then the removal operation returns a copy of that object (or possibly a pointer to the object); the operation will fail if the table contains no object having the specified key; if the operation is successful, the table will no longer contain an object with the specified key.

4-2

Other operations may or may not be defined on a table. Some common operations are the following:

(4) **Test for Empty**: the test returns some indication of whether the table is or is not empty.

(5) **Make Empty**: this operation discards all the objects in the table.

(6) **Traverse in Key Order**: this operation visits each of the objects in the table in increasing key order; this assumes that there is a well-defined ordering on the set of all possible keys (e.g., if the keys are character strings, the objects would be visited in alphabetical order).

(7) **Find Minimum**: this operation accesses (or perhaps removes) the object with the smallest key in the table; this assumes a well-defined ordering on the set of all possible keys.

(8) **Find Maximum**: this operation accesses (or perhaps removes) the object with the largest key in the table.

4.1. Table Implementation 1

Use an unordered array. That is, keep the objects in the table in no particular order. When n objects are in the table they can occupy cells 0, 1, . . . , n–1 . Use an integer variable, say "`tsize`", to keep track of the number of objects in the table. (Then `tsize - 1` will be the subscript of the cell containing the right-most object in the table; when `tsize` is zero, that will indicate that the table is empty.) If the array is dynamically allocated and it is desired to allow the table to expand indefinitely, use an integer variable, say "`tcap`", to keep track of the current capacity of the array.

To insert a copy of an object x :

Make a linear search of the objects already in the table to determine whether the table already has a key that matches the key of x .

If a match is discovered, report failure.

If `tsize` is equal to `tcap` , enlarge the array. If this is impossible, report failure.

Copy x into cell `tsize` and then increment `tsize` .

Report success.

To access an object with a specified key k:

Make a linear search of the objects in the table to try to find an object with key k .

If no such object is found, report failure.

Report success and return a copy of the unique object with key k .

To remove an object with a specified key k :

Make a linear search of the objects in the table to try to find an object with key k .

If no such object is found, report failure.

Copy the object with key k into a temporary location.

Overwrite the object whose key is k with the last (right-most) object in the table.

Decrement `tsize`.

Report success and return a copy of the object in the temporary location.

To test for empty:

Test whether `tsize == 0` .

To make the table empty:

Change the value of `tsize` to 0 . (There is no need to "clean out" the array cells.)

To traverse the table in key order:

THIS IS NOT A GOOD IMPLEMENTATION FOR THIS OPERATION.

To perform the Find Minimum operation (the version that does not change the table):

If the table is empty, report failure.

Use a standard linear search algorithm to find the object with smallest key.

Report success and return a copy of the object with smallest key.

The Find Maximum is similar.

Analysis of Table Implementation 1.

When n objects are in the table, then

- Successful insertion requires $\Theta(n)$ time; this is true even if the array must be expanded and the n objects in the table copied to the larger array. Unsuccessful insertion requires $\Theta(1)$ time in the best case (a duplicate key is found as soon as the search begins) and $\Theta(n)$ time in the worst case (a duplicate key is found in the n-th cell). Thus unsuccessful insertion is a $O(n)$ operation.
- Successful access requires $\Theta(1)$ time in the best case (the specified key is found immediagtely) and $\Theta(n)$ time in the worst case. Thus successful access is a $O(n)$ operation. Unsuccessful access requires $\Theta(n)$ time since the entire table must be searched.
- Removal times are identical with those for access because the removal operation involves an access followed by a few operations that are independent of n, i.e., that are $\Theta(1)$.
- Test for empty and making the table empty are $\Theta(1)$ operations.
- Find min and find max are $\Theta(n)$ operations if the table is non-empty, $\Theta(1)$ operations if the table is empty.

If the size n of a proposed table is expected to be *always small* (say 10 or less) and the Traverse in Order operation will not be needed, then Table Implementation 1 is a perfectly reasonable implementation: it is easy to code and the operations will all be efficient.

4.2. Table Implementation 2

Use an ordered array. That is, keep the objects in the table in increasing key order. When n objects are in the table they can occupy cells $0, 1, \ldots, n-1$. As before, use an integer variable, say "`tsize`", to keep track of the number of objects in the table. If the array is dynamically allocated and it is desired to allow the table to expand indefinitely, use an integer variable, say "`tcap`", to keep track of the current capacity of the array.

To insert a copy of an object x:

Make a *binary* search of the objects already in the table to determine whether the table already has a key that matches the key of x.
If a match is discovered, report failure.
Else the binary search should return an "insertion point".
If `tsize` is equal to `tcap`, enlarge the array. If this is impossible, report failure.
Shift right by one cell all objects in and to the right of the insertion point.
Copy x into the cell at the insertion point, and then increment `tsize`.
Report success.

To access an object with a specified key k:

Make a binary search of the objects in the table to try to find an object with key k.
If no such object is found, report failure.
Else report success and return a copy of the object with key k.

To remove an object with a specified key k :

Make a binary search of the objects in the table to try to find an object with key k .

If no such object is found, report failure .

Else copy the object with key k into a temporary location.

Shift left by one cell every object to the right of the one with key k .

Decrement `tsize`.

Report success and return a copy of the object in the temporary location.

To test for empty:

Test whether `tsize == 0` .

To make the table empty:

Change the value of `tsize` to 0 . There is no need to "clean out" the array cells, unless the objects contain dynamically allocated memory, in which case we might decide we want to destroy them by deallocating the array and getting a new one.

To traverse the table in key order:

Traverse the table from left to right.

To perform the Find Minimum operation (the version that does not change the table):

If the table is empty, report failure.

Else report success and return a copy of the object in cell 0 .

The Find Maximum is similar.

Analysis of Table Implementation 2.

When n objects are in the table, then

- Successful insertion begins with an unsuccessful binary search that requires $\Theta(\log(n))$ time; then, if the array is full it must be expanded, which requires $\Theta(n)$ time; then a shift operation must be performed that requires $\Theta(1)$ time in the best case (the object to be inserted belongs at the right end of the array) and $\Theta(n)$ time in the worst case (the object belongs in cell 0); finally the object must be copied into the vacated cell, which requires $\Theta(1)$ time. Thus successful insertion requires $\Theta(\log(n))$ time in the best case and $\Theta(n)$ time in the worst case. Unsuccessful insertion begins with a successful binary search or an unsuccessful binary search and a failure to expand the array; in the best case this takes $\Theta(1)$ time and in the worst case $\Theta(\log(n))$ time; reporting failure takes $\Theta(1)$ time.
- Successful access requires $\Theta(1)$ time in the best case (the specified key is found immediately) and $\Theta(\log(n))$ time in the worst case. Thus successful access is a $O(\log(n))$ operation. Unsuccessful access requires $\Theta(\log(n))$ time since the entire table must be searched.
- Removal times are identical with those for insertion because the removal operation involves an access followed by a left-ward shift of all the objects to the right of the one removed.
- Test for empty and making the table empty are $\Theta(1)$ operations.
- Find min and find max are $\Theta(1)$ operations.

4.3. Table Implementation 3

We get fast access times with an ordered array, but insertion and removal can be expensive operations (when cost is measured by execution time). The problem lies in the shifting that must take place. Insertion and removal in a *linked list*, however, do NOT require shifts of data. We simply "splice in" or "cut out" the node carrying the object to be inserted or removed. Thus it makes sense to look at the possibility of using an ordered linked list to implement a table. That is, we'll keep the objects in the nodes of a linked list, arranged in key order. All that this requires is a pointer, call it `front`, to the first node of the list. When the table is empty, we'll give `front` the value NULL.

To insert a copy of an object x:

> Make a *binary* search of the linked list to determine whether the table already contains an object with a key that matches the key of x. WAIT A MINUTE. We can't do that. There is no way to make a *binary* search of a linked list because it is not a random access structure. We have to make a linear search. Note, however, that we usually do not have to search the entire linked list. If the key of x is already present in the table, it is seldom in the last position. On average only half the list will have to be searched. Also, if the key of x is *not* present in the table, the search is likely to arrive eventually at a key larger than the key of x. At that point, the search can end and a new node can be inserted just in front of the node carrying the object with larger key. (If this is a singly linked list, then it will be helpful to advance two pointers along the list during the search, one behind the other, to make it easy to insert the new node in front of the one with larger key.)

To access an object with a specified key k:

> Make a linear search of the objects in the linked list to try to find an object with key k.
> If no such object is found, report failure.
> Else report success and return a copy of the object with key k.

To remove an object with a specified key k:

> Make a linear search of the objects in the linked list to try to find an object with key k.
> (Again, the search should involve moving two pointers along the list, one behind the other.)
> If no such object is found, report failure.
> Else copy the object with key k into a temporary location.
> Remove and deallocate the node containing the object with key k.
> Report success and return a copy of the object in the temporary location.

To test for empty:

> Test whether `front == NULL`.

To make the table empty:

> One by one, deallocate the nodes in the linked list.

To traverse the table in key order:

Traverse the list from front to back.

To perform the Find Minimum operation (the version that does not change the table):

If the table is empty, report failure.

Else report success and return a copy of the object in the first node of the list.

To perform the Find Maximum operation:

If the table is empty, report failure.

Else advance a pointer to the end of the list and return a copy of the object in the last node.

Analysis of Table Implementation 3.

When n objects are in the table, then

-> Successful insertion begins with an unsuccessful linear search that requires at least $\Theta(1)$ time and at most $\Theta(n)$ time. On average, half the list must be searched, so the average search time is $\Theta(n)$. Splicing in a new node requires only $\Theta(1)$ time. Unsuccessful insertion begins with a successful linear search, with the same minimum, maximum, and average execution times.

-> Successful and unsuccessful access have the same minimum, maximum, and average execution times as insertion.

-> Test for empty is a $\Theta(1)$ operation.

-> Making the table empty is a $\Theta(n)$ operation.

-> Find Minimum is a $\Theta(1)$ operation.

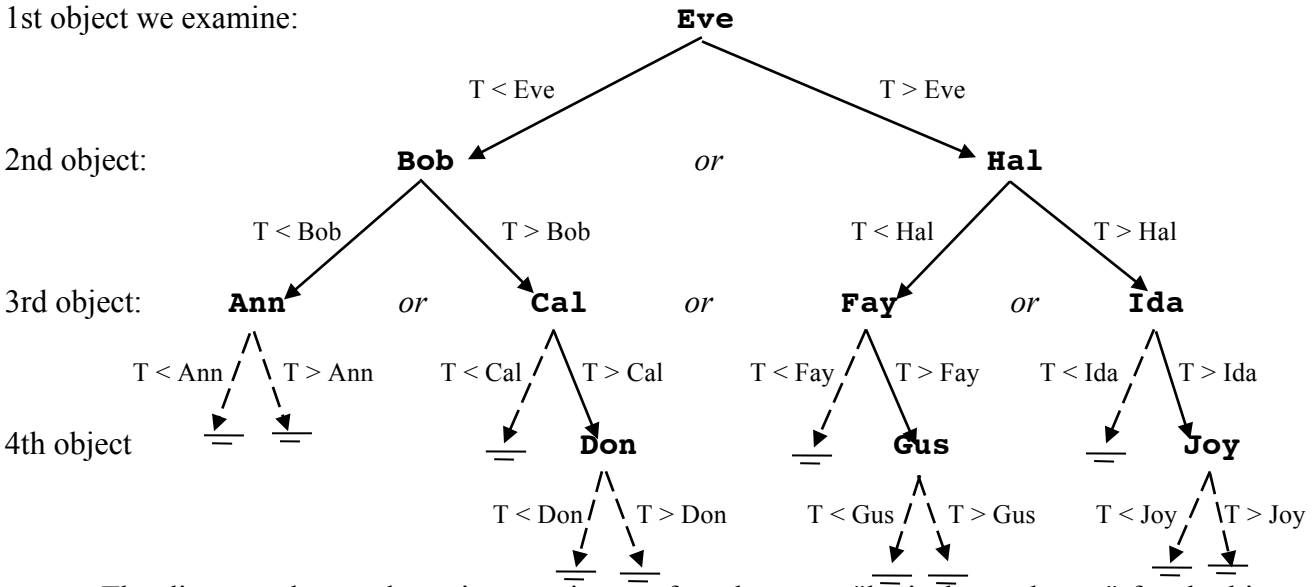
-> Find Maximum is a $\Theta(n)$ operation (unless we decide to keep a pointer to the last node in the list at all times, in which case it is a $\Theta(1)$ operation).

Conclusion: in almost every way, the ordered linked list implementation is inferior to the ordered array implementation of the table ADT.

It would be nice if there were some kind of data structure that would combine the two most desirable features of ordered arrays and ordered linked lists: we can perform binary searches on ordered arrays, and we can insert and remove objects from a linked list in $\Theta(1)$ time. As it turns out, there is such a data structure. To introduce it, let's review how a binary search works on an ordered array (see page 3-9). Suppose we are about to make a binary search for a target T in the following array:

	0	1	2	3	4	5	6	7	8	9
a	Ann	Bob	Cal	Don	Eve	Fay	Gus	Hal	Ida	Joy

The diagram below shows the "logical structure" of the search. We begin by calculating the "middle" subscript of the array by averaging the extreme subscripts: `mid = (0 + 9)/2`, which gives 4. Thus we examine the contents of cell 4. If T, the target of our search, is Eve, we can stop. If T alphabetically precedes Eve, we restrict our search to the subarray `a[0..3]` and calculate the middle subscript of that array: (0 + 3)/2 = 1. If, instead, T follows Eve alphabetically, we restrict our search to the subarray `a[5..9]` and calculate the middle subscript of that array: (5 + 9)/2 = 7. Then we examine the appropriate cell, and if the target T does not match, we go left or right to examine a third cell. Etc.



The diagram shown above is sometimes referred to as a "logical search tree" for the binary search algorithm on the array shown above. It is a "binary" tree because at each "node" shown in boldface, there is a two-way branch away from that node.

Now suppose that each of the bold-face names above is actually carried in a "`node`" struct object having a place for a name and two pointers. Then we would have a linked structure that looks just like the logical search tree above. A pointer to the top node in the structure would give us access to all parts of the structure. Binary search could be carried out on the structure, starting from the top, and insertions could take place at the bottom of the structure. Before we pursue this further, we're going to make a study of the important notion of a "binary tree". This is the subject of Chapter 7.

5. Hash Tables

Problem 1: Suppose we need to create a table in which to store approximately 7000 objects. Each object will have a numerical key that's an integer in the range from 0000 to 9999 (i.e., 4-digit integers). The operations to be performed on this table are access, insert, and remove. We will never need to perform a find-min, a find-max, or a traverse-in-order. What are some implementations we might consider for this table? Looking back over what we have done so far with the ADT "table", the following possibilities present themselves.

(1) Unordered array. There are way too many objects to consider using an unordered array.

(2) Ordered array. If the table is going to be fairly static (i.e., there will be far more access operations on the table than insertions or removals), then we might want to consider an ordered array, which is easy to implement and provides pretty fast access times using binary search. Insertions and removals will be slow, but they are expected to be relatively rare events.

(3) Binary search tree. If the table is going to be dynamic (i.e., insertion and removal will occur frequently), then a better choice might be one of the good kinds of binary search trees (AVL, red-black, or splay).

In both (2) and (3), average operation times will be $O(\log_2(N))$, where N is the number of objects in the table at the moment when the operation occurs. (For more precise and detailed information about execution times, refer back to the sections on binary search and shifting in arrays and to the various operations in binary search trees.)

As it turns out, in this particular situation there is an implementation that is far better than (2) or (3) above. The idea is this: we set up an array having 10,000 cells, each capable of holding one object. The subscripts for this array run from 0 to 9999, and the integers in this range are exactly the range from which the keys for our objects will have been taken. This means that we can store each object in the cell having subscript equal to the key for that object. One might think of it this way: since no two objects in the table can have the same key, each object has a "reserved room" waiting for it at a motel with 10,000 rooms. Many of the rooms (about 3,000 of them in our problem) will never be claimed, but whenever an object shows up and asks its reserved room, it will be placed there immediately (unless an object with the same key has already occupied the room). [Technical note: there will have to be some way of placing a "sentinel object" in every unoccupied room to allow the program to determine whether a room is occupied or not; perhaps the "sentinel object" could the the object with a zero in every data bit. Alternatively (and this is generally considered to be a better method), each cell could have two compartments, one of which is boolean and tells whether the other compartment is empty or not. Before any objects are placed in the array, all cells will have to be initialized.]

How much execution time will be required by each of the three table operations?

Insert: $\Theta(1)$

Access: $\Theta(1)$

Remove: $\Theta(1)$

Problem 2: Now let's change the problem slightly. Assume that we want to build a table to hold personnel records for about 7,000 employees. The employee records will be accessed by the SSN (Social Security Number) of the employees. That is, the key for each record in the table will be the employee's SSN. Each SSN will be treated as a 9-digit integer (the hyphens will be omitted). How should the records be stored?

There are one billion possible SSNs. They lie in the range from 000000000 to 999999999 . If the memory of our computer is so gigantic that it can hold an array of one billion records, each of which may be several thousand bytes long (giving a total of several trillion bytes), then we can make an obvious modification of the solution given for Problem 1 on the preceding page. Such gigantic main memories are not yet available (though we can expect them in the next ten or fifteen years). At present, we must seek a different solution.

How about the following: let's create an array of size 10,000 (as in Problem 1), and then use just the last four digits to determine where a personnel record will go. That is, when we insert an object into the table, we'll ignore the first five digits and use the last four as the index of the cell into which the object will be placed.

Immediately you see a problem here. Two different individuals may have different keys, but the last four digits of their keys may match. Thus when an object is about to be inserted into the table, the cell into which we want to insert it may not be empty, but may contain an object with different key (but same last four digits). That is, during an insertion operation a "collision" may occur. If we want to use this method, then we are going to have to formulate some "collision resolution scheme" that will allow us to divert the incoming object to a different cell, such as the one "next door to the right" (with wrap-around). And if that cell is also occupied, then we will divert the incoming object to still some other cell.

Of course, this means that the access operation will be more complicated. Now, when we want to access a record with a given 9-digit key, we will first look in the cell given by the last four digits of the key. If the cell is occupied but the other five digits don't match, then we will have to look in the cell to which an incoming record would be diverted, and if that is occupied by a non-matching record, then we will have to look farther.

This method for creating and maintaining a table is called the "hash table method". The name comes from the fact that we "chop up" the key and make a "hash" out of it, and then the hash is used as the index into a storage array. The array is called a "hash table". A more precise definition is given on the following page.

Definition: A **hash table** is an implementation of the ADT "table" in which an array of some size M is used to store the objects held by the table. Associated with the array is a **hash function**, call it H, that acts on keys of the objects and turns them into integers in the range from 0 to $M - 1$ (the subscripts of the array). When an object with key k is to be inserted into the table, an attempt will be made to store it at its **hash address** $H(k)$, i.e., in the cell with subscript $H(k)$. The hash table must also have a **collision resolution scheme**, which is required to be an algorithm for determining where an object will be placed if there is already an object with different key stored in the cell with subscript $H(k)$.

As you can see from the definition about, when a programmer wants to consider using a hash table to implement the ADT "table", the programmer must decide on an array size adequate to the data expected, decide on (or construct) a hash function, and choose some well-defined collision resolution scheme. We'll begin our discussion of hash tables by looking at the problem of how to create good hash functions. Later we will look at the problem of choosing a good collision resolution scheme.

5.1. Creating Good Hash Functions

For the most part, programmers use hash tables only when the keys of the objects to be stored are *integers or character strings*. (Other types of keys usually require hash functions that are too time consuming to compute efficiently.) Thus we will restrict our attention in this discussion to hash functions on keys that are integers or character strings.

What are the desirable properties for a hash function? One important property is that the function not be extremely difficult to compute. When we choose to use a hash table, it's because we are hoping for extremely fast access times, $\Theta(1)$ on average, if possible. It will defeat our purpose if each time an access is performed the computation of the hash address for the prescribed key requires a large number of computational steps by the CPU. This criterion can be deceptive, however. A hash function that would be extremely tedious and time consuming for a human being to compute may sometimes be quite easy and "natural" for the CPU. Judging this kind of thing accurately requires some understanding of machine operations -- the kind of thing you study in our C335 course.

Perhaps the single most important property that a hash function should have is that it scatters the keys uniformly and randomly over the "address space" (i.e., the set of all hash addresses in the table). Ideally, given a key chosen at random from the set of all keys, the hash function should be equally likely to hash that key into any of the addresses in the table. Crucially, there should be no address in the table that the hash function can never generate, and there should be no "regions" in the table that are more likely to receive hashed keys than other regions in the table

Question: What is the formula for the the hash function H(k) that we proposed to use in the solution to Problem 2 on page 5-2?

Hash functions on integers and strings almost always have the following form (in C++):

```
int H (keytype k, int M)  // M is the size of the array
{
    .....
    .....      // code to "chop up" the bits that constitute the
    .....      // the key  k  and produce a large integer hashVal
    .....

    return hashVal % M;  // return remainder after division by M
} // H()
```

Hash Functions on Integer Keys

In many cases, it is possible to have a hash function on integer keys that's as simple as this:

```
int H (keytype k, const int M)  // M is the size of the array
{
    return k % M;
} // H()
```

That's exactly the form of the hash function we were using in our solution to Problem 2. In that problem, M was 10,000 .

Under certain circumstances, however, this simple technique may not work very well. Consider this example (in which the numbers involved are unrealistically small).

Example 1:Suppose we plan to create a hash table with 100 addresses (0 to 99). Suppose that the keys will be integers in the range from 300 to 449 . Then the hash function H(k) = k % 100 will transform the 100 keys in the range 300 to 399 into the integers 0 to 99 in linear way. The remaining 50 keys from 400 to 449 will be transformed into the integers from 0 to 50 . This means that the addresses in the left half of the array will be twice as likely to be produced by H(k) as the addresses in the right half of the array. This kind of "bunching up" is undesirable.

What might we do in this particular case to distribute the keys more evenly over the address space? How about this: double each key before taking the remainder after division by 100. This would send keys in the range

300 to 349 into the range 600 to 698 and then reduce them into the range from 0 to 98,

350 to 399 into the range 700 to 798 and then reduce them into the range from 0 to 98,

400 to 449 into the range 800 to 898 and then reduce them into the range from 0 to 98.

This is better in one respect, but it has a very bad feature: this modified hash function will never produce an odd address. Thus half of the addresses in the table will never be the destination of a keyed object being inserted into the table. This guarantees twice as many collisions as would occur with a better hash function.

In this case, multiplying each key by 3 before taking the remainder turns out to work reasonably well. The reason that 3 works well here but 2 does not is that the integer 3 has no common prime factor with the table size 100 , whereas 100 is divisible by 2 . More generally, if you want to choose a hash function of the form

$$H(k) = (c * k) \% M$$

where c is a constant integer designed to "scatter" the keys, then you must make sure that c and M do not have a common factor other than 1 . This can be guaranteed by choosing the table size M to be a prime number, although that is not necessary.

In the example we are considering, some programmers would advise using a somewhat larger constant multiplier than 3 for the table of size 100 . The reason is that in many situations it is a good idea to force the hash function to transform "neighboring" keys into hash addresses that are far apart. The hash function H(k) = (c * k) % M transforms keys that are just 1 apart into hash addresses that are c cells apart. Thus a choice such as c = 19 might work much better. Question: is it harder for the computer to multiply a key by 19 than to multiply the key by 3 (the way it is for ordinary humans)? No: the computer takes exactly the same amount of time to multiply together any two integers in the usual integer data types available in most programming languages.

Hash Functions on Character Strings

We could argue that a character string is really nothing but an integer in disguise. After all, each character in a string is represented in the computer by an 8-bit pattern (its ASCII value), so we can turn the string into an integer simply by concatenating the bits that make up the characters in the string. For example, given the string "KNIGHT" we can look up the ASCII values of these six letters, express them in binary (i.e., bit) notation, and then put them side by side ("concatenate them") to form a 48-bit integer.

```
010010110100111001001001010001110100100001010100

    K       N       I       G       H       T
```

Thus every character string can be considered to be an integer, and we can just use the ideas we have already discussed for creating hash functions on integer keys.

Unfortunately there is a problem with this approach. Most CPUs today do not have machine instructions to handle integers larger than 32 bits. This means that arithmetic with integers having 33 or more bits must be handled by special software routines. (Perhaps you have written such software in C201?) What this means is that if we force our hash functions to do arithmetic that requires software arithmetic routines, the hash functions will be seriously inefficient. For this reason, the approach suggested above is used only when all the character string keys are guaranteed to be no longer than 4 characters (32 bits), which is not a very common situation. Thus we need to look at some other methods for hashing character strings.

Method 1: We could just treat each ASCII value as an 8-bit integer, add up the ASCII values, and divide by the table size.

```
int H (char k[], const int M)
{
    int hashVal = 0;

    for (int i = 0;  i < strlen(k);  ++i)
        hashVal += int(k[i]); // cast char k[i] to int and sum it

    return hashVal % M;
} // H()
```

While this may seem like the natural and sensible thing to do, experience has shown that in practice it gives very poor performance. The reason for this is that in real life, most of the character strings keys tend to be close to the same length (think of family names in America, which tend to be about 7 characters long on average). Adding up the ASCII values tends to produce sums that cluster together rather strongly, with many different strings producing exactly the same sum (STOP, POST, POTS, TOPS, SPOT, OPTS all hash alike), while other sums are seldom if ever produced. Thus the addresses produced by this method are not well distributed over the address space.

Method 2: We can do better by taking into account the position of a character within a string. That is, we can give different numerical weights to the same character, depending on where it appears in the string. For example, we might do something like this:

```
hashVal = 3*k[0] + 8*k[1] + 13*k[2] + 18*k[3] + ....
```

```
int H (char k[], const int M)
{
    int hashVal = 0;

    for (int i = 0;  i < strlen(k);  ++i)
        hashVal += (5 * i + 3) * int(k[i]);

    return hashVal % M;
} // H()
```

This function will hash STOP, SPOT, POST, POTS, TOPS, and OPTS to different addresses.

While this is a definite improvement over Method 1, experience has shown that Method 2 does not give as good performance in many situations as one might expect. Also, integer multiplication is a somewhat lengthy operation, even when performed in hardware, so this function will generally take more time than we like on keys that are not short.

Method 3: An operation that is much faster in hardware than addition or multiplication is the bitwise Exclusive-Or operation. Suppose we let the symbol ^ denote Exclusive-Or (this is the symbol used in the C and C++ programming languages). Then the operation on two bits is defined by these equations:

```
0 ^ 0 = 0     0 ^ 1 = 1     1 ^ 0 = 1     1 ^ 1 = 0 .
```

The Exclusive-Or of two bytes is peformed bit-wise. For example,

```
10011101 ^ 11001001  =  01010100 .
```

We could create a hash function by forming the Exclusive-Or of all the characters in the key, treating the result as an 8-bit integer, and forming the remainder after division by M .

```
int H (char k[], const int M)
{
    char hashVal = 0;
    for (int i = 0;  i < strlen(k);  ++i)
        hashVal = hashVal ^ k[i];
    return int(hashVal) % M;
} // H()
```

As you can guess, this method suffers from exactly the same problems as Method 1. It's even worse however because it never produces addresses larger than 255 (the largest 8-bit integer).

Method 4: This hash function on character strings is recommended by the author of your text. Although it seems complicated (and would be very time consuming for a human being to carry out), it makes use of two operations that are very fast for a CPU: Exclusive-Or and Bit-Shift.

```
int H (char k[], const int M)
{
    int hashVal = 0;

    for (int i = 0;  i < strlen(k);  ++i)
    {
        int temp = (hashVal << 5) ^ int(k[i]);

        hashVal = temp ^ hashVal;
    }

    return hashVal % M;
} // H()
```

The expression `(hashVal << 5)` gives the integer obtained by left-shifting the bits in the `hashVal` integer by 5 positions. For example, if `hashVal` had the value
`0000 0010 1101 1001`, then the expression `(hashVal << 5)` would have the value
`0101 1011 0010 0000`

[Reassurance: I will never ask you to calculate by hand any hash values given by the function above. In fact, you should not even memorize the code. I just thought it was important for you to see a hash function of the kind that is used in practice in the real world.]

There is really no such thing as a general-purpose hash function that performs well in all situations. A programmer really has to think about the particular keys to be hashed and the amount of data to be stored in the table. Pains should be taken to make the hash function break apart naturally occuring clusters among the keys.

There have been many papers written on how to choose a good hash function in various situations. You can get an idea of the extent of the literature by consulting a particularly good book called ***The Art of Computer Programming: vol. 3 : Sorting and Searching***, by Donald Knuth. In this book (now in its 2nd edition), Knuth devotes approximately 50 pages to the subject of the construction of good hash functions. This is an invaluable resouce for the serious programmer.

The three volumes (of a projected seven volume series) were the first books to give a well-organized and encyclopedic treatment of the subject of data structures. In these books Knuth distilled the discoveries in thousands of papers on various topics in data structures. For this reason, Knuth (a professor of Mathematics and Computer Science at Stanford University) is often called "the Father of Data Structures".

Special Purpose Hash Functions

In the preceding discussion we assumed tacitly that the implementer of the hash table had in mind a situation in which it would probably not be possible to predict exactly what keys were going to be inserted into the table during its lifetime. There are situations, however, in which the maker of the hash table knows exactly what keys will go into the table. Here is an example.

Suppose you are helping write a compiler for a complicated language such as C++ with many keywords. When the compiler reads a source code file, it must be able to recognize the keywords in the program and generate machine code to carry out the actions specified by the keywords. Thus it is necessary to have a table of the keywords and their "meanings". Then as it compiler reads through the source code, it can take each separate word in the program and look it up in the keyword-table to see whether it is a keyword, and if it is, what its meaning is. Since this look-up operation must be performed on evey word in the program, it is desirable that the look-up operation be as fast as possible. This suggests that a hash table should be used to store the keywords and their meanings.

Certainly you can look up in a reference book the complete list of keywords for C++ (or whatever language your compiler will act on). This means that you can construct a table and a hash function that are specially designed for that particular set of character strings. You can study the characteristics of these strings (keywords) and then make a hash function that's as simple as possible while producing as few collisions as possible (ideally none). Typically, the people who build the keyword hash table in a compiler spend a month or more just experimenting with many hash functions until they find one that's extremely fast.

What distinguishes this case from the general one is that the designer of the hash table tailors the hash function to a known set of keys. By contrast, when faced with making a table to hold customer records or student records, the designer of the hash table cannot predict in advance exactly which keys will arrive for inclusion in the table.

5.2. Collision Resolution Schemes

Now let's look at the other important component of a hash table, a collision resolution scheme, i.e., an algorithm for dealing with the problem that occurs during insertion when the key of the entering object hashes to an address already occupied by an object with a different key. We'll look at three popular collision resolution schemes. They are called "linear probing", "quadratic probing", and "separate chaining".

Linear Probing (the "go next door algorithm")

This simple algorithm works as follows. Suppose we want to insert an object with key k. We form the hash address $H(k)$ and then examine the cell with that subscript. If the cell is empty, we copy the object into the cell. If the cell is not empty, we compare the key of the object in the cell with k. If they are the same, then the insertion fails; but if they are different, then we go to the cell just to the right (or to cell 0 if $H(k)$ is the subscript of the right-most cell) and attempt the insertion there. Again, if the cell is empty, we copy the object into that cell. If the cell is not empty and the key of the object in the cell does not match k (a match causes failure), then we go to the next cell to the right. And so on. That is, we make a "linear" rightward search for a cell into which we can place the entering object.

How can we tell whether a cell in the table is empty (and thus may contain "garbage bits" that might coincidentally match a key)? The best way to do this is to create a parallel array of boolean values, in which "false" indicates that the corresponding table cell is empty ("nothing there") and "true" indicates that the corresponding table cell contains a valid object.

What would happen during an insertion if we had previously inserted objects into all the cells of the array? In that case, if the object we wanted to insert did not have a key that matched a key in the table, then the search for a place to insert the object would loop forever, because every cell would be found to be non-empty. Programmers usually take care of this problem by keeping a counter that tracks the number of empty cells in the table at any given time. If the counter's value drops to 1, then no further insertions are allowed until one or more removals have taken place. This means that there is always at least one empty cell in the table to stop searches.

How is the access operation carried out on a table in which Linear Probing is used? Suppose we want to access an object with specified key k in the table. We form the hash address $H(k)$ and then examine the cell with that subscript. If the cell is empty, the access operation fails. If the cell is not empty, then we compare the key of the object in the cell with k. If they are the same, the access operation has succeeded and we return a copy of the object in that cell. If, however, the keys do not match, then we move one cell to the right (with wrap-around) and repeat the procedure there. We continue in this way until we find an object whose key matches k (successful search) or we arrive at an empty cell (failure). Since we never allow the table to fill completely, we can be sure there will always be at least one empty cell.

Note that unsuccessful access operations end only when an empty cell is encountered. As noted above, we will always keep at least one cell in the array empty to stop such searches.

Now consider what happens if a removal operation is carried out on a table with Linear Probing. A key must be specified. We begin with an access operation on the table, as in the preceding paragraph, but now if we find an object with the specified key we remove it from the table instead of just returning a copy. Should we now mark that cell as empty? NO!! If we do that, we may break a "chain" by which we reach other objects in the table during searches. On the other hand, we also can't just leave the cell as it was, for then the object has not been removed. We have a problem that requires a reasonable solution (unless we want to forbid removals on this table).

Here is the solution that is usually used. Instead of creating a parallel array of boolean values to indicate whether each cell is empty or not, we create a parallel array of characters that indicate whether the corresponding table cells are "empty" (`'e'`), "occupied" (`'o'`), or "previously occupied" (`'p'`). Then when an object is removed from the table, we can change its corresponding status value from `'o'` to `'p'`. Of course, this change requires that our insertion and access algorithms be modified to take into account this new way of marking the cells.

Below is pseudo-code for the various operations on a table with Linear Probing. The following notation is used:

```
a = name of the storage array
status = name of the parallel array of status characters
H = name of the hash function
key(a[h]) = the key of the data object at address  h
numberOfECells = the number of empty cells

Access an object with specified key  k
    h = H(k);
    loop until a return occurs
    {
        if (status[h] == 'e')
            return indication of failure;

        else if (status[h] == 'p')
            increment  h   by  1  (with wrap-around);

        else // status[h] must be 'o'
            if (key(a[h]) == k)
                return a copy of the object in cell a[h];
            else
                increment  h  by  1  (with wrap-around);
    }
```

The following algorithm begins by performing a search for a cell into which a given object x can be placed. The first cell it comes to that's empty or previously occupied is where x will be placed. But the search must continue past cells that were previously occupied, so a variable is needed to keep track of the first previously occupied cell encountered during the search. We'll let `availableCell` be our name for that variable.

Insertion (with Linear Probing) of an object x

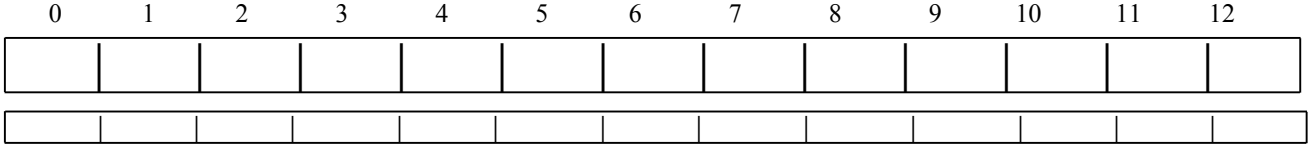
```
int availableCell = -1;  // a sentinel value

int h = H(key(x));  // compute the hash address for the key of x

loop until a return occurs
{
    if (status[h] == 'o')
        if ( key(x) == key(a[h]) )
            return indication of failure;
        else
            increment  h  by  1  (with wrap-around);

    else if (status[h] == 'p')
    {
        if (availableCell == -1)
            availableCell = h;  // "remember" where to put x
        increment  h  by  1  (with wrap-around);
    }

    else // status[h] must be 'e';  we can insert x somewhere
    {
        if (availableCell != -1) // we previously hit a p-cell
            h = availableCell;
        else if (numberOfECells == 1)     // not allowed to insert
            return indication of failure;     // a new object
        else
            -- numberOfECells;

        // Now h is the insertion location

        a[h] = x;
        status[h] = 'o';

        return indication of success;
    }
}
```

Removal of an object with specified key k

```
h = H(k);

loop until a return
{
    if (status[h] == 'e')
        return indication of failure;

    else if (status[h] == 'p')
        increment  h  by  1  (with wrap-around);

    else // a[h].status must be 'o'
        if ( key(a[h]) == k )
        {
            status[h] = 'p';
            return a copy of the object in cell a[h];
        }
        else
            increment  h  by  1  (with wrap-around);
}
```

Example: Suppose we create a hash table using an array of 13 cells. Suppose that we decide to use the Linear Probing algorithm for collision resolution. Show the evolution of the table as we perform the following operations:

Insert an object whose key COW hashes to location 10 .
Insert an object whose key BAT hashes to location 0 .
Insert an object whose key PIG hashes to location 11 .
Insert an object whose key DOG hashes to location 10 .
Insert an object whose key ANT hashes to location 11 .
Attempt to access an object whose key is DOG , which hashes to location 10 .
Attempt to access an object whose key is RAT , which hashes to location 12 .
Attempt to remove an object whose key is COW, which hashes to location 10.
Attempt to access an object whose key is DOG , which hashes to location 10 .
Attempt to remove an object whose key is BAT, which hashes to location 0.
Insert an object whose key EEL hashes to location 9 .
Insert an object whose key FOX hashes to location 9 .

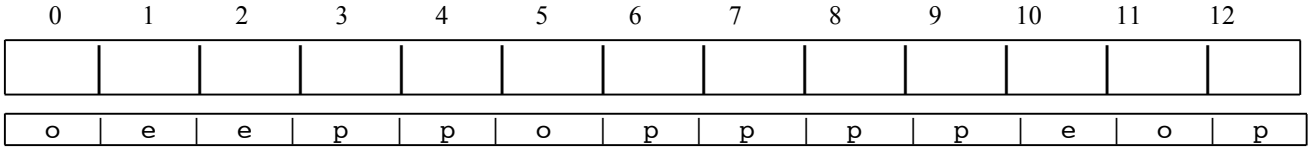


Question: at the end of all these operations, what is the *average* number of cells that will be examined during a successful search in this table?

A disturbing property of hash tables with Linear Probing is that when the table has seen a lot of activity over time, many cells may be marked `'p'`. This means that when searches are made for items with specified keys, long chains must be searched, and many of the cells in the chains will contain no data. (Remember, all searches must go until the key in question is found or a cell marked `'e'` is reached.) Thus the execution time for all unsuccessful searches will get slower and slower when fewer and fewer cells marked `'e'` are present in the table.

Another disturbing property is that the variable that keeps track of the number of empty cells is never allowed to increase. It is changed only when a cell that was empty becomes occupied, at which point it is decreased. This means that once the number of cells marked `'e'` falls to 1, no further insertions can *ever* take place in the table -- at least not the way we have set up the insertion algorithm. Even if all the data objects were removed from the table, no insertion of a new data object would be possible. There are two remedies for this problem (and both of them also help alleviate the problem mentioned in the preceding paragraph).

(1) From time to time, a right-to-left search of the table could be made, and all chains of cells having status `'p'` and terminated by a cell with status `'e'` could be re-set to have status `'e'`. For example, in the array below it would be possible to start at cell 9 and move to the left, changing all the status fields in cells 9, 8, 7, and 6 to `'e'` and increase `numberOfECells` by 4.



This is a partial remedy, since it does not remove all the `'p'` entries from the table.

(2) When the number of empty cells reaches (or perhaps merely gets close to) the limit 1, create a fresh, empty copy of the hash table and then go through the entire old table and hash the data objects in the `'o'` cells to the new table. (Then deallocate the old table, of course.)

Both of these remedies are somewhat time consuming, especially on large tables, so they are not often used.

Execution Times for Hash Tables with Linear Probing

Suppose we have a hash table with M cells, and suppose the table uses linear probing. What execution times should we expect for the three primary operations of access, insertion, and removal? A little thought suggests that the execution times will be strongly influenced by the number of data objects that are stored in the table, and also the number of cells that are marked "previously occupied", since all searches continue past `'o'` and `'p'` cells. If almost all the cells in the table are marked `'e'` (empty), then we can be almost certain that searches will never require that more than one cell be examined, and thus all three operations can be expected to run in $\Theta(1)$ time.

More generally, let n denote the number of data objects actually in the table (i.e., the number of cells marked `'o'`) and let k denote the number of previously occupied cells. A little thought tells us that we always have $n + k < M$. The best possible execution time in this table is $\Theta(1)$ for all three of the primary operations. This best possible time occurs when a search stops at the very first cell it examines. The worst that can happen is that all the objects in the table will have hashed to the same address (an unlikely event if the hash function is a good one), in which case the longest possible search will require $\Theta(n + k)$ time. Since such a search can occur in any of the three primary operations, this is the worst case execution time for all three operations.

The worst case will occur very rarely, so a much more interesting question is this one: What is the *average* execution time we can expect for the three primary operations on these tables? Since all three operations begin with a search for a given key, the answer depends on the fraction of the table that is occupied or has been previously occupied. This fraction is called the "load factor" for the table, and it is defined as follows:

$$\text{load factor} \;=\; \lambda \;=\; \frac{n + k}{M}.$$

Since $n + k$ is always less than M in Linear Probing, we see immediately that $0 \le \lambda < 1$ at all times during the life of the table.

Successful searches: the *predicted average* number of cells that will have to be examined is

$$\frac{1}{2}\left[\, 1 + \sum_{p=1}^{M-1} \frac{1}{M^p} \frac{(n+k)!}{(n+k-p)!} \,\right] \;\approx\; \frac{1}{2}\left(1 + \frac{1}{1-\lambda} \right)$$

"Predicted average" number here means "theoretical average" number. It is obtained by averaging all averages taken over all possible tables that have load λ. The actual average for any particular table will probably deviate more or less (usually only slightly) from the predicted average. ***I DO NOT EXPECT YOU TO MEMORIZE THIS FORMULA.***

The approximation breaks down as $\lambda \to 1$. In that case, the expected number is about $\frac{M}{2}$.

Unsuccessful searches: the predicted average number of cells that will have to be examined is

$$\frac{1}{2}\left[\,1 + \sum_{p=1}^{M-1} \frac{p+1}{M^p}\,\frac{(n+k)!}{(n+k-p)!}\,\right] \;\approx\; \frac{1}{2}\left(\,1 + \frac{1}{(1-\lambda)^2}\right)$$

The approximation breaks down as $\lambda \to 1$. In that case the predicted average is about $\sqrt{\frac{\pi M}{8}}$ (again, you are not expected to memorize these formulas).

What do the preceding formulas tell us about the *predicted* execution times for access, insertion, and removal operations? Since each of these operations involves a search followed by work that requires contant time, we see that the execution times are dominated by the times required by the searches. Thus we have the following execution times when the load factor λ is not near 1 :

Predicted Access Execution Times:	Successful :	$\Theta\left(\frac{1}{1-\lambda}\right)$	(Do not memorize.)
	Unsuccessful:	$\Theta\left(\frac{1}{(1-\lambda)^2}\right)$	"
Predicted Insertion Execution Times:	Successful :	$\Theta\left(\frac{1}{(1-\lambda)^2}\right)$	"
(Successful insertion begins with an *unsuccessful* search.)	Unsuccessful:	$\Theta\left(\frac{1}{1-\lambda}\right)$	"
Predicted Removal Execution Times:	Successful :	$\Theta\left(\frac{1}{1-\lambda}\right)$	"
(Successful removal begins with a *successful* search.)	Unsuccessful:	$\Theta\left(\frac{1}{(1-\lambda)^2}\right)$	"

It is important to remember that the values given above are what we expect before we ever construct our table. That is, they are *predicted values* for hash tables with linear probing, and the predictions are based on the assumption that the hash table distributes keys uniformly over the table. When an actual table has been constructed, the operation times may be better or worse than the predicted ones. In very rare cases by bad luck, and in cases when the hash function turns out to be poor, the observed execution times will be much worse than predicted.

In practice, experienced programmers use hash tables with linear probing only if the load factor on the table can be expected to remain under 0.5 (50%). This makes it extremely likely (though it does not guarantee) that all operation times will be satisfactorily fast. Notice that this means that 50% or more of the memory space occupied by the table will be wasted (go unused). This is a price that programmers are often willing to pay for fast table operations. It is an important example of a space-time trade-off.

Quadratic Probing

With Linear Probing, chains of objects that hash to different addresses tend to "coalesce" into long chains that increase predicted search times. The Quadratic Probing collision resolution scheme atteempts to avoid or reduce coalescing of chains by storing displaced objects farther from their hash addresses. In Linear Probing, when a collision occurs at hash address H(k) , we then try addresses

$H(k) + 1$, $H(k) + 2$, $H(k) + 3$, $H(k) + 4$, et c. (with wrap-around)

successively. These are reached by setting h = H(k) initially and then incrementing h by 1 each time through the search loop. In Quadratic Probing, when a collision occurs at H(k) , we then try

$H(k) + 1^2$, $H(k) + 2^2$, $H(k) + 3^2$, $H(k) + 4^2$, et c. (with wrap-around)

successively. Since

$[H(k) + 1^2] - [H(k) + 0] = 1$
$[H(k) + 2^2] - [H(k) + 1^2] = 3$
$[H(k) + 3^2] - [H(k) + 2^2] = 5$
$[H(k) + 4^2] - [H(k) + 3^2] = 7$
$[H(k) + 5^2] - [H(k) + 4^2] = 9$

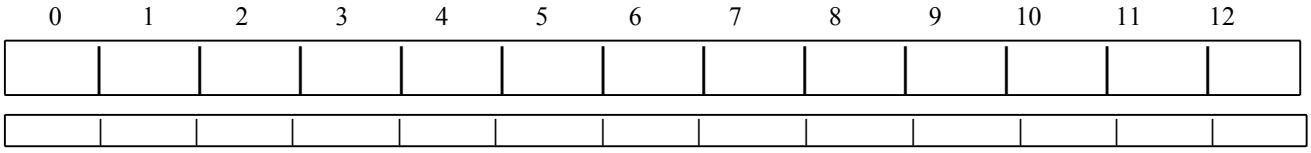
we see that our search can be conducted by setting a variable h = H(k) initially and then incrementing it by successivly larger odd numbers as the search proceeds. For example, the access operation would look like this (cf. page 5-11; the bold type below shows the modified code).

Access an object with specified key k

```
h = H(k);
oddIncrement = 1;
loop until a return occurs
{
    if (a[h].status == 'e')
       return indication of failure;
    else if (a[h].status == 'p')
    {
        h += oddIncrement  (with wrap-around);
        oddIncrement += 2;  // go to next higher odd number
    }
    else // a[h].status must be 'o'
        if (key(a[h].datum) == k)
            return a copy of a[h].datum;
        else
        {
            h += oddIncrement  (with wrap-around);
            oddIncrement += 2;  // go to next higher odd number
        }
}
```

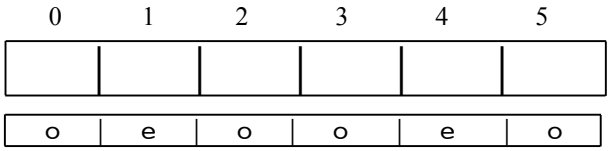
Similar modifications would be made in the code for the insertion and removal operations.

Example: Let's re-do the example given on page 5-13, but this time let's use Quadratic Probing. We'll show the evolution of the table as we perform the following operations:
Insert an object whose key COW hashes to location 10 .
Insert an object whose key BAT hashes to location 0 .
Insert an object whose key PIG hashes to location 11 .
Insert an object whose key DOG hashes to location 10 .
Insert an object whose key ANT hashes to location 11 .
Attempt to access an object whose key is DOG , which hashes to location 10 .
Attempt to access an object whose key is RAT , which hashes to location 12 .
Attempt to remove an object whose key is COW, which hashes to location 10.
Attempt to access an object whose key is DOG , which hashes to location 10 .
Attempt to remove an object whose key is BAT, which hashes to location 0.
Insert an object whose key EEL hashes to location 9 .
Insert an object whose key FOX hashes to location 9 .



Question: at the end of all these operations, what is the *average* number of cells that will be examined during a successful search in this table?

In Quadratic Probing, a new hazard awaits the unwary. If the size of the array is chosen incautiously, then an unfortunate thing can occur during an insertion: the "jumping" during the search can fail to reach every cell. Large numbers of cells may be missed entirely by the search algorithm, which may go into an endless loop when it never reaches an empty cell, even though many of the cells may be empty. Here is an example:



Suppose we need to insert an object whose key hashes to cell 2 . We try 2 , then 2 + 1 = 3 , then

$2 + 4 = 0$ (with wrap-around), then $2 + 9 = 5$, then $2 + 16 = 0$, then $2 + 25 = 3$, then $2 + 36 = 2$, and so on. It can be proved that this search will never hit any cells except 2, 3, 5, and 0. Thus even though there are two empty cells, the search will never find one.

The following theorem gives us the information we need in order to use Quadratic Probing safely.

Theorem: If we implement a hash table using Quadratic Probing, and if we
(a) choose a prime number for the size of the array, and
(b) make sure the load factor on the table never exceeds 0.5,
then a search in the table for a specified key will never probe the same location twice before reaching an empty cell.
Proof. See the text by Weiss.

What does an analysis of the execution times for the operations yield in the case of Quadratic Probing? The best and worst cases are exactly what we saw for Linear Probing: best case is $\Theta(1)$ execution time for all operations; worst case is $\Theta(n + k)$ for all operations, where n is the number of occupied cells, and k is the number of previously occupied cells. How about the "average" performance? No one has yet succeeded in mkaing an exact analysis, but various "plausibility arguments" suggest that the predictied performance will be a little better than Linear Probing, and experience has confirmed that this is generally the case. Since only very minor modifications in the code for Linear Proing are required to change to Quadratic Probing, knowledgeable programmers generally choose Quadratic Probing over Linear Probing.

Separate Chaining

If a hash table is being implemented in a programming language that allows for dynamic allocation of memory space, then it is possible (and usually desirable) to use linked lists to handle collision resolution. In this method, the array cells hold pointers to linked lists. We give the name "Separate Chaining" to this collision resolution algorithm.

When an insertion of a data object is attempted, we hash the key of the object to obtain its hash address, and then we search the (possibly empty) linked list at that address to determine whether one of the nodes contains an object with matching key. If not, then we dynamically allocate a new node, copy the data object into the node, and insert the node at the front of the linked list at its hash address. Thus every data object in the table is "at" its correct address, although finding it at that address may involve searching a linked list. This is what we do during an access operation. Finally, removal takes place in the obvious way: to remove an object with a specified key, we hash the given key, go to the resulting hash address, search the linked list, and if we find a matching key, take that node out of the list.

Separate Chaining is very straightforward. We do not have to keep track of 'e', 'o', and 'p' cells. We can even put more objects into the table than there are cells in the array: every list can hold more than one node. Its simplicity makes Separate Chaining the method of choice in situations where dynamic allocation is possible and lots of run-time heap space is expected to be available.

What are the performance characteristics for Separate Chaining? The best and worst cases are just like they were in Linear and Quadratic Probing. As for *predicted* average behavior, we have the following simple formulas, in which n denotes the number of objects in the table:

Successful searches: the predicted average number of nodes that will be examined is

$$1 + \frac{n - 1}{2M} \approx 1 + \frac{\lambda}{2} ; \qquad \text{(Do not memorize.)}$$

Unsuccessful searches: the predicted average number of nodes that will be examined is

$$\left(1 - \frac{1}{M} \right)^n + \frac{n}{M} \approx e^{-\lambda} + \lambda , \qquad \text{(Do not memorize.)}$$

where $\lambda = \frac{n}{M}$ and $e \approx 2.718$.

As in Linear and Quadratic Probing, these formulas tell us everything we need to know about predicted execution times for successful and unsuccessful accesses, insertions, and removals.

A FINAL REMINDER ABOUT HASH TABLES

A hash table should ***never*** be chosen to implement the ADT "table" if any of the following operations are to be performed on the table: FindMin, FindMax, and Traverse in Order.

Written Exercises

5-1. Here is a partial table of ASCII values for the upper case letters of the alphabet. The ASCII value is given in base 10 and base 2 ("binary") representation:

Letter:	A	B	C	D	E	F
ASCII base 10	65	66	67	68	69	70
ASCII base 2	01000001	01000010	01000011	01000100	01000101	01000110

O	P	Q	R	S	T	U
79	80	81	82	83	84	85
01001111	01010000	01010001	01010010	01010011	01010100	01010101

Suppose we have a hash table with 10 locations, indexed (therefore) by the numbers 0, 1, 2, . . ., 9. Calculate the hash address of each of the following three character strings using the four different hash functions defined below.

TOP POT OPT

Function 1: Add up the ASCII values of the characters, divide by 10, and return the remainder.

Function 2: Form the sum 3(ASCII value of 1st char) + 13(ASCII value of 2nd char) + 23(ASCII value of 3rd char); then divide that sum by 10 and return the remainder.

Function 3: Form the product of the ASCII values of the characters (use a pocket calculator), divide by 10, and return the remainder.

Function 4: Form the Exclusive OR of the ASCII base 2 representation of the first and second characters; then form the Exclusive OR of the result with the ASCII base 2 representation of the third character. Convert the resulting binary number to base ten (see below), divide by 10, and return the remainder.

Note: a base 2 (i.e., binary) number is a sequence of bits (0s and 1s) such as $b_6b_5b_4b_3b_2b_1b_0$ with the following meaning: $b_6 \cdot 2^6 + b_5 \cdot 2^5 + b_4 \cdot 2^4 + b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0$.

Here is an example: the base 2 number 1101001 means

$1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 64 + 32 + 8 + 1 = 105$ (base ten).

5-5. Suppose you have an empty hash table with 8 locations, indexed (therefore) by the addresses 0, 1, 2, . . . , 7. Suppose that the hash function you are using hashes the following 3-character key strings into the addresses shown:

First, ART --> 7 , then RAT --> 1 , then TAR --> 0 , then BAR --> 7 , then BAT --> 5 , then APT --> 2 , then PAT --> 7 .

(a) Suppose you are using linear probing as your collision resolution algorithm. What will the hash table look like after these keys have been inserted into the table in the order given above? When BAR is accessed, how many cells will have to be examined to find BAR ? Answer the same question about each of the other six keys. What is the *average* number of cells that will have to be examined during a *successful* access operation in this particular table? (Answer this by averaging the numbers you have calculated for the seven different keys in the table.)

(b) In class you were told that when linear probing is used for collision resolution, the predicted average number of cells that will have to be examined during a successful search is approximately $\frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right)$, where λ is the load factor. Compute the load factor in part (a) above after all seven keys have been inserted. Then use that λ to compute the value of the formula given above. Does it match the average value you calculated in part (a)? If not, give an explanation.

(c) After the table is constructed in part (a) above, suppose we try to access an object that has a key not in the table. If that key hashes to address 0 , how many array cells will have to be examined in order to discover that the key is not in the table? Answer the same question about each of the other 7 addresses in the table. What is the *average* number of cells that will have to be examined in this particular table during an unsuccessful access operation? (Assume that the key on which access is attempted is equally likely to hash to any of the 8 addresses.) Compare this with the predicted average value stated in class: $\frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$. If the values do not match, offer an explanation.

(d) Suppose that instead of using linear probing, you are using separate chaining as your collision resolution algorithm. What will the hash table look like if the seven keys shown above are inserted into the empty table in the order given ? When BAR is accessed, how many nodes will have to be examined to find BAR ? Answer the same question about each of the other six keys. What is the *average* number of nodes that will have to be examined during a *successful* access operation? How does your answer compare with the predicted average stated in class: $1 + \frac{\lambda}{2}$?

(e) After the table in part (d) is constructed, suppose we try to access an object that has a key not in the table. If that key hashes to address 0 , how many nodes will have to be examined to discover the key is not in the table? Answer the same questions about the other 7 addresses in the table. What is the average number of nodes that will have to be examined during an unsuccessful access operation?

5-8. (a) Suppose a hash table with 13 cells (numbered 0 through 12) uses *linear probing* for collision resolution. Suppose the table is empty initially, and then the operations listed below are performed on the table, one after another, in the order given. Show the contents of the table when all the operations have been performed. Be sure to show the "status codes" of the cells ("e" for empty, "o" for occupied, "p" for previously occupied.) Don't erase when changes are made. Just mark out the things that are changed.

1. Insert PAT (hashes to address 9)
2. Insert ANN (hashes to address 11)
3. Insert LEE (hashes to address 1)
4. Insert SUE (hashes to address 9)
5. Insert KEN (hashes to address 10)
6. Insert JOY (hashes to address 0)
7. Insert TED (hashes to address 12)
8. Remove SUE (hashes to address 9)
9. Remove JOY (hashes to address 0)
10. Insert MAY (hashes to address 9)

When MAY is inserted (the last operation above), list the cells that are examined before the insertion occurs.

(b) Calulate the average number of cells that must be examined during a successful search for an object in the table produced in part (a). Also calculate the average number of cells that must be examined during an unsuccessful search. Calculate the load factor on the table and use it to calculate the predicted number of cells that must be examined during a successful search, and also during an unsuccessful search. Do the predicted values match your actual values? If not, why not?

(c) Do the problem in part (a) again, but this time use quadratic probing instead of linear probing. Then calculate the average number of cells that must be examined during a successful search for an object in the table you produce. Also calculate the average number of cells that must be examined during an unsuccessful search.

(d) Do the problem in part (a) again, but this time use separate chaining instead of linear probing. Then calculate the average number of nodes that must be examined during a successful search for an object in the table you produce. Also calculate the average number of nodes that must be examined during an unsuccessful search. Calculate the predicated number of nodes that must be examined during a successful search, and also during an unsuccessful search. Compare with the actual values.

Solutions for Odd Numbered Exercises

5-1. Function 1: The sum of the ASCII values for the letters in TOP is 84+79+80 = 243 . When 243 is divided by 10, the remainder is 3 , so TOP hashes to address 3 . The same is true for POT and OPT because they use exactly the same letters.

Function 2: 3(84) + 13(79) + 23(80) = 3119; when 3119 is divided by 10 , the remainder is 9 , so TOP is hashed to address 9 . 3(80) + 13(79) + 23(84) = 3199 ; when 3199 is divided by 10 , the remainder is 9 , so POT is hashed to address 9 . 3(79) + 13(80) + 23(84) = 3209; when 3209 is divided by 10 , the remainder is 9 , so OPT is hashed to address 9 .

Function 3: The product of the ASCII values for the letters in TOP is 84*79*80 = 530880 . When this product is divided by 10 the remainder is 0 , so TOP hashes to address 0 . The same is true for POT and OPT because they use exactly the same letters.

Function 4: To calculate the hash address for TOP , we first form the Exclusive OR of the binary representations of T and O : 01010100 ^ 01001111 = 00011011 ; then we form the Exclusive OR of the result with the binary representation of P: 00011011 ^ 01010000 = 01001011 .

The binary number 01001011 represents the integer 64 + 8 + 2 + 1 = 75 . When this is divided by 10 the remainder is 5 , so TOP hashes to address 5 .

To hash POT, we form 01010000 ^ 01001111 = 00011111 ; then 00011111 ^ 01010100 = 01001011 ; this is the same result as for TOP, so POT also hashes to address 5 .

To hash OPT, we form 01001111 ^ 01010000 = 00011111 ; then 00011111 ^ 01010100 = 01001011; this is the same result as for TOP, so OPT also hashes to address 5 .

5-5. (a) Inserting the objects in the order they were given produces this table:

0	1	2	3	4	5	6	7
TAR	RAT	BAR	APT	PAT	BAT		ART

When BAR is accessed, 4 cells will have to be examined (namely 7, 0, 1, and 2).

When ART is accessed, 1 cell will have to be examined (namely 7).

When RAT is accessed, 1 cell will have to be examined (namely 1).

When TAR is accessed, 1 cell will have to be examined (namely 0).

When BAT is accessed, 1 cell will have to be examined (namely 5).

When APT is accessed, 2 cells will have to be examined (namely 2 and 3).

When PAT is accessed, 6 cells will have to be examined (namely 7, 0, 1, 2, 3, and 4).

The average number of cells that will be examined during a successful access operation in this particular table (after its 7 strings have been inserted) is $\frac{4+1+1+1+1+2+6}{7} = \frac{16}{7} \approx 2.3$.

(b) The load factor for the filled table in part (a) is $\lambda = \frac{7}{8}$. Theory predicts that the average number of cells that will have to be examined during a successful search is approximately $\frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right)$ provided λ is not near 1 . Unfortunately, our λ is close to 1 , so this approximation cannot be

expected to be very good. It predicts as the average number of cells examined in a successful search the value $\frac{1}{2}\left(1 + \frac{1}{1 - 7/8}\right) = \frac{1}{2}(1 + 8) = 4.5$. NOTE: even if the value of λ were .5 or less, however, the *predicted* average number of cells examined does not have to match the computed average number of cells examined in any particular table. The *predicted* average comes from looking at *all possible tables*. Any particular table may deviate in behavior from what is predicted.

(c)

If the key hashes to address 0, then 7 cells will have to be examined in order to discover failure.
If the key hashes to address 1, then 6 cells will have to be examined in order to discover failure.
If the key hashes to address 2, then 5 cells will have to be examined in order to discover failure.
If the key hashes to address 3, then 4 cells will have to be examined in order to discover failure.
If the key hashes to address 4, then 3 cells will have to be examined in order to discover failure.
If the key hashes to address 5, then 2 cells will have to be examined in order to discover failure.
If the key hashes to address 6, then 1 cells will have to be examined in order to discover failure.
If the key hashes to address 7, then 8 cells will have to be examined in order to discover failure.

The average number of cells examined in an unsuccessful access is $\frac{7 + 6 + 5 + 4 + 3 + 2 + 1 + 8}{8}$ $= \frac{36}{8} = 4.5$. The predicted average is approximately $\frac{1}{2}\left(1 + \frac{1}{(1 - \lambda)^2}\right)$, but the approximation is poor if λ is close to 1. In our case, $\lambda = 7/8$ is close to 1, so the approximate predicted value of $\frac{1}{2}(1 + 64) = 32.5$ makes no sense at all because you would never examine 32 cells during a search of an array of size 8.

(d)

0 ☐ TAR | /
1 ☐ RAT | /
2 ☐ APT | /
3 ☐
4 ☐
5 ☐ BAT | /
6 ☐
7 ☐ PAT | BAR | ART | /

When BAR is accessed, 2 nodes will have to be examined.
When RAT is accessed, 1 node will have to be examined.
When APT is accessed, 1 node will have to be examined.
When BAT is accessed, 1 node will have to be examined.
When PAT is accessed, 1 node will have to be examined.
When ART is accessed, 3 nodes will have to be examined.
When TAR is accessed, 1 node will have to be examined.

The average number of nodes examined during a successful access is $\frac{2+1+1+1+1+3+1}{7}$ $= \frac{10}{7} = 1.43$.

The predicted average is $1+\frac{\lambda}{2} = 1+\frac{7/8}{2} = \frac{23}{16} = 1.44$. This is close to the actual average in this particular table, but in general, the actual average in a particular table may deviate quite a bit from the predicted average, which is an average over all tables.

(e)

If the key hashes to address 0 , then 1 node will have to be examined to discover failure.
If the key hashes to address 1 , then 1 node will have to be examined to discover failure.
If the key hashes to address 2 , then 1 node will have to be examined to discover failure.
If the key hashes to address 3 , then 0 nodes will have to be examined to discover failure.
If the key hashes to address 4 , then 0 nodes will have to be examined to discover failure.
If the key hashes to address 5 , then 1 node will have to be examined to discover failure.
If the key hashes to address 6 , then 0 nodes will have to be examined to discover failure.
If the key hashes to address 7 , then 3 nodes will have to be examined to discover failure.

The average number of nodes examined during a failed access is $\frac{1+1+1+0+0+1+0+3}{8} = \frac{7}{8}$.

The predicted average is approximately $e^{-\lambda}+\lambda \approx 1.3$. Our table performs better than predicted.

6. Tail Recursion

When a function makes a recursive call, we say that the call is a "tail recursion" if it is the last action in the function before the return. Generally speaking, if all the recursive calls in a function are tail recursions, then they should be removed because a more efficient "iterative" version of the function (a version that uses a loop) will be very easy to write. Let's look at some examples.

Example 1. Recall the Selection Sort Algorithm. When given an array `a[1..n]` to sort, it locates the largest element, swaps it to the last position of the unsorted array, and then repeats this process until the unsorted portion of the array has shrunk to nothing. Here is a recursive version.

```
template <class otype>
void recursiveSelectionSort (otype a[], int n)    //sort a[1..n]
{
    if (n <= 1)   // base case; no sorting is necessary
        return;
    else
    {
        int bestLocation = 1;              // find the cell in a[1..n]
        for (int j = 2; j <= n; ++j)       // that contains the largest
            if (a[j] > a[bestLocation])    // data object
                bestLocation = j;

        swap (a[bestLocation], a[n]);      // put largest object in a[n]

        recursiveSelectionSort (a, n-1);  // sort a[1..n-1]
    }
} // recursiveSelectionSort()
```

Example 2. Here is a recursive version of binary search.

```
template <class data>
void recursiveBinarySearch (const data a[], const data &target,
                            int first, int last, bool &found, int & sub)
{
    if (last < first)     // Don't search an empty array.
    {
        found = false;
        sub   = first;
    }

    else
    {
        int mid = (first + last)/2;

        if (target < a[mid])  // the following call sets found, sub
            recursiveBinarySearch (a, target, first, mid-1, found, sub);

        else if (a[mid] < target)
            recursiveBinarySearch (a, target, mid+1, last, found, sub);

        else          // only remaining possibility is a[mid] == target
        {
            sub = mid;
            found = true;
        }
    }
} // recursiveBinarySearch()
```

Example 3. Factorials can be calculated recursively.

```
unsigned long factorial (unsigned long n)
{
    if (n <= 1)
        return 1;
    else
        return n * factorial (n - 1);
} // factorial()
```

In Example 3, the recursive call is not quite the last action in the function before the function returns, but even so, it is very easy to re-write the factorial function so that it is iterative (uses a "for" loop).

A **list** can be thought of as a recursively defined structure, just like a binary tree. If you think about it, you see that a list is either

(1) an empty list (signaled by a NULL pointer), or

(2) a node followed by a list (to which the node points).

Once you see this, it is tempting to consider writing some list functions recursively.

Example 4: Here is a recursive function that takes a list as a parameter and returns a (deep) copy of the list.

```
NodePtr copy (NodePtr list)
{
    if (!list)
        return NULL;
    else
    {
        NodePtr temp = new node;
        test_allocation (temp);
        temp->datum = list->datum;
        temp->link = copy (list->link);
        return temp;
    }
} // copy()
```

This is another example in which we almost -- but don't quite -- have tail recursion. The code here is somewhat simpler than the code for an iterative version, but there is a strong reason for not using recursion in this case: a different stack frame must be created for every node of the list being copied. If the list is long, the run-time stack could overflow, especially since the new nodes that are allocated for the new list must come from the same general area of memory (recall that the run-time stack and run-time heap share a large empty section of memory). The iterative version of the function is therefore more "robust" (less likely to run into limitations imposed by the machine on which it runs).

Exercise

6-2. Most modern data encryption systems, such as the RSA public key system, involve performing arithmetic with integers having several hundred digits. Integers of this magnitude have to be implemented in software instead of hardware. For example, the digits of such integers may be stored in large arrays or linked lists, and the operations of addition, subtraction, multiplication, division, and exponentiation have to be written by programmers. In some C201 classes in our department, students implement "unlimited integers" and the necessary arithmetic functions on those integers.

Exponentiation involving integers raised to gigantic powers plays a critical role in data encryption. There are two ways we can perform exponentiation in software, and both can be expressed recursively. One of the methods uses the following recursive definition of a non-zero integer n raised to a non-negative integer power p :

$$n^p = \begin{cases} 1 & \text{if } p \text{ is } 0 \\ n * n^{p-1} & \text{if } p > 0 \end{cases}$$

For example, $n^4 = n * n^3$, $n^3 = n * n^2$, $n^2 = n * n^1$, $n^1 = n * n^0 = n * 1 = n$.

This can be coded in a recursive function as follows:

```
UnlimitedInt exponential (UnlimitedInt n, UnlimitedInt p)
{
    if (p == 0)
        return 1;  // The 1 will be type cast to UnlimitedInt.
    else
        return n * exponential(n, p-1); //Uses overloaded operators.
} // exponential()
```

The other method for performing exponentiation uses this alternative recursive definition:

$$n^p = \begin{cases} 1 & \text{if } p \text{ is } 0 \\ n^{p/2} * n^{p/2} & \text{if } p \text{ is even and } > 0 \\ n * np/2 * np/2 & \text{if } p \text{ is odd, where } p/2 \text{ indicates integer division in which the remainder is discarded} \end{cases}$$

For example, $n^{10} = n^5 * n^5$, $n^5 = n * n^2 * n^2$, $n^2 = n^1 * n^1$, $n^1 = n * n^0 * n^0 = n * 1 * 1 = n$.

This can be coded in a recursive function as follows:

[continued on the following page]

```
UnlimitedInt exponential (UnlimitedInt n, UnlimitedInt p)
{
    if (p == 0)
        return 1;  // The 1 will be type cast to UnlimitedInt.
    else
    {
        UnlimitedInt factor = exponential(n, p/2);

        if (p % 2 == 0)  // if p is even
            return factor * factor;
        else
            return n * factor * factor;
    }
} // exponential ()
```

(a) Both versions of the exponential function have been written recursively, but in one case, the recursion is *almost* an instance of tail recursion, and it can easily be rewritten in an iterative form. Determine which of the versions it is, and rewrite it so that it is iterative instead of recursive.

(b) In your iterative version of the exponential function, how many multiplications will be performed? Express your answer in terms of p .

(c) In the recursive version of the exponential function that you did not rewrite iteratively, approximately how many *recursive* function calls will have to be made to reach the base case. Express your answer in terms of the initial value of p . (The *recursive* function calls are the calls that a function makes to itself.)

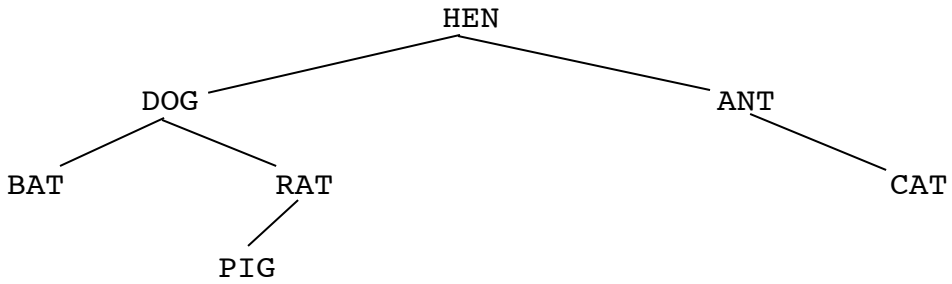
(d) Which of these two exponential functions do you think is used in data encryption? Explain.

7. Binary Trees

Definition: A ***binary tree*** is a finite set, call it T, that's either empty or else has its elements partitioned into three distinguished subsets: a singleton subset containing the ***root*** of T, a ***left subtree***, which is itself a binary tree, and a ***right subtree***, which is also a binary tree. The objects that make up the set T are called the ***nodes*** of the tree T. If x and y are nodes such that y is the root of the left subtree of the binary tree rooted at x, then we say that y is the ***left child*** of x, and that x is the ***parent*** of y. ***Right child*** is defined similarly. Two nodes are ***siblings*** if and only if they have the same parent. The ***descendants*** of a node x are the nodes that make up the left and right subtrees of x. The ***ancestors*** of x consist of the parent of x (if any), the parent of that parent (i.e., the ***grandparent*** of x), the parent of the grandparent, and so on. [Note: in some texts, a node x is considered to be both an *ancestor* and a *descendent of itself*! We shall not adopt that definition in this course.] The root of the tree T has no parent. A node x is called a ***leaf*** if and only if x has no children (i.e., both the left and the right subtrees of x are empty).

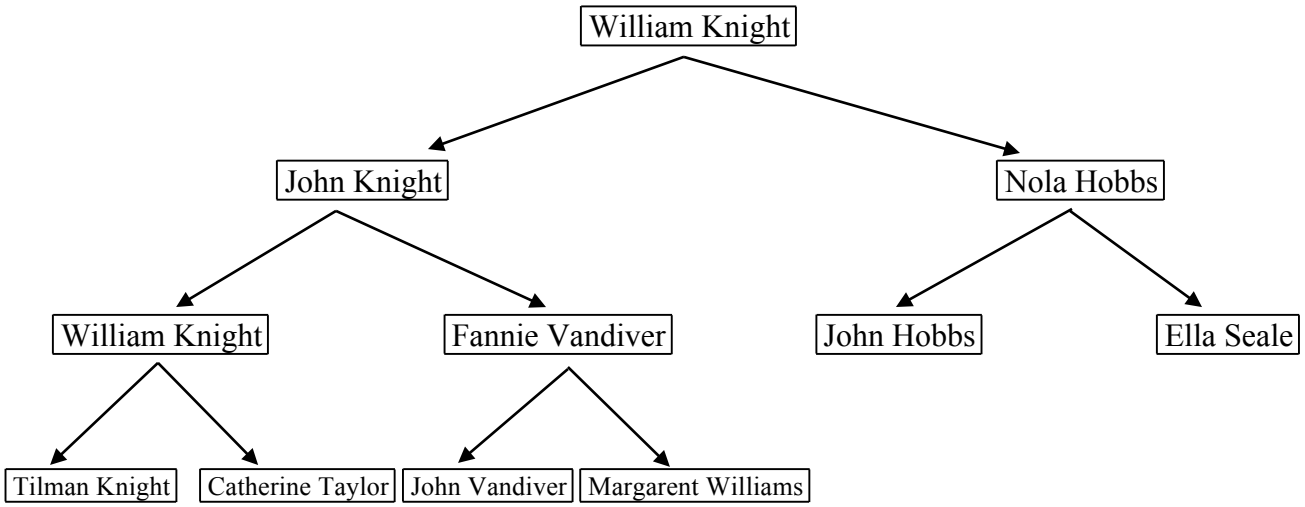
Note that the definition of "binary tree" is ***recursive***, by which we mean that the defining sentence for "binary tree" uses the phrase that's being defined ("binary tree"). Recursive definitions are very common in computer science. Note also that the definition is "conceptual", not specifically computer oriented. We can think of a "binary tree" as an abstract data structure (also called an abstract data type, or ADT).

Example. Here is a drawing of a not-very-interesting binary tree:



The nodes of this tree are 3-letter character strings. The root of the entire tree is HEN. The left subtree of the tree contains the nodes DOG, BAT, RAT, and PIG, and these four nodes form the left subtree of HEN. The root of this left subtree is DOG, so DOG is the left child of HEN. The right subtree of the entire tree (i.e., the right subtree of HEN) contains the nodes ANT and CAT. The root of this right subtree is ANT, so ANT is the right child of HEN. The left subtree of DOG contains only one node, namely BAT, which is the left child of DOG. The right subtree of DOG contains two nodes, namely RAT and PIG. The node ANT has an empty left subtree, so it has no left child. The right child of ANT is CAT. The parent of PIG is RAT. The node HEN has no parent. The ancestors of PIG are RAT, DOG, and HEN. The descendants of DOG are BAT, RAT, and PIG. The nodes DOG and ANT are siblings. Similarly, RAT is a sibling of BAT. The nodes CAT, PIG, and HEN have no siblings. The leaves of this tree are BAT, PIG, and CAT.

Example: Your family tree can be regarded as a binary tree. Here is part of the family tree for a certain William Knight



Note, however, that the parent-child terminology introduced on the preceding page for abstract binary trees is exactly the opposite of the parent-child relationships that are expressed by a family tree. So when the tree above is viewed as a binary tree, then Nola Hobbs is the parent of Ella Seale, even though Ella was actually the mother of Nola.

Definition: The ***height*** of a binary tree T is defined to be -1 if T is empty; otherwise it is defined to be 1 greater than the larger of the heights of the two subtrees of T (if the subtrees are of equal height, then the "larger" height is understood to be their common height). The root of T is said to be **at level 0** or to **have 0 as its level number**; for any other node in T, the ***level number*** of the node is 1 greater than the level number of the node's parent.

What does this definition tell us about the height of a binary tree that contains only one node?
What is the height of a binary tree with two nodes?
What are the two possible heights of a binary tree with three nodes?
How can the height of the tree be defined in terms of the level numbers of the nodes in the tree?

7.1. Programming Implementations of Binary Trees

Suppose you want to implement a binary tree in a computer program. One way to accomplish this is to use an array (this was discovered long ago by genealogists). The root of the tree is stored in location 1. Its left child (if any) is stored in location 2 and its right child (if any) in location 3. In general, the left and right child of an object at location k are stored at locations $2k$ and $2k+1$ respectively. Where there are "gaps" in the tree, some special object must be stored in a cell to indicate "no object is stored here".

An alternative method for implementing a binary tree is to create a "Node" data type that can carry the objects you want to store in the nodes and that also contains two pointers to other nodes. For example, here is a definition of "node" that allows us to store strings of some bounded length.

```
const int MAXLENGTH = 50;

struct Node
{
    char  label[MAXLENGTH + 1]; // The "label" of a Node is a string.
    Node  *left;      // Pointer to the left child
    Node  *right;     // Pointer to the right child
};

typedef Node *NodePtr;

NodePtr rootp;  // A variable that will point to the root.
```

An empty tree can be signified by giving the `rootp` pointer the value NULL. You could construct the first three levels of the tree in Example 1 by using the following bit of code:

```
rootp = new Node;  // Should check to make sure rootp is not NULL
strcpy(rootp->label, "HEN");
rootp->left = new Node; // Should check allocation.
strcpy(rootp->left->label, "DOG");
rootp->left->left = new Node;
strcpy(rootp->left->left->label, "ANT");
rootp->left->left->left = rootp->left->left->right = NULL;
rootp->left->right = new Node;
strcpy(rootp->left->right->label, "RAT");
    etc.
```

Clearly it would be very tedious indeed to use this method to build a large binary tree. Later we'll see better ways for building trees. Until we do that, let's just assume that large trees can be built, and let's look at some functions that perform various operations on binary trees.

Problem 1: Write a function that counts the number of nodes in a binary tree and returns that number as the value of the function. The function will be passed a single argument, namely a pointer to the root of the tree whose "weight" (number of nodes) we want to know.

Problem 2: Write a function that prints out, in a column, all the nodes (more precisely, all the strings stored in the nodes) of a binary tree. The order in which the nodes are to be printed is as follows: no node must be printed until all the nodes in its left subtree have been printed, and it must be printed before any node in its right subtree has been printed.

Solutions for Problem 1:

The following solution uses a stack to keep track of the path by which we have descended from the root of the given tree down to any particular node. The stack is popped whenever we need to "climb" back up the tree by one or more levels.

```
int weight(NodePtr rootp)     // rootp points to a binary tree
{
    int nodeCount = 0;
    stackOf<NodePtr> s;  // will hold pointers to nodes on paths
    NodePtr p = rootp;     // will move around the binary tree

    while (p != NULL) // go as far down to the left as possible
    {
        ++nodeCount; // tally the node we have reached for 1st time
        s += p;      // push a pointer to the current node on stack
        p = p->left; // go down to the left subtree of current node
    }

    while (s)   // while the stack of pointers is not empty
    {
        p = --s;   // pop stack; p points to a node previously seen
        p = p->right;  // go down to the right subtree of current
                       // node
        while (p != NULL) // go as far down to the left as possible
        {
            ++nodeCount; // tally node we have reached for 1st time
            s += p;      // push a pointer to current node on stack
            p = p->left; // go down to left subtree of current node
        }
    }

    return nodeCount;
} // weight()
```

The running time for this function is $\Theta(N)$, where N is the number of nodes in the tree. The reason for this is that a pointer to each node is pushed and later popped exactly once.

Does this function handle the case of an empty tree correctly? _____

The following solution uses recursion to count the number of nodes in the tree. Its structure is based on the fact that the definition of a binary tree is "recursive"; that is, a binary tree is defined in terms of other binary trees.

```
int weight(NodePtr rootp)     // rootp points to a binary tree
{
    if (rootp == NULL)  // base case: the binary tree is empty
        return 0;
    else
    {
        int leftWeight  = weight (rootp->left);  // recursive call
        int rightWeight = weight (rootp->right); // recursive call
        return (1 + leftWeight + rightWeight);
    }
} // weight()
```

The running time for the recursive function is also $\Theta(N)$, where N is the number of nodes in the binary tree. How can we see that this is true? The function is called exactly once for every node in the tree and once for every empty subtreee in the tree. There are $N+1$ empty subtrees in a tree with N nodes, so the total number of calls is $2N + 1$. Each call takes some minimum amount of time A and some maximum time B, so the running time is bounded below by $A(2N + 1)$ and above by $B(2N + 1)$.

How does the recursive function `weight`, which calls itself, avoid getting confused about what the values of "`leftWeight`" and "`rightWeight`" are when there have been calls within calls within calls....? The answer requires an understanding of something called ***stack frames***, or ***activation records***, for function calls, which forces us to talk about the program at a level below the C/C++ code. That is, we need to descend to the machine code level and see what is happening there.

When a C/C++ program is set up by the operating system so that the program can run, the operating system sets aside temporarily a segment the computer's main memory. The program will run in that segment. The segment has space for the machine code (instructions) of the program, storage space for all global data, and two large empty spaces, one for the ***run-time stack*** and the other for the ***run-time heap***. The run-time heap is the space from which the "`new`" function allocates new memory space whenever your program requests a dynamic memory allocation. The run-time stack is a stack onto which a stack frame is pushed *every time a function call is made during execution of the program*. In fact, since the function "`main`" is the first function called in a C/C++ program, a stack frame is constructed for it when the program begins to run, and that stack frame is pushed onto the run-time stack. (Actually, it is constructed right on the top of the empty run-time stack.)

What does a stack frame contain? The most critical thing it holds is the "return address" of the place in memory to which execution must be transferred when the function returns. (For "`main`", this will be the address of an instruction in the operating system's code.) The stack frame also contains a space for each of the local variables that's declared in the function. In addition, it contains a space for the value of (or reference to) each of the arguments passed to the function in the parameter list. Finally, if the function does not return `void`, then there is a space to hold the value being returned by the function.

When a function call occurs during execution of a program, then after the stack frame has been set up for that function call, all actions involving variables within that function, whether local

variables or parameters, take place on the variables in the stack frame. (The only exception to this is that when the value of a reference parameter is changed, that change is done by following one or more pointers to the actual argument aliased by the reference parameter.) When the function executes a return instruction, the return address in the stack frame is used to divert execution back to the calling function, and the stack frame for the function that has ceased to execute is considered to have been popped off the stack.

It is particularly important, when studying induction, to understand that each successive call to the function produces a separate instance of the stack frame, so that each call has its own separate set of local variables. That is, when you see a local variable in the source code for a function, you have to understand that during a sequence of recursive calls there will be many different "versions" of that local variable.

Here another version of the `weight` function. At first it may appear to be more efficient in both space required and execution time because there is no local variable and no assignment statement (cf. the preceding version). In fact, however, the compiler will have to set up variables in which to hold the values returned by the function calls, so there is no actual saving. This new version is a little more elegant, however.

`// Version 2` This version does away with the need for two local variables.

```
int weight(NodePtr rootp)     // rootp points to a binary tree
{
    if (!rootp)     // base case: the binary tree is empty
        return 0;   // EXIT FROM FUNCTION
    else
        return ( 1 + weight(rootp->left) + weight(rootp->right) );
} // weight()
```

Solution for Problem 2 (see page 3):

```
void printLabels (NodePtr rootp)  // VERSION 1
{
    if (!rootp)     // base case: tree is empty
        return;

    else
    {
        printLabels (rootp->left);

        cout << rootp->label << endl;

        printLabels (rootp->right);
    }
} // printLabels()
```

In what order would this function print the labels of the tree nodes in the "animals" binary tree on page 1? (This is called the ***symmetric order*** print-out of the nodes.)

The following version of `printLabels` uses the fact that no action (other than returning from the function call) needs to be taken if the argument passed to the function is NULL. The "return" in this version is "tacit" (not explicitly stated).

```
void printLabels (NodePtr rootp)  // VERSION 2
{
    if (rootp)          // base case is "tacit"
    {
        printLabels (rootp->left);

        cout << rootp->label << endl;

        printLabels (rootp->right);
    }
} // printLabels()
```

This compact way of writing the code does *not* make execution of the function any more efficient. Exactly the same number of tests will be made to determine whether the pointer parameter is NULL, and exactly the same function calls will take place.

Problem 3. The two `printLabels` functions that we have written print the node labels in a column against the left margin of the screen. It is not possible to examine this list and reconstruct the tree from which it is printed. All the "structure" of the tree has been suppressed. So now let's write a tree-print function that displays the labels of a binary tree in such a way that by inspection we can determine the structure of the tree. We'll print the tree "sideways", as if it had been rotated 90 degrees counter-clockwise. The animals tree on page 1, for example, will be printed like this:

```
                CAT
          ANT
     HEN
                RAT
                     PIG
          DOG
                BAT
```

Let's call our function `prettyPrint`. We'll pass it two arguments: a pointer to the root of the tree that we want to print and an "indentation number", i.e., a number that tells how many spaces away from the left margin of the screen we want the root of the tree to be printed. We'll use a default value of `0` for the indentation number. A prototype for the function could look like this:

```
void prettyPrint (NodePtr rootp, int indentation = 0);

void prettyPrint (NodePtr rootp, int indentation) // default value 0
{
    const int INDENT = 5; // Each tree level is indented 5 spaces
                          // beyond its parent's level

    if (rootp)
    {
        prettyPrint(rootp->right, indentation + INDENT);

        for (int i = 1; i <= indentation; ++i)
            cout << " ";  // This loop prints the appropriate
                          // number of blank spaces

        cout << rootp->label << endl;

        prettyPrint(rootp->left, indentation + INDENT);
    }
} // prettyPrint()
```

Problem 4. Next let's write a function that can examine two separate binary trees and determine whether they have exactly the same structure with exactly the same labels. The function should return `true` if the trees are exactly alike, and `false` otherwise.

```
bool areAlike (NodePtr first, NodePtr second)
{
    if (!first && !second) // both trees are empty
        return true;

    else if ( !first  ||  !second )
        return false;  // one tree is empty, the other is not

    else if ( (strcmp(first->label, second->label) == 0) // labels same
         && areAlike(first->left, second->left) // left subtrees alike
         && areAlike(first->right, second->right)) // right are alike
        return true;

    else
        return false;
} // areAlike()
```

The first "`else`" in the function takes advantage of the fact that if execution reaches that test, then it has already been established that the trees are *not* both empty, so at least one of them

must be non-empty. Thus if the first is empty (`!first`) then the second must be non-empty -- hence different from the first; and vice versa. If execution reaches the second "`else`", then it has already been established that *neither* of the trees is empty, so it is safe to derefence both `first` and `second` (if either were NULL, that would crash the function).

Problem 5: Write a function that prints out, in a column, all the nodes of a binary tree. The order in which the nodes are to be printed is as follows: each node in the tree is to be printed before any of the nodes in its subtrees, and all the nodes in its left subtree must all be printed before any node in its right subtree is printed. (This will be called the ***pre-order*** print-out of the nodes in the tree.)

Solution to Problem 5:

```
void printLabelsInPreOrder (NodePtr rootp)  // VERSION 1
{
    if (!rootp)    // base case: tree is empty
        return;
    else
    {
        cout << rootp->label << endl;
        printLabelsInPreOrder (rootp->left);
        printLabelsInPreOrder (rootp->right);
    }
} // printLabelsInPreOrder()
```

Here is a version in which the return is tacit.

```
void printLabelsInPreOrder (NodePtr rootp)   // VERSION 2
{
    if (rootp)
    {
        cout << rootp->label << endl;
        printLabelsInPreOrder (rootp->left);
        printLabelsInPreOrder (rootp->right);
    }
} // printLabelsInPreOrder()
```

In what order will the nodes of the "animals" tree on page 1 be printed by the "`printLabelsInPreOrder`" function?

Problem 6. There is a printing strategy that makes a ***post-order traversal*** of a tree. The idea is that each root label is printed *after* ("post") the labels in its left and right subtrees have been printed.

```
void printLabelsInPostOrder (NodePtr rootp)
{
    if (rootp)
    {
        printLabelsInPostOrder (rootp->left);
        printLabelsInPostOrder (rootp->right);
        cout << rootp->label << endl;
    }
} // printLabelsInPostOrder()
```

In what order will the function above print out the nodes in the "animal tree" on page 1?

It's worth noting that no matter whether the pre-order traversal, the symmetric order traversal, or the post-order traversal is executed, the order in which the activation records go on the run-time stack is the same. Pointers to the nodes along the left side of the tree are stacked before pointers to nodes in the center and the right side of the tree.

More Classroom Exercises

In each exercise below you are asked to write a function to perform some operation on a binary tree. Assume that the binary trees to which these functions will be applied consist of nodes implemented as objects of the following structure data type:

```
struct Node
{
    int  datum; //NOTE that the labels are integers in these
                // exercises
    Node *left,
    Node *right;
};
```

1. Write a function that returns the sum of all the `"datum"` integers in a binary tree.

2. Write a function that returns an exact copy of a binary tree. That is, it returns a pointer to a newly constructed tree that's an exact copy of the original tree.

3. Write a function that replaces a binary tree by its mirror image.

4. Write a function that searches a binary tree for a node having a specified datum value. The function should return a pointer to the node it finds -- if it finds one. Otherwise, it should return the value `NULL`. (Note: the binary tree is NOT assumed to be a binary *search* tree.)

5. Write a function that returns the height of a binary tree.

6. Assuming that the nodes that make up a binary tree have been dynamically allocated from the run time heap using the C++ function "new", write a function that properly destroys a binary tree by deallocating all its nodes (i.e., it returns all the nodes to the run-time heap), and that sets to NULL the pointer that was pointing to the root of the tree.

7. Write a function that returns the number of leaves in a binary tree.

8. Write a function that returns a pointer to a node with the largest `datum` value in the tree. (Do not assume that the tree is a binary *search* tree.) Return NULL if the tree is empty.

9. An "only child" is a child who has no siblings (i.e., no brothers or sisters). In a binary tree, an "only child" is a non-root node that has no sibling. In a binary tree with only 1 node, there is no only child (remember, the root is never counted as an only child), but in a binary tree with 2 nodes, the non-root node *is* an only child. In a binary tree with exactly 3 nodes, there may be no only children (if the height of the tree is only 1) or exactly 2 only children (if the height of the tree is 2). Write a function that returns the number of only children in a binary tree.

10. A node in a binary tree is a "grandparent" if and only if it has a child that has a child. Write a function that returns the number of grandparents in a binary tree.

11. A binary tree is called "perfect" if and only if all its empty trees are at the same level in the tree. You can verify that there is a perfect tree with just one node, another with 3 nodes, another with 7 nodes, another with 15 nodes, and so on. We also say that the empty tree is "perfect". Write a function that examines a binary tree to determine whether it is a perfect tree. [Note: this is a harder exercise than the preceding ones.]

1.
```
int sumOfDatumValues (NodePtr rootp)
{
    if (!rootp)  // equivalent to "if (rootp == NULL)"
        return 0;  // base case:  tree is empty

    else
        return  rootp->datum + sumOfDatum_values(rootp->left)
                             + sumOfDatum_values(rootp->right);
} // sumOfDatumValues()
```

2.
```
NodePtr copy (NodePtr tp) // For variety let's use "tp" for pointer to tree.
{
    if (!tp)         // if (tp == NULL)
        return NULL;   // return NULL pointer value to indicate empty tree

    else
    {
        NodePtr temp;
        temp = new Node;  // Really ought to check whether "new" succeeded
        temp->datum = tp->datum;
        temp->left  = copy (tp->left);
        temp->right = copy (tp->right);
        return temp;
    }
} // copy()
```

3.
```
void reverse (NodePtr tp)   // tp points to a tree; it's a value parameter
{
    if (tp)   // if (tp != NULL)
    {
        NodePtr temp;
        temp = tp->left;
        tp->left = tp->right;
        tp->right = temp;
        reverse (tp->left);
        reverse (tp->right);
    }
} // reverse()
```

4.
```
NodePtr location (int k, NodePtr tp)  // k is the specified datum value
{
    if (!tp)         // if (tp == NULL)     One base case:  tree is empty.
        return NULL;

    else if (tp->datum == k)               Another base case:  k is the root.
        return tp;

    else
    {
        NodePtr temp;

        temp = location (k, tp->left); // If k not at root,

                                       // search left subtree.

        if (temp)    // If temp is not NULL, then k was found in the

                     // left subtree.
          return temp;
        else
            return location (k, tp->right);     // Otherwise,
                                                // try the right subtree.
    }
} // location()
```

5.
```
int height (NodePtr t)    // t is a pointer to a binary tree
{
    int  leftHeight, rightHeight;

    if (!t)          // if (t == NULL)    Base case:  tree is empty
        return -1;   // The height of the empty tree is -1 by convention.

    else
    {
        leftHeight  = height (t->left);
        rightHeight = height (t->right);

        if (leftHeight >= rightHeight)
            return 1 + leftHeight;
        else
            return 1 + rightHeight;
    }
} // height()
```

6.
```
void destroy (NodePtr &t)  // a reference parameter must be used here
{
    if (t) // if t != NULL
    {
        destroy (t->left);  // Recursive calls on left and right subtrees.
        destroy (t->right);
        delete (t);  // This deallocates the node pointed to by t, but does
        t = NULL;    // not make t NULL; that must be done by a separate
    }                // instruction
} // destroy()
```

7.
```
int numberOfLeaves (NodePtr t)
{
    if (!t) // if (t == NULL)    One base case:  empty tree.
        return 0;

    else if (! t->left && ! t->right) // Another base case:  the root has
        return 1;        // no left child or right child, so it is a leaf.

    else
        return numberOfLeaves (t->left) + numberOfLeaves (t->right);
} // numberOfLeaves()
```

8. See next page.

9. The trick to doing this problem is to have the only children counted by their parents. That is, each node looks to its left and right and decides whether it has an only child.

```
int numberOfOnlyChildren (NodePtr t)
{
    if (!t || (!t->left && !t->right))  // if tree is empty or consists of just
        return 0;                       // one node, then it has no only children

    else if (!t->left)   // then t->right is not null, so the root of the right
                         // subtree is an only child
        return 1 + numberOfOnlyChildren (t->right);

    else if (!t->right)
        return 1 + numberOfOnlyChildren (t->left);

    else
        return numberOfOnlyChildren (t->left)
             + numberOfOnlyChildren (t->right);
} // numberOfOnlyChildren()
```

8.
```
NodePtr locationMaxDatum (NodePtr t)
{
    NodePtr subtreeMaxLocation, treeMaxLocation;

    if (!t)
        return NULL;

    else
    {
        treeMaxLocation = t;  // Root contains largest key seen so far.

        if (t->left)  // if the left subtree is non-empty,
        {             // find its maximum key
            subtreeMaxLocation = locationMaxDatum (t->left);

            if (subtreeMaxLocation->datum > treeMaxLocation->datum)
                treeMaxLocation = subtreeMaxLocation;
        }

        if (t->right) // if the right subtree is non-empty,
        {             // find its maximum key
            subtreeMaxLocation = locationMaxDatum (t->right);

            if (subtreeMaxLocation->datum > treeMaxLocation->datum)
                treeMaxLocation = subtreeMaxLocation;
        }

        return treeMaxLocation;
    }
} // locationMaxDatum()
```

9. See preceding page.

10.
```
int numberOfGrandparents (NodePtr t)
{
    if (!t || (!t->left && !t->right))
        return 0;

    else if (!t->left)    // then t->right is not NULL
    {
        if (!t->right->left && !t->right->right)
            return 0;
        else
            return 1 + numberOfGrandparents (t->right);
    }

    else if (!t->right)   // then t->left is not NULL
    {
        if (!t->left->left && !t->left->right)
            return 0;

        return 1 + numberOfGrandparents (t->left);
    }

    else if (!t->left->left && !t->left->right && !t->right->left
                                               && !t->right->right)
        return 0;

    else
        return 1 + numberOfGrandparents (t->left)
                 + numberOfGrandparents (t->right);
} // numberOfGrandparents()
```

11. We begin by thinking about what the heading of the function should look like. Presumably we want to return a boolean value (`true` or `false`), depending on whether the specified tree is perfect or not. Thus the heading ought to look like this:

```
bool isPerfect (NodePtr t)
```

Now how will we *recursively* determine whether a tree is perfect? It will be perfect if and only if it is empty or its left and right subtrees are perfect *and have the same height*. Since we have already written a recursive `height` function for binary trees, it is *possible* to write the solution as follows:

```
bool isPerfect (NodePtr t)      // INEFFICIENT SOLUTION!!!
{
    if (!t)
        return true;
    else
        return  ( isPerfect (t->left)  &&  isPerfect (t->right)
                  &&  height (t->left) == height (t->right) );
} // isPerfect()
```

Although the code above works correctly, it is an ***unacceptable*** solution because it is inefficient. The reason it is inefficient is that at every node N in the tree it calls the `height` function, which recursively calculates the height of the subtree rooted at N. Each call to the `height` function traverses pieces of the tree that have already been traversed over and over again. In C455 (Analysis of Algorithms) you will be asked to calculate just how inefficient the function above can be.

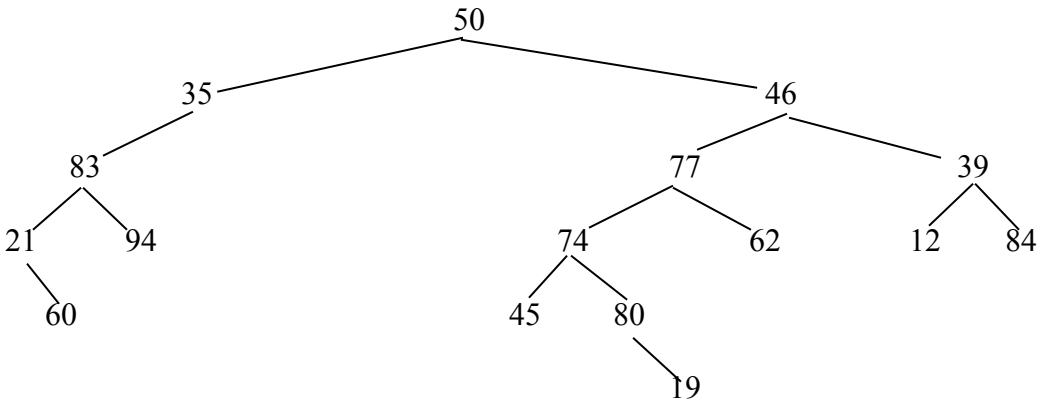
A more enlightened approach is to recognize that at the same time that the function is determining whether it is perfect, it can also calculate its own height and pass this number upward to its parent. Unfortunately the function heading given above does not provide any way to return the height of the given tree. Here is the solution to that difficulty. We write two functions, one of which will "drive" the other. The "driver" will have the heading shown above. It will be non-recursive. It will call an "auxiliary" function that *will* be recursive. The auxiliary fuction will not only return information about whether the tree it is given is perfect, but it will also return the height of the tree.

The full solution is given on the following page.

```
bool isPerfect (NodePtr t)     // Driver function.  Non-recursive.
{
    int heightOf_t; // We won't use the value that this picks up in the call
                    // below but we MUST pass the function below a 2nd parameter.

    return testPerfectAndHeight (t, heightOf_t);
} // isPerfect()


// The following function does most of the work.  It determines whether the tree
// pointed to by  tp  is perfect, and it also calculates the height of that tree
// (and puts it in the "export" variable "height") if the tree is perfect.

bool testPerfectAndHeight (NodePtr tp, int & height)
{
    if (!tp)          // The base case
    {
        height  = -1;  // The height of an empty tree is -1.
        return true;   // An empty tree is perfect.
    }

    else
    {
        int leftHeight, rightHeight;

        if ( ! testPerfectAndHeight (tp->left, leftHeight)   // if left subtree
            || ! testPerfectAndHeight (tp->right, rightHeight) )// or right is
                                                                // not perfect
            return false;  // no need to give "height" a value"

        else  // At this point we know the two subtrees are both perfect, and
              // their separate heights have been calculated by the two function.
              // calls Do the two subtrees have the same height?

            if (leftHeight != rightHeight)
                return false;               // no need to give "height" a value

            else
            {
                height = 1 + leftHeight;   // left and right heights are the same
                return true;
            }
    }
} // testPerfectAndHeight()
```

Written Exercises

7-1.



(a) Which node is the root of the binary tree above?

(b) What is the height of this tree?

(c) How many nodes are in the right subtree of 50 ? In the left subtree of 35?

(d) How many empty subtrees does this tree contain? [Hint: The node 21 has an empty left subtree. The node 35 has an empty right subtree. Both subtrees of node 94 are empty.]

(e) What are the descendants of node 77 ?

(f) Which nodes are ancestors of node 74 ?

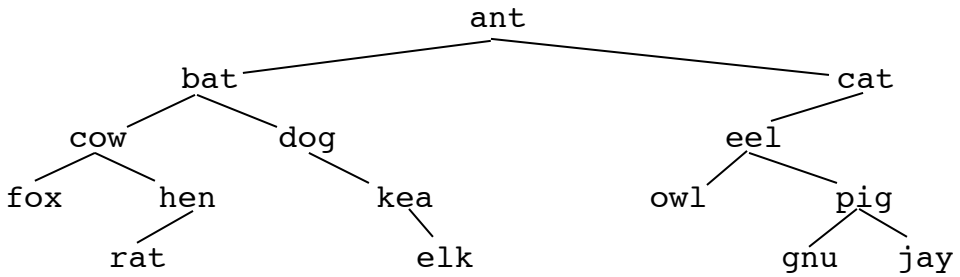
(g) Which nodes have siblings? (List all the sibling pairs.)

(h) What is the height of the left subtree of node 77 ? What is the height of its right subtree?

(i) What is the level number of node 80 ?

(j) List all the leaves of this tree.

7-2.



(a) What is the height of the binary tree above?

(b) How many nodes are in the left subtree of `eel`? In the right subtree of `bat` ?

(c) How many empty subtrees does this tree contain?

(d) What are the descendants of node `cow` ?

(e) Which nodes are ancestors of node `gnu` ?

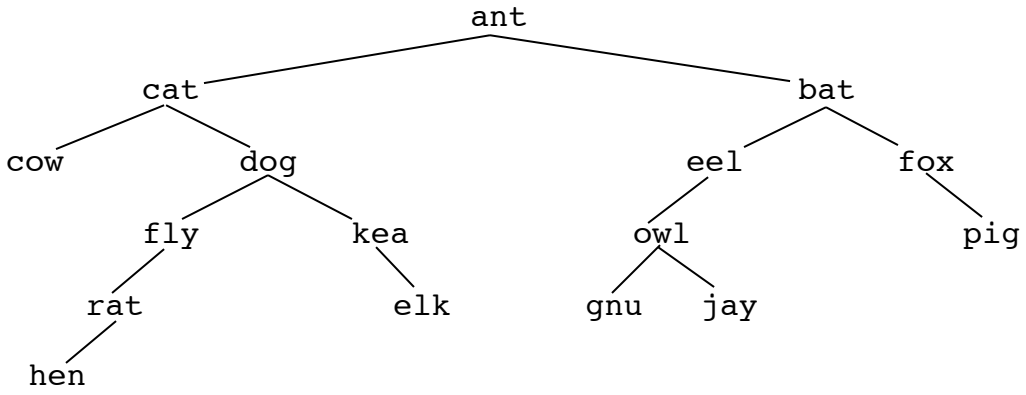
(f) Which nodes have siblings? (List all the sibling pairs.)

(g) What is the height of the left subtree of node `cat` ? What is the height of its right subtree?

(h) What is the level number of node `eel` ?

(i) List all the leaves of this tree.

7-4.



(a) What is the height of the binary tree above?

(b) How many nodes are in the left subtree of `dog`? In the right subtree of `eel` ?

(c) How many empty subtrees does this tree contain?

(d) Which nodes are descendants of node `bat` ?

(e) Which nodes are ancestors of node `gnu` ?

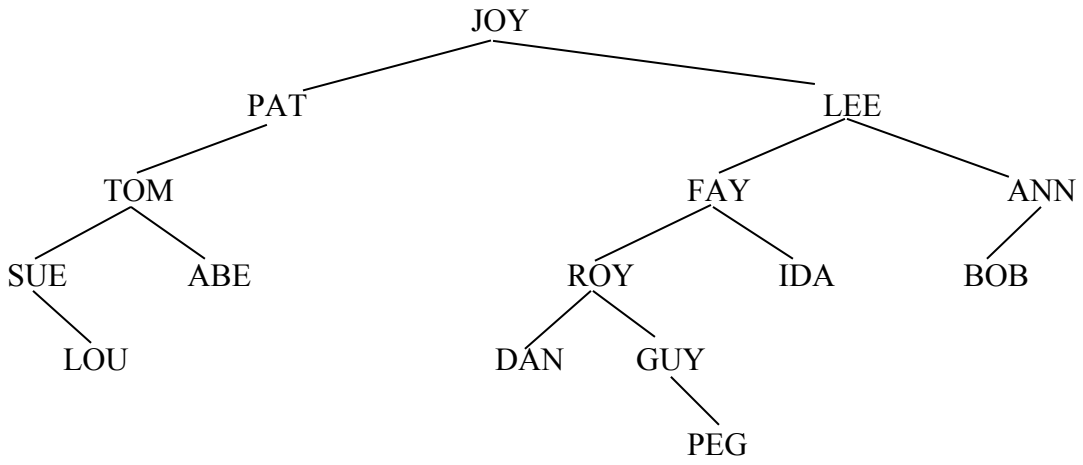
(f) Which nodes have siblings? (List all the sibling pairs.)

(g) What is the height of the left subtree of node `eel` ? What is the height of its right subtree?

(h) What is the level number of node `fly` ?

(i) List all the leaves of this tree.

7-5. (a) Suppose the nodes of the binary tree below are printed using any of the "`printLabels`" functions shown on page 7-7 in the class notes. What will the print-out look like?



(b) When the `printLabels` function prints the tree above, what is the *total* number of stack frames that will go onto the run time stack? (Each will be abandoned after it has been used.)

(c) While the `printLabels` function is printing the tree above, what is the *maximum* number of non-abandoned stack frames that will be on the run-time stack *at any one time*?

7-6. (a) Suppose the nodes of the binary tree in Exercise 7-2 are printed using any of the "`printLabels`" functions shown on page 7-6 in the class notes. What will the print-out look like?
(b) When the `printLabels` function prints the tree, what is the *total* number of stack frames that will go onto the run time stack? (Each will be abandoned after it has been used.)
(c) While the `printLabels` function is printing the tree, what is the *maximum* number of non-abandoned stack frames that will be on the run-time stack *at any one time*?

7-8. (a) Suppose the nodes of the binary tree in Exercise 7-4 are printed using the "`printLabels`" function shown on page 7-6 in the class notes. What will the print-out look like?
(b) When the `printLabels` function prints the tree, what is the *total* number of stack frames that will go onto the run time stack? (Each will be abandoned after it has been used.)
(c) While the `printLabels` function is printing the tree, what is the *maximum* number of non-abandoned stack frames that will be on the run-time stack *at any one time*?

7-9. What is the height of an empty binary tree? What is the height of a binary tree with one node? What is the height of a binary tree with two nodes? What are the possible heights of a binary tree with three nodes?

7-10. What are the possible heights of a binary tree with five nodes? Draw pictures to illustrate your answer.

7-13. Suppose the nodes of the binary tree in Exercise 7-5 are printed in pre-order. What will the print-out look like? Answer the same question for a post-order print-out.

7-14. Suppose the nodes of the binary tree in Exercise 7-2 are printed in pre-order. What will the print-out look like? Answer the same question for a post-order print-out.

7-16 Suppose the nodes of the binary tree in Exercise 7-4 are printed in pre-order. What will the print-out look like? Answer the same question for a post-order print-out.

7-17. Suppose that when the character nodes of a certain binary tree are printed out in *pre-order*, the list looks like this (it would be in a column instead of on a single line):

```
A  C  D  G  H  I  J  B  E  K  L  M  F
```

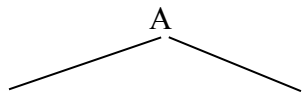
Suppose that when the nodes of the *same* tree are printed out in *symmetric* order, the list looks like this:

```
C  H  G  J  I  D  A  E  L  K  M  B  F
```

Use this information to reconstruct the tree in its entirety. Show at every stage of your solution how you are *systematically* working out the answer.

I'll start you off. First we note that the pre-order print-out tells us that A is the root of the tree. Then the symmetric order print-out tells us that the left subtree contains C H G J I D and the right subtree contains E L K M B F . Because of the recursive nature of the symmetric order print-out, these sequences are the *symmetric order print-outs* of the left and right subtrees.

Going back to the pre-order print-out and remembering that it *recursively* prints the left subtree before the right subtree, we find that the pre-order print-out of the left subtree is C D G H I J and the right is B E K L M F . To summarize, we know the tree looks like this at the top:



pre-order left subtree:	C D G H I J	B E K L M F	pre-order right subtree
symmetric left subtree:	C H G J I D	E L K M B F	symmetric right subtree

Continue in this manner on each of the subtrees.

7-18. Suppose that when the integer nodes of a certain binary tree are printed out in *symmetric order*, the list looks like this (except that it would be in a column instead of on a single line):

```
70  12  96  51  83  62  38  44  25
```

Suppose that when the nodes of the same tree are printed in *post-order*, the list looks like this:

```
70  96  83  51  38  62  44  25  12
```

Use this information to reconstruct the entire tree. Show at every stage of your solution how you are *systematically* working out the answer.

7-20. Suppose that when the integer nodes of a certain binary tree are printed out in *symmetric order*, the list looks like this (except that it would be in a column instead of on a single line):

```
70  12  96  51  83  62  38  44  25
```

Suppose that when the nodes of the same tree are printed in *post-order*, the list looks like this:

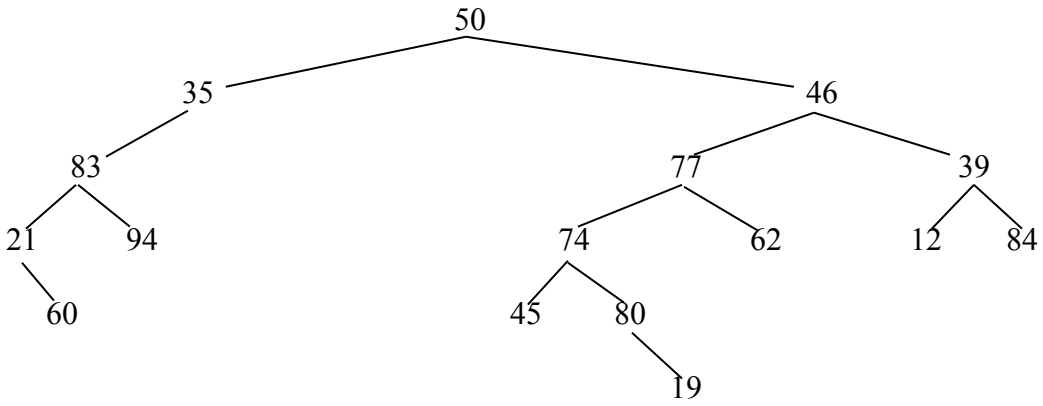
```
12  70  51  83  62  38  96  25  44
```

Use this information to reconstruct the entire tree. Show at every stage of your solution how you are *systematically* working out the answer.

7-22. Using the `struct node` definition near the top of page 7-11 in the class notes, write a function that prints the nodes of a binary tree in ***level order***, that is, by levels, starting with level 0 and going down in the tree. At each level, print all the nodes in that level from left to right. For example, if the binary tree shown below were printed in level order, the nodes would appear this way (possibly in a column):

50 35 46 83 77 39 21 94 74 62 12 84 60 45 80 19 .

HINT: declare and use a *queue* of pointers, but *no recursion*.



Solutions of Odd Numbered Exercises

7-1. (a) Which node is the root of this tree? **50**
(b) What is the height of this tree? **5**
(c) How many nodes are in the right subtree of 50 ? **10** In the left subtree of 35 ? **4**
(d) How many empty subtrees does this tree contain? **17**
(e) What are the descendants of node 77 ? **74, 45, 80, 19, 62**
(f) Which nodes are ancestors of node 74 ? **77, 46, 50**
(g) List all the sibling pairs. **35 & 46, 21 & 94, 77 & 39, 12 & 84, 74 & 62, 45 & 80**
(h) What's the height of the left subtree of 77 ? **2** What's the height of its right subtree? **0**
(i) What is the level number of node 80 ? **4**
(j) List all the leaves of this tree. **60, 94, 45, 19, 62, 12, 84**

7-5. (a) I'll show the labels on one line, although they would be printed on separate lines.

```
SUE LOU TOM ABE PAT JOY DAN ROY GUY PEG FAY IDA LEE BOB ANN
```

(b) There will be one stack frame for every pointer in or to the tree, because the function is called on every pointer in the tree and also on the pointer to the root. There are 15 nodes in the tree, so there are 30 pointers in the nodes. The pointer to the root makes 31 . Thus the answer is that there will be 31 stack frames that will go onto the run-time stack during execution.
(c) The greatest "stack frame depth" occurs when the recursion reaches either of the empty subtrees of the node PEG . At that point there is a stack frame with a pointer to JOY, another with a pointer to LEE, and similarly FAY, ROY, GUY, and PEG. Finally there is the stack frame with NULL, indicating the left, then later the right, subtrees of PEG. Counting up we see that the maximum number of stack frames is 7 .

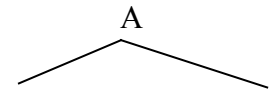
7-9. An empty tree has height -1 .
A tree with exactly one node has height 0 .
A tree with exactly two nodes has height 1 .
A tree with exactly three nodes can have height 1 or 2 .

7-13. I'll show the labels on one line, although they would be printed on separate lines.
The pre-order print-out is
JOY PAT TOM SUE LOU ABE LEE FAY ROY DAN GUY PEG IDA ANN BOB
and the post-order print-out is
LOU SUE ABE TOM PAT DAN PEG GUY ROY IDA FAY BOB ANN LEE JOY

7-17. First we note that the pre-order print-out tells us that A is the root of the tree.

Then the symmetric order print-out tells us that the left subtree contains C H G J I D and the right subtree contains E L K M B F . Because of the recursive nature of the symmetric order print-out, these sequences are the *symmetric order print-outs* of the left and right subtrees.

Going back to the pre-order print-out and remembering that it *recursively* prints the left subtree before the right subtree, we find that the pre-order print-out of the left subtree is C D G H I J and the right is B E K L M F . To summarize



pre-order left subtree:	C D G H I J	B E K L M F	pre-order right subtree
symmetric left subtree:	C H G J I D	E L K M B F	symmetric right subtree

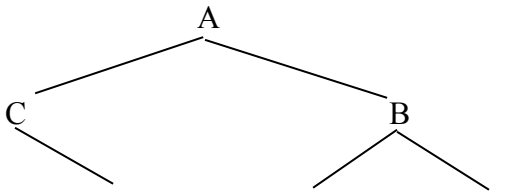
The root of the left subtree is C .
Its left and right subtrees are given by

pre:	empty	D G H I J
sym:	empty	H G J I D

The root of the right subtree is B .
Its left and right subtrees are given by

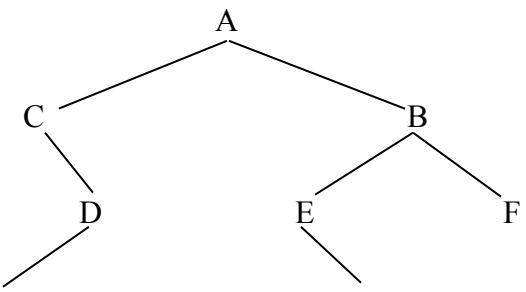
pre:	E K L M	F
sym:	E L K M	F

So far, then we have



pre:	D G H I J	E K L M	F
sym:	H G J I D	E L K M	F

The root of the right subtree of C is D , which is then easily seen to have an empty right subtree.
The root of the left subtree of G is E , which is then easily seen to have an empty left subtree.
Thus we are now at this stage:

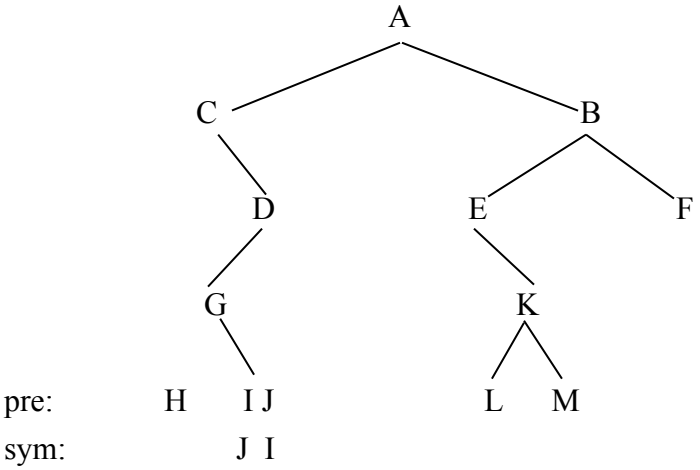


pre:	G H I J	K L M
sym:	H G J I	L K M

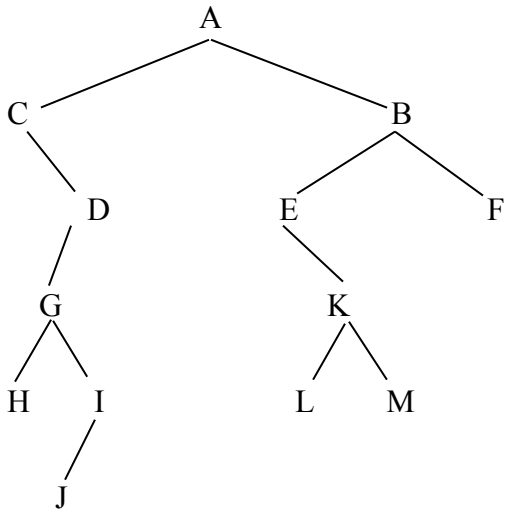
The root of the left subtree of D is G .

The root of the right subtree of E is K , which is then seen to have subtrees of size 1.

Thus we have now reached this stage:



Now it is easy to draw the final tree:



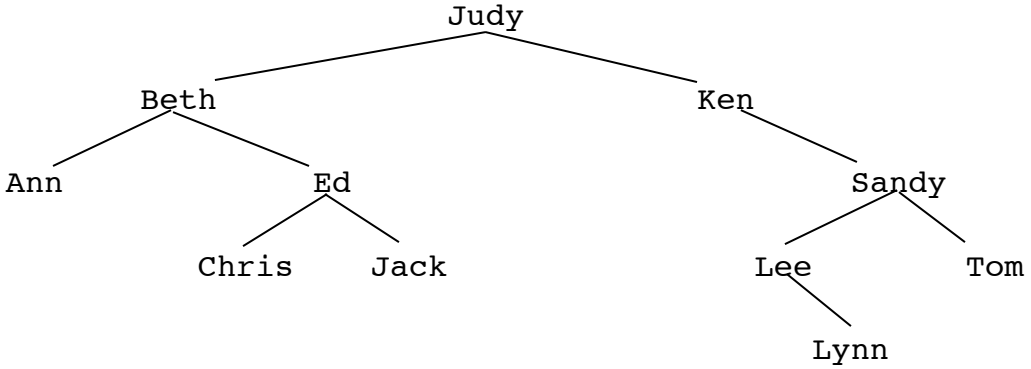
8. Binary Search Trees

Definition: A **binary search tree** is a binary tree with the following properties:

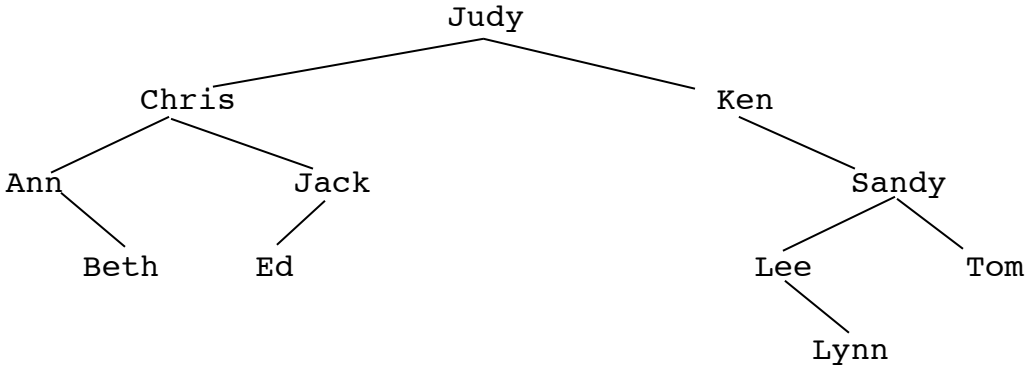
(a) Each node in the tree carries one object of some type containing a "key" value that distinguishes it from all other objects stored in the tree.

(b) For each (i.e., for every) node x in the tree, all the keys in the left subtree of x are smaller than the key in x , and all the keys in the right subtree of x are larger than the key in x .

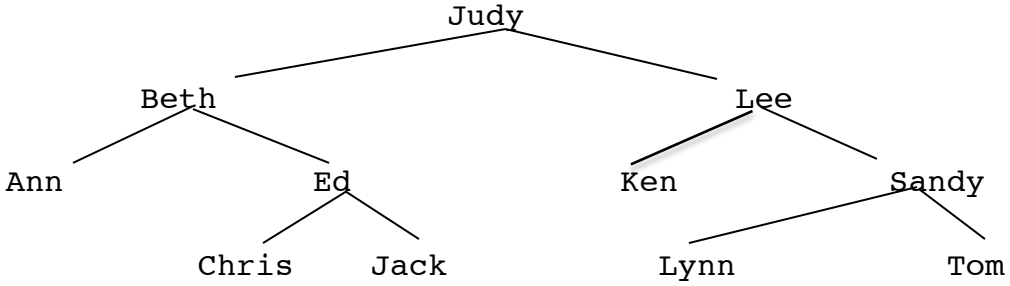
Is the tree shown here a binary *search* tree?



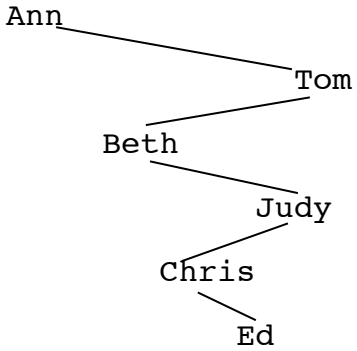
How about the following tree?



And the following?



And this one?



In a binary search tree,

- where is the node with smallest key value located?
- where is the node with largest key value located?
- given a particular node with a key value K , where is the node containing the "next larger key" (if any) in the tree? We call this node, or its key value, the **in-order successor** of the node with key K .
- given a particular node with a key value K , where is the node containing the "next smaller key" (if any) in the tree? We call this node, or its key value, the **in-order predecessor** of the node with key K .

Suppose we have a program that constructs and manipulates binary search trees. For concreteness, suppose the nodes that make up the trees have the same structure that was used for the examples starting on page 08-3. The data object stored in each node is just a character string named "`label`". In this case, the "key" of the data object will just be the data object itself. In what follows, when we are writing code we will use the identifier "`label`" for the key, but in the discussion of the code we will refer to the "key" in the node.

Let's write a search function that takes as its arguments a pointer to the root of a binary search tree and a key value and searches the tree for a node with that key. Have the function return a pointer to the node containing the key if such a node exists. Otherwise have the function return NULL.

```
NodePtr location (NodePtr rootp, char * target) //iterative version
{
    while (rootp)
    {
        int comparison = strcmp(rootp->label, target);

        if ( comparison < 0 )     // rootp->label < target
            rootp = rootp->right;

        else if (comparison > 0 ) // rootp->label > target
            rootp = rootp->left;

        else                      // rootp->label matches target
            return rootp;         // RETURN from inside the loop
    }

    return NULL; // The search was unsuccessful
} // location()
```

It is also possible to write this function recursively.

```
NodePtr location (NodePtr rootp, char * target) //recursive version
{
    if (!rootp)     // if rootp is NULL
        return NULL;

    else
    {
        int comparison = strcmp(rootp->label, target);

        if ( comparison < 0 )  // rootp->label < target
            return location (rootp->right, target);

        else if (comparison > 0 ) // rootp->label > target
            return location (rootp->left, target);

        else
            return rootp;  // rootp->label matches target
    }
} // location()
```

Note that all the recursive calls in the function above are instances of "tail recursion", which is to say that each recursive call is the last statement to be executed before the return from the function. It is usually the case that a recursive function whose recursive calls are all tail recursions can be written about as simply in "iterative" form, i.e., in a form that uses a simple loop in place of the recursive calls. The interative version will then be more time and space efficient than the recursive version because there will be no stack frame "overhead" of the kind that occurs during the execution of recursive functions.

Now let's write a function that inserts a string into a binary search tree. More precisely, given a character string S , the function will determine whether S is already in the tree, and if it is not, the function will create a new node, put a copy of S into the node, and attach the node as a leaf at the correct position in the search tree. We'll make the function return `true` if it succeeds in inserting S ; if it fails (because a copy of S is already in the tree), the function will return `false` . If the function fails because dynamic allocation fails, we'll terminate the entire program (in this version; we don't always do this).

```
bool insert (NodePtr & rootp, char * newLabel)  //iterative code
{
    if (!rootp)  // if rootp is NULL
    {
        rootp = new node;
        testAllocation (rootp);
        strcpy(rootp->label, newLabel);
        rootp->left = rootp->right = NULL;
        return true;
    }
    else
    {
        NodePtr  p        = rootp;
        bool     finished = false;
        int      comparison;

        while (! finished)
        {
            comparison = strcmp(p->label, newLabel);
            if ( comparison < 0 )  // p->label < newLabel; go right
                if (p->right)
                    p = p->right;
                else // p->right is NULL; want to insert newLabel there
                {
                    p->right = new node;
                    testAllocation (p->right);
                    p = p->right;
                    strcpy(p->label, newLabel);
                    p->left = p->right = NULL;
                    finished = true;               // force loop to stop
                }

            else if ( comparison > 0 ) // p->label > newLabel; go left
                if (p->left)
                   p = p->left;
                else // p->left is NULL; want to insert newLabel there
                {
                    p->left = new node;
                    testAllocation (p->left);
                    p = p->left;
                    strcpy(p->label, newLabel);
                    p->left = p->right = NULL;
                    finished = true;              // force loop to stop
                }

            else // comparison == 0  is the only remaining possibility
                finished = true;
        }

        return (comparison != 0);
    }
} // insert()
```

Note that there is considerable duplication of code in the function shown above. This can be avoided in two ways. One way is to write a helper function that can be called to construct and attach a new node to the tree. The second way is to use recursion.

```
bool insert (NodePtr & rootp, char * newLabel) //recursive code
{
    if (!rootp)  // base case
    {
        rootp = new node;
        testAllocation (rootp);
        strcpy(rootp->label, newLabel);
        rootp->left = rootp->right = NULL;
        return true;
    }

    else
    {
        int comparison = strcmp(rootp->label, newLabel);

        if ( comparison < 0 )  // rootp->label < newLabel
            return insert (rootp->right, newLabel);

        else if ( comparison > 0 ) // p->label > newLabel
            return insert (rootp->left, newLabel);

        else
            return false; //rootp->label matches newLabel
    }
} // insert()
```

All the recursive calls in this version of the function are instances of tail recursion, so efficiency suggests that we use the iterative (i.e., the loop-controlled, non-recursive) version of the function, perhaps rewritten with a helper function. The recursive version is so much more elegant, however, that unless efficiency is of utmost importance, any person with taste will likely choose the recursive version in this situation.

Let's write a function that removes a string from a binary search tree. More precisely, given a character string S , the function will determine whether a copy of S is in the tree, and if it is, the function will remove the node containing it from the tree, taking care to keep the tree a binary *search* tree. We'll make the function return `true` if it succeeds in removing S ; if it fails (because a copy of S is not in the tree), the function will return `false` .

Removing a node N from a binary search tree requires some care. If the node N is a leaf, then the operation is very simple: call the C++ "`delete`" function on the pointer that's pointing to N and then set that pointer to NULL. If N has only one child, the process is also fairly easy: point an auxiliary pointer P to N , switch the tree's pointer from N to the child node of N , and then call "`delete`" on P . The hard case occurs when N has two children. In this case, we can locate the in-order predecessor of N in the left subtree of N or else the in-order successor of N in the right subtree of N , and then move one of those nodes into the position occupied by N . Alternatively, we can copy the information carried by the in-order predecessor (or successor) node into N and then delete the in-order predecessor (or successor) node.

```
bool remove (NodePtr & rootp, char * target) //recursive version
{
    if (!rootp)   // if rootp is NULL
        return false;

    else
    {
        int comparison = strcmp (rootp->label, target);

        if (comparison < 0)
            return remove (rootp->right, target);

        else if (comparison > 0)
            return remove (rootp->left, target);

        //If execution reaches this point, we know rootp->label matches
        //the specified target. Now we determine whether the node
        //*rootp has no children, one child, or two children.

        else if (!rootp->left && !rootp->right) // no children
        {
            delete rootp;   // deallocate the node pointed to by rootp
            rootp = NULL;
            return true;
        }

        else if (!rootp->left) // no left, but right child is present
        {
            NodePtr temp = rootp;
            rootp = rootp->right; // make parent point to grandchild
            delete temp;          // deallocate the node being removed
            return true;
        }

        else if (!rootp->right) // no right, but left child is present
        {
            NodePtr temp = rootp;
            rootp = rootp->left;  // make parent point to grandchild
            delete temp;          // deallocate the node being removed
            return true;
        }

        else    // The only remaining case: the label to delete has two
                // children.  We'll copy the label of the in-order
                // predecessor into the current node, and remove the
                // in-order predecessor node.
        {
            temp = rootp->left;  // find predecessor of target label
            while (temp->right)
                temp = temp->right;

            strcpy (rootp->label, temp->label);

            return remove (rootp->left, temp->label); //kill predecessr
        }
    }
} // remove()
```

You may wonder why we make a recursive call in the last line of the code above in order to "kill the in-order predecessor". Since `temp` already points to the in-order predecessor, why not just write

```
delete temp;
temp = NULL.
```

Here is the answer: `temp` is pointing to the node to be deleted, but that same node is also pointed to by a pointer in the parent node, and it's the pointer in the parent node that needs to be set to NULL. The only way to get to that parent node is through a recursion that eventually calls `remove` and passes it the pointer from the parent to the node to be deleted.

Why are we interested in binary search trees? The answer is that we hope that the binary search tree data structure will turn out to be a good storage structure for implementing a "table of keyed objects" in the case where the table is expected to have all of the following properties:
--> the table will be highly dynamic (there will be lots of insertions and removals);
--> many accesses will occur;
--> at least one of the operations FindMin, FindMax, TraverseInOrder will occur frequently.

Note that when we perform an access on a binary *search* tree T, the maximum number of nodes we will have to examine will be _____.
The same is true of insertions and removals. Thus we will want to keep our binary search trees as short as possible while performing insertions and removals.

Suppose we have N nodes to "play with". What is the height of the shortest binary tree we can build with those N nodes? To put the same question in different words, for a given non-negative integer N, what is the minimum possible height of a binary tree with that number of nodes?

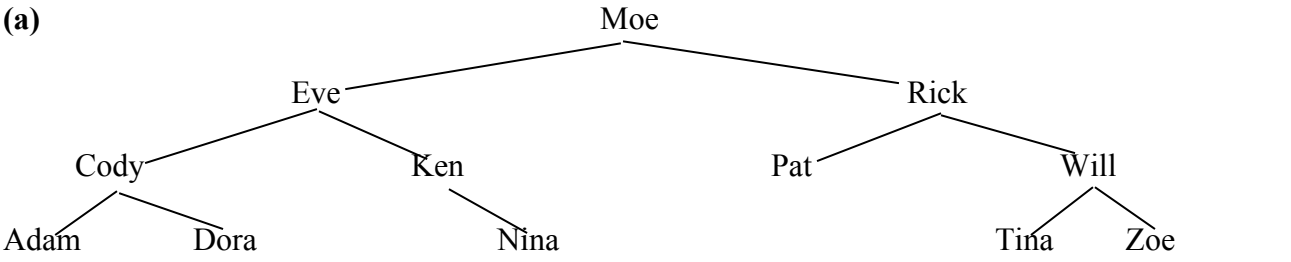
N	Minimum height of bin. tree with N nodes

If we implement a table as a binary search tree, and if we take no precautions about keeping the tree short, what kind of behavior can we expect? In C455, we will prove that when N nodes are inserted into a binary tree *in random order*, the "expected" (average) number of object comparisons that must be made when accessing a key is approximately $1.3 \log_2(N)$. Thus it might appear that no precautions are necessary when performing insertions and removals. Be warned, however, that when batches of data arrive to be placed in a table, the data objects are often already sorted, or at least partially sorted, by key. That is, the data does not arrive *in random order*. When a binary search tree is being used for table implementation, this can result in "tall, sparse, stringy" trees that give poor performance for the usual operations. In fact, in the worst case, the tree might degenerate into a linked list of N nodes, in which case performance will be inferior to what we have with an ordered array implementation.

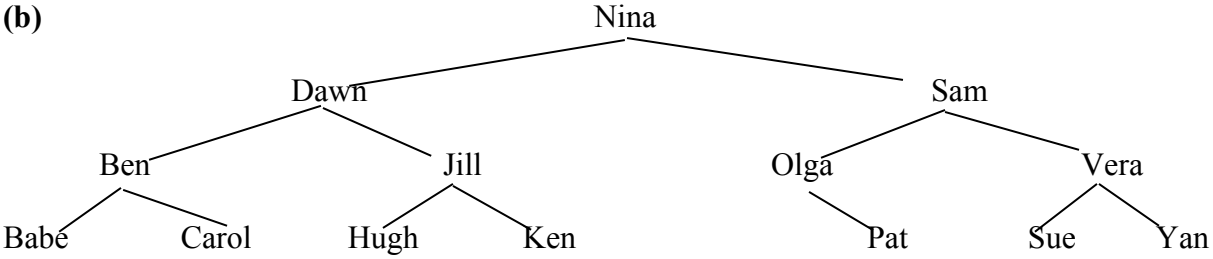
Written Exercises

8-1. Which of the following two binary trees is a binary *search* tree? For the one that is NOT a search tree, explain the way(s) in which it fails to be a search tree.

(a)

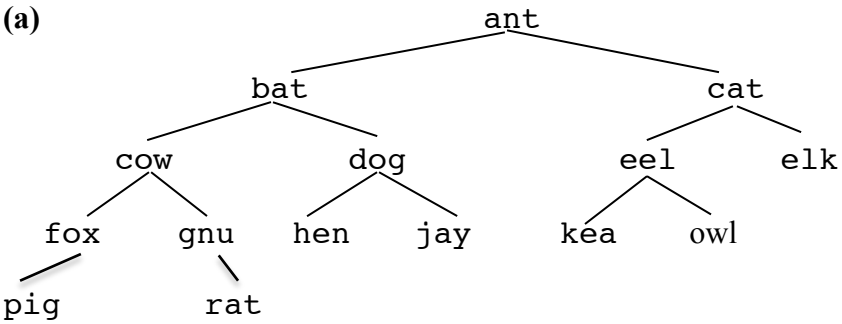


(b)

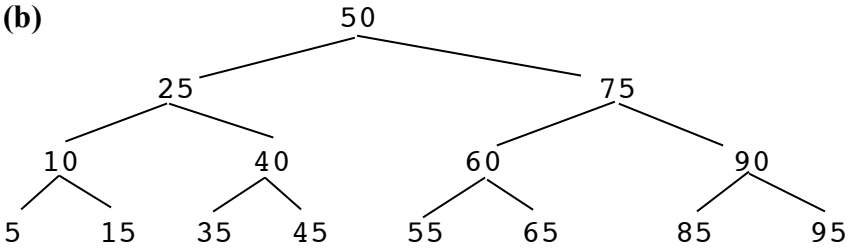


8-2. Which -- if any -- of the following binary trees are binary *search* trees?

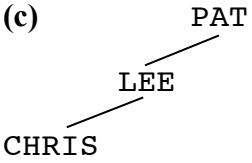
(a)



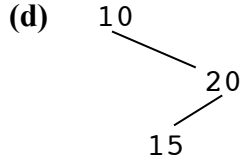
(b)



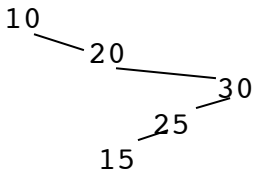
(c)



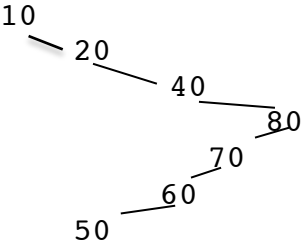
(d)



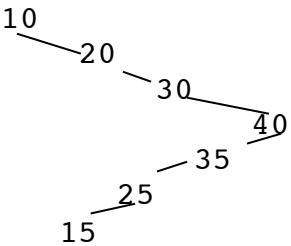
(e)



(f)

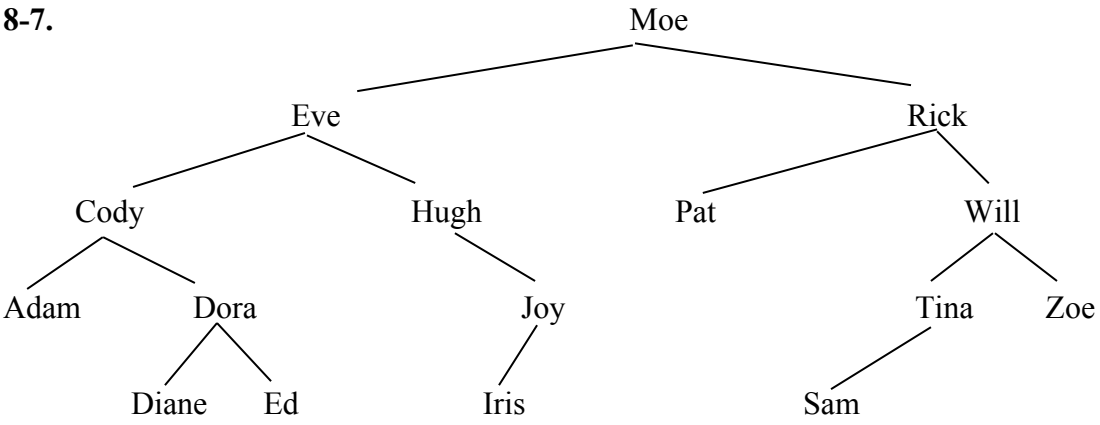


(g)



8-4. Suppose the keys in a binary *search* tree are character strings, and suppose we perform a symmetric order print-out of those keys. What special property will the resulting list of keys have?

8-7.



In the binary search tree shown above, identify the following:

(a) the in-order predecessor of Eve;

(b) the in-order successor of Moe;

(c) the in-order successor of Joy.

8-8. In the binary search tree shown in Exercise 8-1(b) (preceding page), identify the following:

(a) the in-order predecessor of Sam;

(b) the in-order successor of Dawn;

(c) the in-order predecessor of Olga.

(d) the in-order successor of Ben.

8-10. In the binary search tree shown in Exercise 8-1(b) (preceding page), identify the following:

(a) the in-order predecessor of Dawn;

(b) the in-order successor of Nina;

(c) the in-order predecessor of Hugh.

(d) the in-order successor of Olga.

8-11. Suppose the following sequence of operations is performed *without rebalancing* on the binary search tree shown in Exercise 8-7. Show what the binary search tree will look like after all these operations have been performed. In each case where a removal is performed, use the ***in-order predecessor*** wherever that is appropriate.

Insert Irene (don't re-draw the tree; just add the node Irene in the correct spot in the tree).
Insert Sunny (ditto).
Insert Hal.
Insert Olga.
Remove Zoe.
Remove Iris.
Remove Eve and **re-draw the entire tree** so that I can see the final result of these 7 operations.

8-12. Suppose the following sequence of operations is performed *without rebalancing* on the binary search tree shown in Exercise 8-1(b), page 8-9. Show what the binary search tree will look like after all these operations have been performed. In each case where a removal is performed, use the ***in-order successor*** wherever that is appropriate.

Insert Sarah (don't re-draw the tree; just add the node Sarah in the correct spot in the tree).
Insert Nola (ditto).
Insert Iris.
Remove Yan.
Remove Hugh.
Remove Sam and **re-draw the entire tree** so that I can see the final result of these 6 operations.

8-14. Suppose the following sequence of operations is performed *without rebalancing* on the binary search tree shown in Exercise 8-1(b), page 8-9. Show what the binary search tree will look like after all these operations have been performed. In each case where a removal is performed, use the ***in-order predecessor*** wherever that is appropriate.

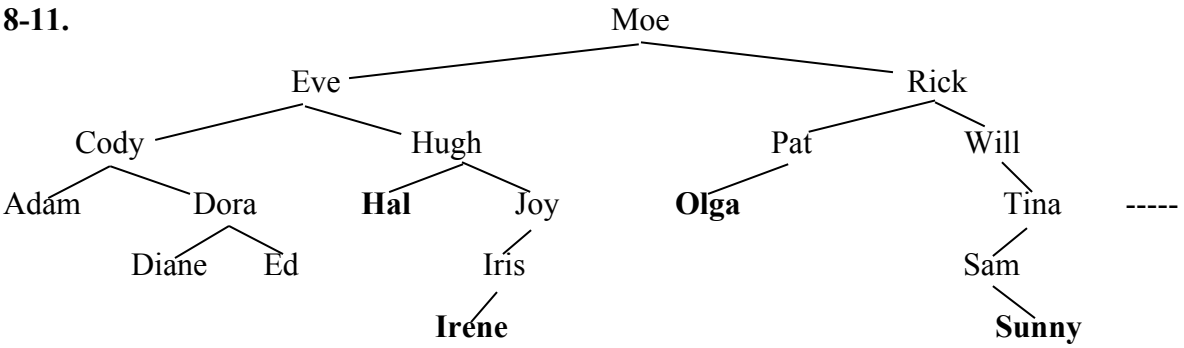
Insert Dave (don't re-draw the tree; just add the node Dave in the correct spot in the tree).
Insert Nora (ditto).
Insert Omar.
Remove Babe.
Remove Pat.
Remove Dawn and **re-draw the entire tree** so that I can see the final result of these 6 operations.

Solutions for Odd-Numbered Exercises

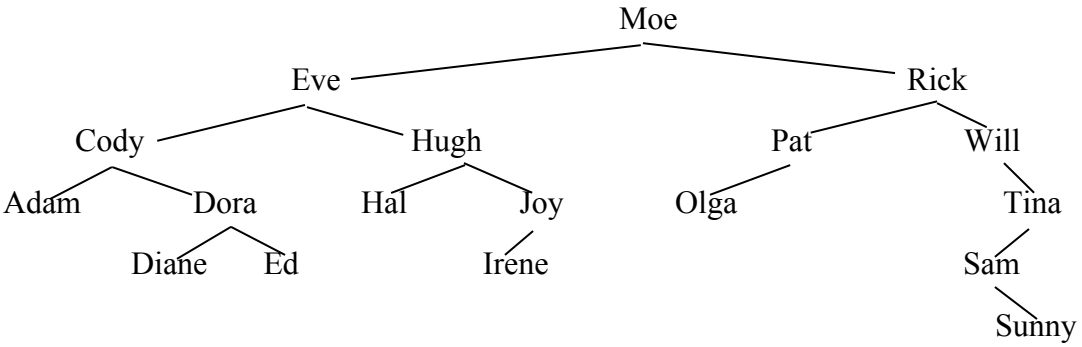
8-1. Binary tree (a) is not a search tree because one of the nodes (Nina) in the left subtree of Moe is not smaller than Moe. Binary tree (b) is a search tree.

8-7. (a) The in-order predecessor of Eve is Ed.
(b) The in-order successor of Moe is Pat. **(c)** The in-order successor of Joy is Moe.

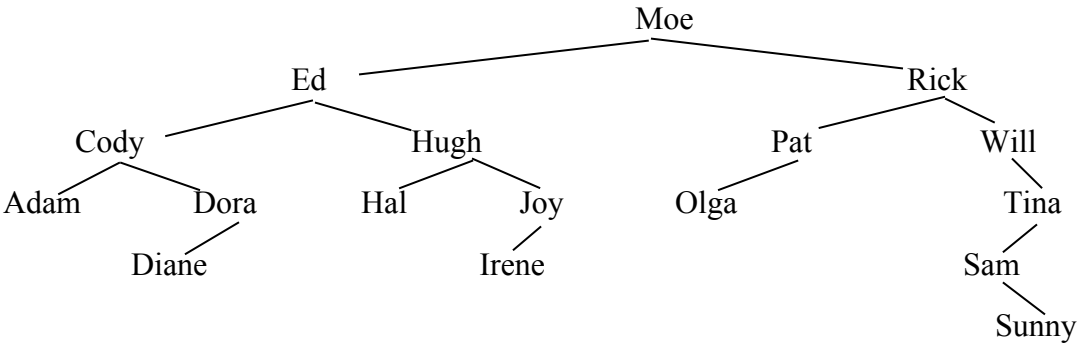
8-11.



The tree above is what we get after inserting Irene, Sunny, Hal, and Olga, and then removing Zoe. When we remove Iris, we can simply make the left pointer in the node Joy point to Iris's child Irene. This produces the tree shown below.



If we want to remove Eve, then since she has two children the removal operation is more difficult. We locate either the in-order predecessor of Eve (namely Ed) or the in-order successor of Eve (namely Hal), replace Eve by one of those, and then remove the predecessor or successor. If we use the in-order predecessor, then the resulting tree will be



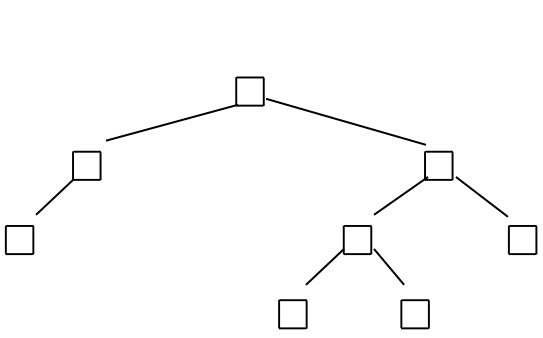
9. AVL Trees

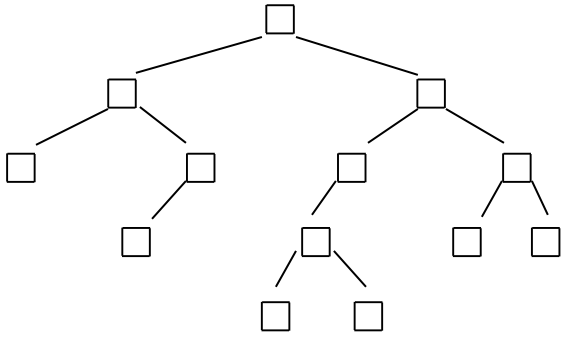
Recall that the height of a binary tree containing n nodes can be as small as $\lfloor \log_2(n) \rfloor$. Since an insertion, an access, or a removal of an object in a binary *search* tree involves going down just one path from the root in the tree, the number of instructions required to perform one of these operations should be roughly proportional to log(n). Thus it is reasonable to expect that a binary search tree can be an excellent data structure for efficient implementation of a *table* containing a lot of data (i.e., many keyed objects). Highly dynamic tables -- those in which many insertions and removals can be expected -- can be implemented efficiently using binary search trees.

We know that the *minimum possible* height of a binary tree with n nodes is $\lfloor \log_2(n) \rfloor$. Is it possible to make sure that every time an object is inserted into or removed from a binary search tree, the height will remain as small as possible? The answer is yes, but unfortunately the amount of time required to restore *minimum possible* height to a tree on which a modifying operation has just been performed is large; in fact, no one has ever found a way to do it in less than $O(n)$ time. If, however, we are willing to be satisfied with less than "perfection", then there are several methods for keeping the height of a binary search tree down to a constant multiple of the minimum possible height in an amount of time that's $O(\log(n))$. The first such method was discovered in the early 1960s by two Russians, Adel'son-Velskii and Landis, and we use the name "AVL tree" for a binary search tree that satisfies a "height-balance" restriction that they invented.

Definition. A binary tree (of any kind) is said to be ***height balanced*** iff it has the following property: for each node x in the tree, the height of the left subtree of x differs from the height of the right subtree of x by at most 1. [Here is an alternative definition that's recursive: a binary tree is said to be height balanced iff it is empty or satisfies both of the following conditions:
(a) its left and right subtrees differ in height by at most 1 ; and
(b) its left and right subtrees are height balanced.]

Examples:
(1) Is a binary tree with one node height balanced?
(2) Is the empty binary tree height balanced?
(3) Is a binary tree with two nodes necessarily height balanced?
(4) Is a binary tree with three nodes necessarily height balanced?
(5) How about the two binary trees shown below?





Question: What is the *maximum* possible height for a binary tree with N nodes?

Question: What is the maximum possible height of a *height-balanced* binary tree with N nodes?
Solution. For $N = 0$, the answer is -1. For $N = 1$, the answer is 0. For $N = 2$, the answer is 1. For $N = 3$, the answer is 1 because any binary tree with 3 nodes and height 2 is not height balanced. For $N = 4$, the answer is 2. Trial and error shows that for $N = 5$ and $N = 6$, the answer is still 2. For $N = 7$, however, we are able to build a height-balanced binary tree of height 3. What is the smallest value of N for which the answer will be 4? The answer can be obtained by noting that in a height balanced tree of height 4, one of the two subtrees must be of height 3, and thus have at least 7 nodes, while the other subtree can be of height 2, which requires only 4 nodes. Counting the one node for the root, we see that the smallest number of nodes in a height balanced tree of height 4 is $7 + 4 + 1 = 12$. A similar argument tells us the smallest number of nodes in a height balanced tree of height 5: one subtree must have height 4, so it must have at least 12 nodes; the other must have height at least 3, so it has at least 7 nodes; counting the root we have 20 nodes.

By generalizing the argument given in the preceding paragraph, it can be shown that the maximum height of a height balanced tree with N nodes is less than

$$\frac{1}{\log_2((1 + \sqrt{5})/2)} \log_2(N) \approx 1.44 \log_2(N) .$$

In C455 Analysis of Algorithms you will see how this formula is derived.

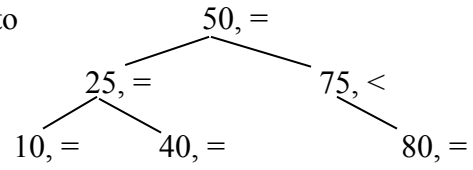
Definition. A binary tree is called an **AVL tree** if and only if it has the following two properties:
(1) it is a search tree, and
(2) it is height balanced.

Suppose we want to implement an AVL tree in C++ (say) in such a way that we can perform insertions, removals, and retrievals of nodes efficiently. How can we do this? The best method known is to use struct objects (called nodes") having the usual three members plus an additional member in which we store a "balance condition" at that node at all times. The balance condition member in a node x needs to be able to represent the following three conditions:
(a) the left and right subtrees of x are of equal height;
(b) the left subtree of x has height that's greater by 1 than the height of the right subtree of x
(c) the right subtree of x has height that's greater by 1 than the height of the left subtree of x
We could call our extra field in each node the "`balance`" member of the node, and it could be of `char` type with possible values `'='`, `'>'`, and `'<'` representing the three conditions listed above. The balance member of a node is sometimes called the "balance code" for that node.

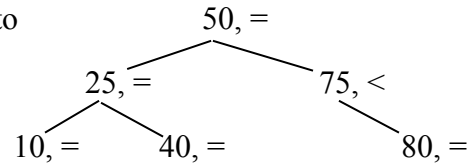
When an insertion or removal of an object is performed on a tree, this alters the structure of the tree and affects the balance conditions above the node where the structural alteration occurred. Thus when such an alteration is made at a node N, it is necessary to have N "report to its parent" a signal to indicate whether the subtree rooted at N grew. This allows the parent to make a change, if necessary, in its own balance code. Then the parent will report to its parent, and so on up the tree.

Let's look at some examples to see what happens when we insert a new object into an AVL tree implemented as described above.

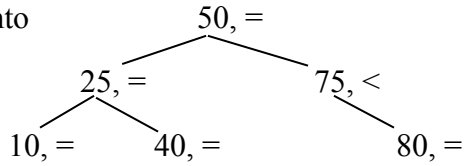
Example 1: Insert 60 into



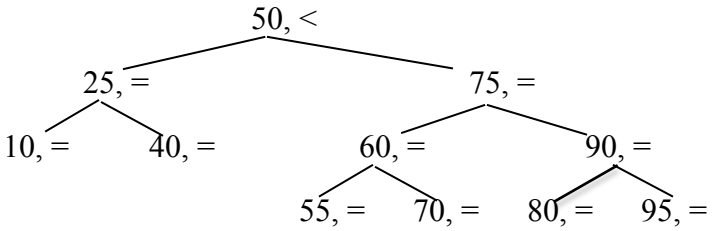
Example 2: Insert 30 into



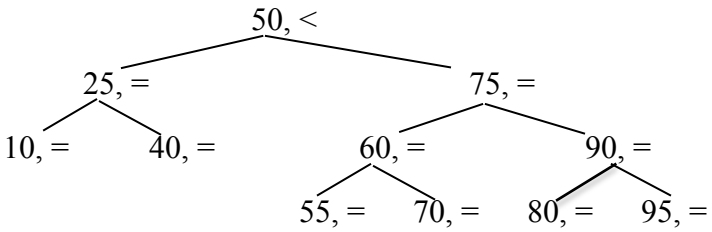
Example 3: Insert 90 into



Example 4: Insert 85 into

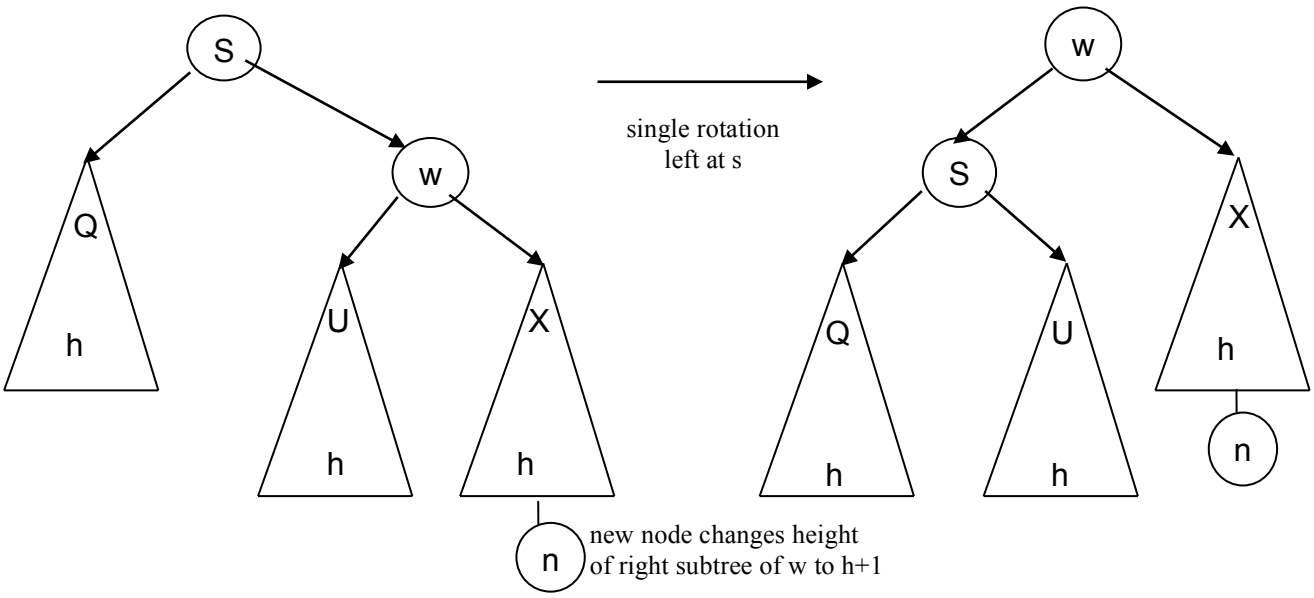


Example 5: Insert 72 into



Procedure: When a new node is inserted into an AVL tree, we set its condition member to '=' and then move up the tree, making adjustments in the condition members. If we arrive at a node s for which the insertion into one of its subtrees has produced a violation of the height-balance requirement of the subtree at s , then we re-balance the tree at that node by performing a "tree-rotation". It turns out that there are two cases to consider for insertion into a right subtree, and two "mirror image" cases for insertion into a left subtree. On the next two pages we'll look at the two possible cases for insertion into a right subtree.

Case 1: Suppose a new node has been inserted into the AVL tree shown below. Assume that the insertion took place in the right subtree of the right child of node s. Assume also that after the insertion, the balance condition at each level going up the tree has been appropriately modified, and the process has arrived back at node s with the "news" that its right subtree has grown in height. Suppose at the instant when this news arrives, the tree contains the balance conditions shown (a question mark indicates that the balance condition at that node is not of interest and can be any of the three possible codes).



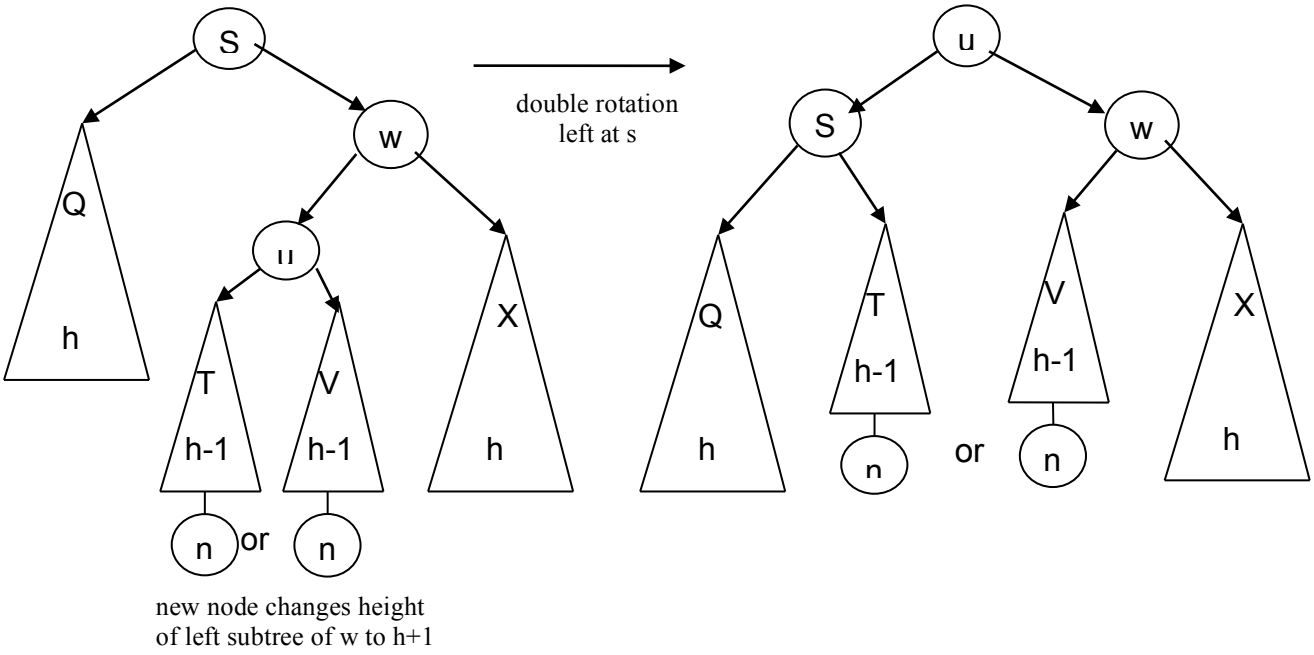
h = original height of the subtrees rooted at q, u, and x. (These subtrees *include* q, u, and x, respectively.)

When the height violation is discovered, and when the right child of s has balance condition < indicating that the subtree rooted at w grew in height due to insertion into its right subtree, then we perform a single rotation to the left at s. This operation

- brings its right child w of s up to the position occupied by s;
- makes s the left child of w; and
- re-attaches the left subtree of w as the right subtree of s.

Note that the result of this single rotation is to restore the height of this entire subtree to h+2, which was the height before the insertion took place. This means that at the next level (if any) up the tree, no adjustment will be necessary in its balance condition because its subtree will not have grown in height; in fact, no further adjustments in the balance conditions will be necessary at any level as we ascend the tree after the insertion.

Case 2: Suppose a new node has been inserted into the AVL tree shown below. Assume that the insertion took place in the left subtree of the right child of node s . Assume also that after the insertion, the height balance condition at each level going up the tree has been appropriately modified, and the process has arrived back at node s with the "news" that its right subtree has grown in height. Suppose at the instant when this news arrives, the tree contains the balance conditions shown (a question mark indicates that the balance condition at that node is not of interest and can be any of the three possible codes).



h = original height of the subtrees rooted at q , u , and x .

When the height violation is discovered, and when the right child of s has balance condition > indicating that the subtree rooted at w grew in height due to insertion into its left subtree, then we perform a double rotation to the left at s . This operation

- brings the grandchild u of s up to the position occupied by s ;
- makes s the left child of u and w the right child of u (so w stays "in place" in the tree);
- re-attaches the left subtree of u as the right subtree of s ; and re-attaches the right subtree of u as the left subtree of w .

Note that the result of this double rotation, like the result of a single rotation, is to restore the height of this entire subtree to h+2 , which was the height before the insertion took place. This means that at all levels (if any) farther up the tree, no adjustments in the balance conditions will be necessary.

Here is a summary of what we have learned by looking at examples and then the two possible cases involving tree rotation.

Suppose an insert has occurred in the *right* subtree of a node s. Suppose moreover that it has been reported that this insertion caused that right subtree rooted at w , say, to increase in height.

- If the balance condition at s is '=', change it to '<', and report an increase in height to the next level up the tree;.
- else if the balance condition at s is '>', change it to '=', and report to the next level up the tree that there was no increase in height in this tree.
- else (the balance condition at s is '<', the tree has now become unbalanced)
  - if the balance condition at w is '<',
    - perform a "single left rotation at s" (move s down to the left, and bring w up to the root position; the left subtree of w is re-attached as the right subtree of s);
    - change the balance condition at w to '=';
    - change the balance condition at s to '=';
    - report to the next level up the tree that there was no increase in height in this tree;
  - else if the balance condition at w is '>',
    - perform a "double left rotation at s " (move s down to the left, and bring the left child of w , call it u , up to the position that was occupied by s ; re-attach the left subtree of u as the right child of s ; re-attach the right subtree of u as the left child of w).
    - if the balance condition at u (the new root) is '<',
      - change the balance condition at s to '>';
      - change the balance condition at w to '=';
      - change the balance condition at u to '=';
    - else if the balance condition at u (the new root) is '>',
      - change the balance condition at s to '=';
      - change the balance condition at w to '<';
      - change the balance condition at u to '=';
    - else (the balance condition at u (the new root) is '='; means u was inserted)
      - change the balance condition at s to '=';
      - change the balance condition at w to '=';
      - change the balance condition at u to '=';
    - report to the next level up the tree that there was no increase in height in this tree;

    [There is no "else" here for a balance condition of '=' at w because it was the growth of the non-empty subtree rooted at w that caused the re-balancing at s to occur, so the balance condition at w cannot be '=' .]

Below is some code that shows how insertion takes place in a binary search tree without rebalancing. (We wrote this code earlier; see page 7-5.) What changes would we make in the code in order to be able to use it for insertion into an AVL tree?

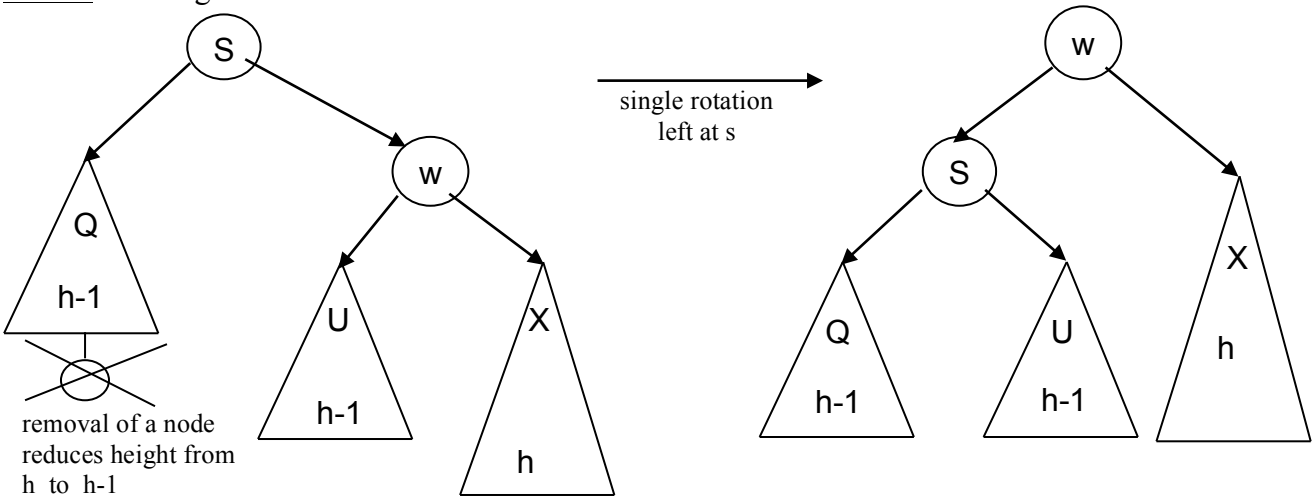
```
//recursive code
bool insert (NodePtr & rootp, char * newLabel, bool &grew)
{
    if (!rootp)  // base case
    {
        rootp = new Node;
        testAllocation (rootp);
        strcpy(rootp->label, newLabel);
        rootp->left = rootp->right = NULL;
        grew = true;
        return true;
    }
    else
    {
        int comparison = strcmp(rootp->label, newLabel);
        if ( comparison < 0 )  // rootp->label < newLabel
            if (! insert (rootp->right, newLabel, grew))
                return false;
        else
        {
            if (grew)                  // This may change grew
                adjustAfter_rightInsert(rootp, grew);
            // No else because if grew is false then the right
            // subtree didn't grow so no adjustment is needed.
            return true;
        }
        else if ( comparison > 0 ) // p->label > newLabel
            if (! insert (rootp->left, newLabel, grew))
                return false;
            else    // Same comments as above.
            {
                if (grew)
                    adjustAfterLeftInsert(rootp, grew);
                return true;
            }
        else
            return false; //rootp->label matches newLabel
    }
} // insert
```

9.1. Removal of a Node from an AVL Tree

Now let's look at what happens when a node is removed from an AVL tree. The process is similar to the algorithm for insertion. We search down the tree for a node containing the specified key. If we find it. we copy the node into a reference parameter and then delete the node from the tree (or, if the node has two children, we replace the node by its in-order predecessor and then delete a node farther down the tree). At each stage we report upward to the parent whether the tree we have been working on stayed the same height or decreased in height by 1 . So now let's consider what happens when a parent node gets the news that one of its subtrees -- say the left one -- has "shrunk" in height.

- Suppose we have just performed a successful removal in the left subtree of a subtree whose root contains the = balance condition, and suppose we know that the left subtree's height decreased by 1.
- Suppose we have just performed a successful removal in the left subtree of a subtree whose root contains the > balance condition, and suppose we know that the left subtree's height decreased by 1.
- Suppose we have just performed a successful removal in the left subtree of a subtree whose root, call it s , contains balance condition < . Suppose, moreover, that it has been reported that the left subtree's height decreased by 1. Then we know that the subtree rooted at s is no longer height balanced, and we must act to remedy that. It turns out that there are three cases to consider:

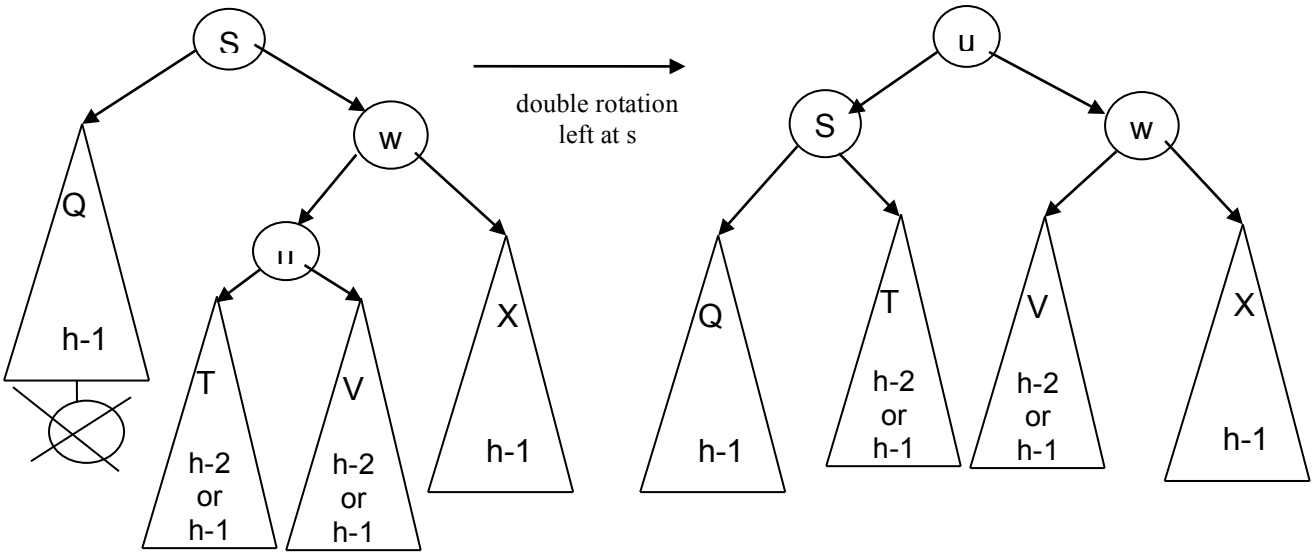
Case 1: The right child of s has balance condition < .



h = original height of the subtrees rooted at q and x .

A single left rotation at s restores the height balance of the entire subtree. This causes the entire subtree to shrink from height h+2 to height h+1 . This shrinkage will have to be reported to the next higher level (if any) in the AVL tree.

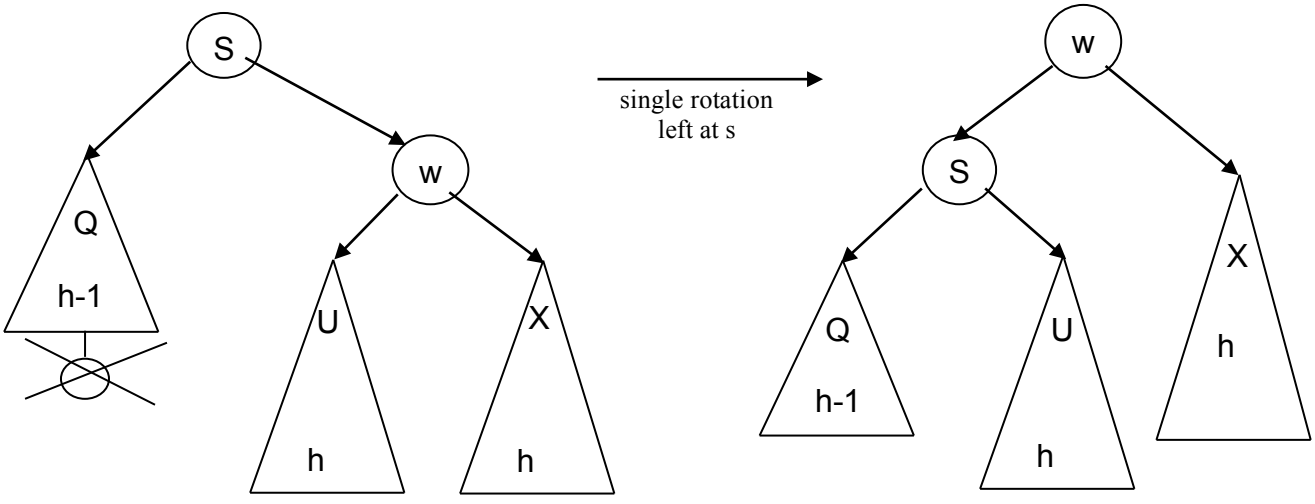
Case 2: The right child of s has balance condition >.



h = original height of the subtrees rooted at q and u.
At least one of the subtrees of u must have height h-1.

A double left rotation at s restores the height balance of the entire subtree. This causes the entire subtree to shrink from height h+2 to height h+1. This shrinkage will have to be reported to the next higher level (if any) in the AVL tree.

Case 3: The right child of s has balance condition =.

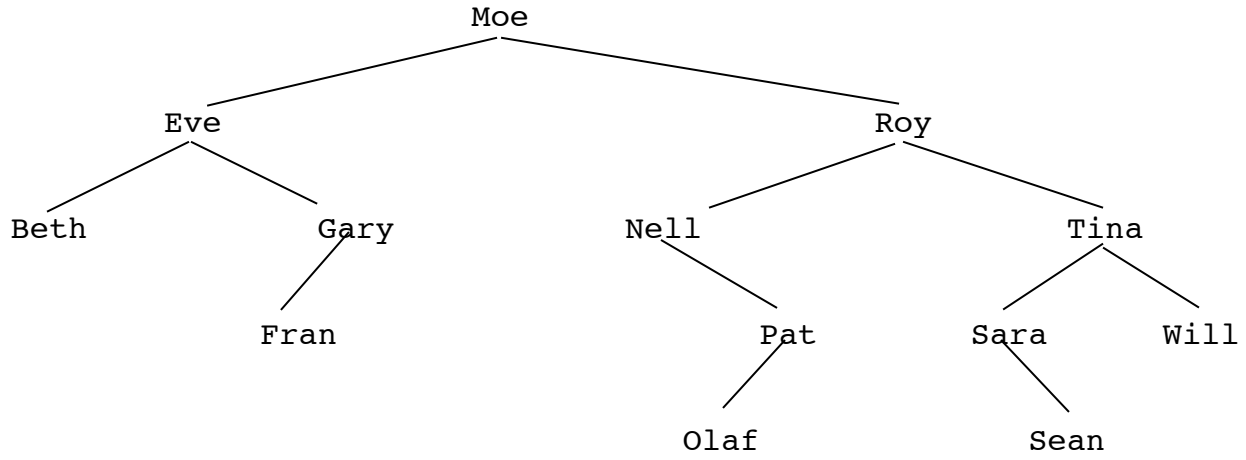


h = original height of the subtrees rooted at q and u.

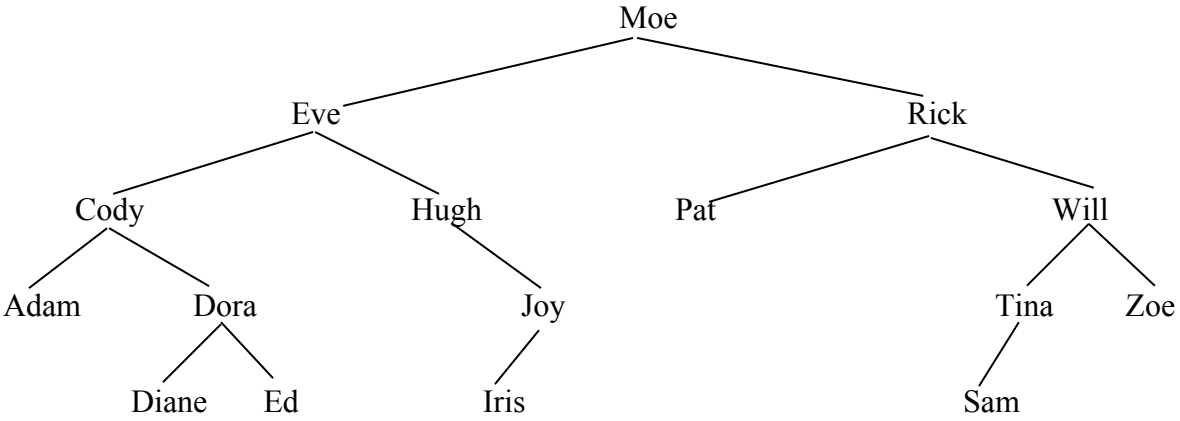
A single left rotation at s restores the height balance of the entire subtree. The height of the tree remains h+2. The fact that this tree did not shrink in height will be reported to the next level up.

Written Exercises

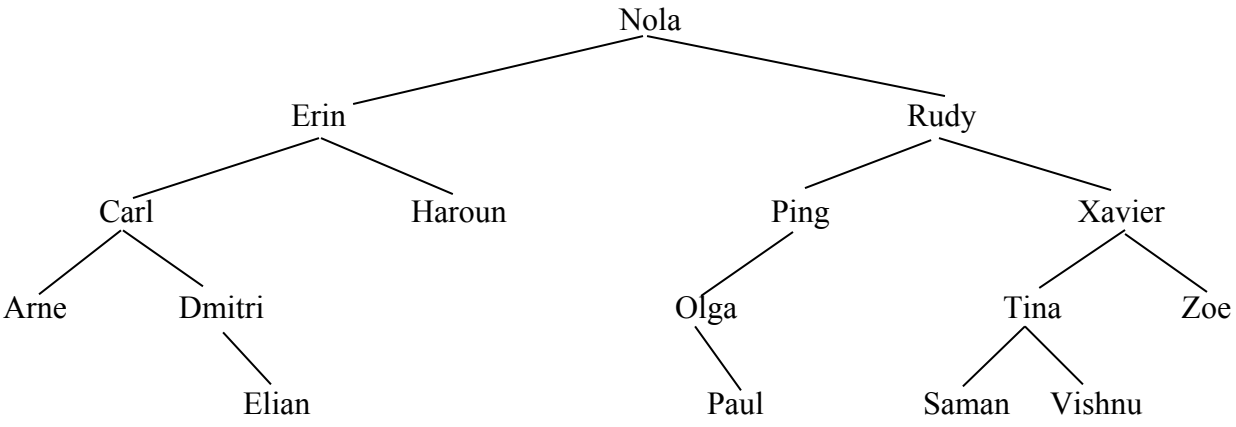
9-1. Is the binary tree shown below height balanced? If not, point out all the nodes at which the height balance condition fails to be satisfied.



9-2. Is the binary tree shown below height balanced? If not, point out all the nodes at which the height balance condition fails to be satisfied.



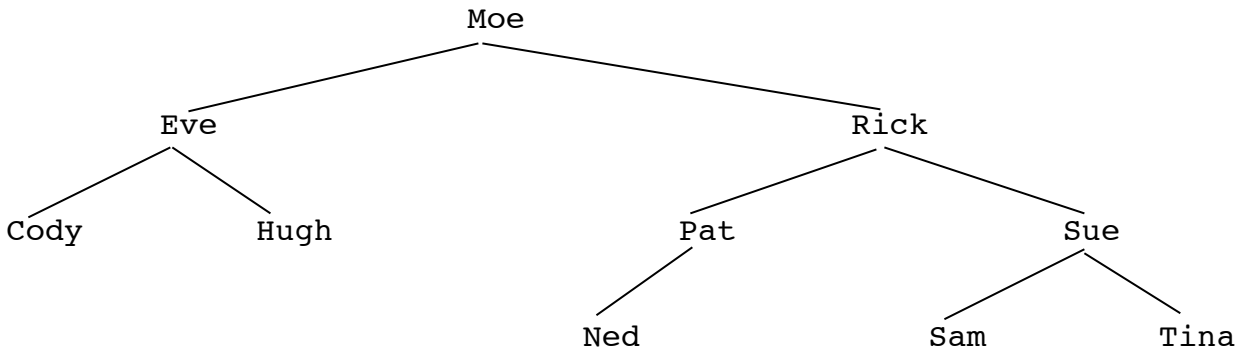
9-4. Is the binary tree shown below height balanced? If not, point out all the nodes at which the height balance condition fails to be satisfied.



9-6. What is the smallest number of nodes that can be used to build a height-balanced binary tree of height 6 ? Explain fully your reasoning. You may use computations we did in class.

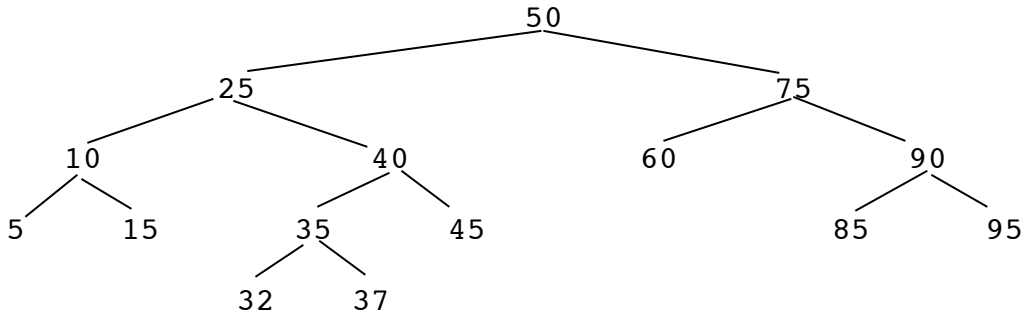
9-8. What is the smallest number of nodes that can be used to build a height-balanced binary tree of height 7 ? Explain fully your reasoning. You may use computations we did in class.

9-9. The binary tree shown below is an AVL tree.



(a) Show what form the tree will have after "Fred" is inserted, while keeping the tree an AVL tree.
(b) Starting from the original tree (without Fred), show what form the tree will have after Roy is inserted, while keeping the tree an AVL tree.
(c) Starting from the original tree (without Fred or Roy), show what form the tree will have after Olga is inserted, keeping the tree an AVL tree.

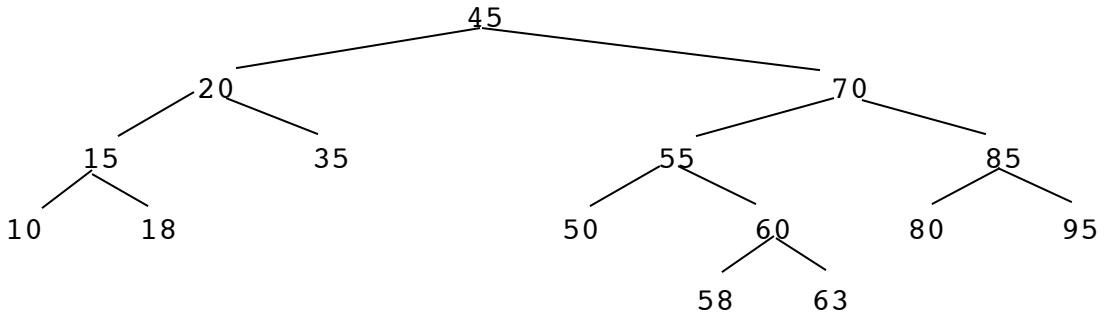
9-10. The binary tree shown below is an AVL tree.



(a) Show what form the tree will have after 65 is inserted, while keeping the tree an AVL tree.
(b) Starting from the original tree (without 65), show what form the tree will have after 36 is inserted, while keeping the tree an AVL tree.
(c) Starting from the original tree (without 65 or 36), show what form the tree will have after 88 is inserted, keeping the tree an AVL tree.

09-12

9-12. The binary tree shown below is an AVL tree.



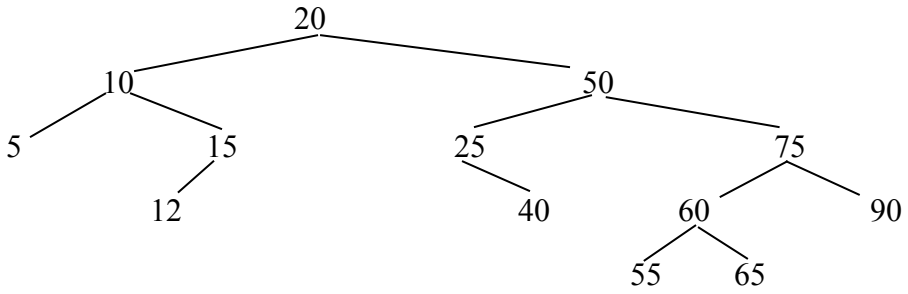
(a) Show what form the tree will have after 30 is inserted, while keeping the tree an AVL tree.

(b) Starting from the original tree (without 30), show what form the tree will have after 61 is inserted, while keeping the tree an AVL tree.

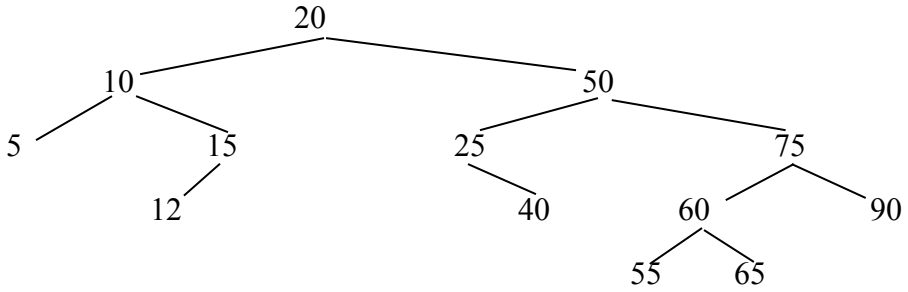
(c) Starting from the original tree (without 30 or 61), show what form the tree will have after 17 is inserted, keeping the tree an AVL tree.

9-13. In each case below, verify for yourself that the tree exhibited is an AVL tree (only the keys are shown). Then show what happens when a node with the indicated key is inserted. You don't need to add all the balance codes to the trees or consult the rules on page 9-6 in the class notes. After performing an insertion, just go up the tree from the inserted node and visually inspect each ancestor for height balance. If/when you come to an ancestor node where height balance is now violated, just figure out what rotation is required and perform it.

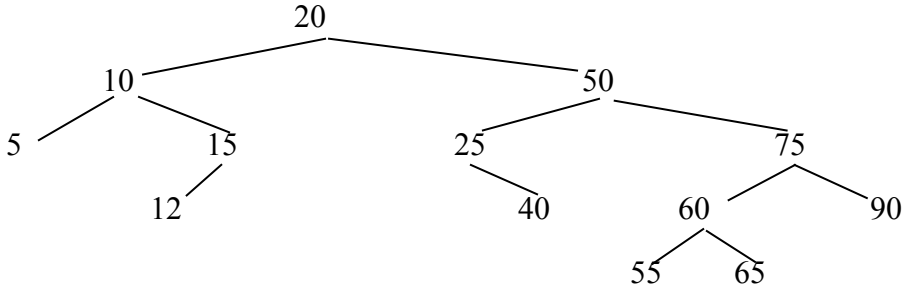
(a) Insert the key 52 into the tree below



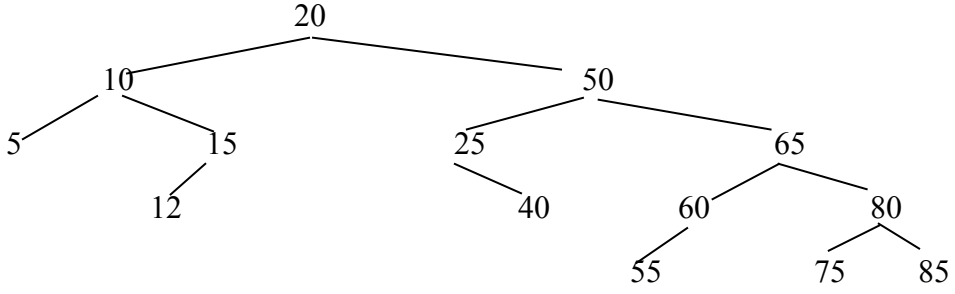
(b) Insert the key 95 into the tree below.



(c) Insert the key 14 into the tree below

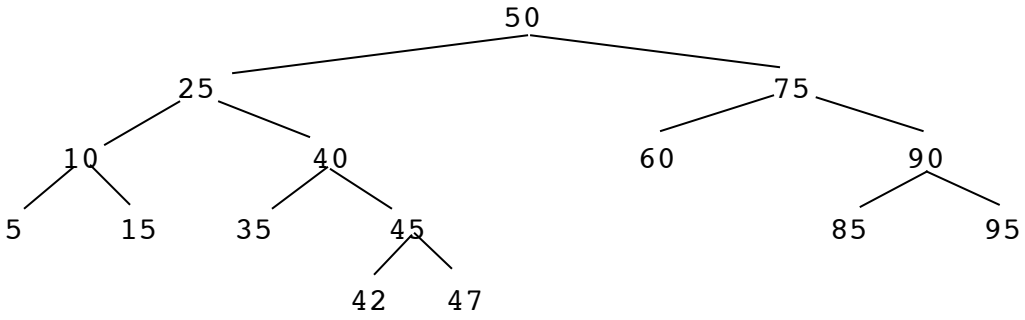


(d) Insert the key 70 into the tree below

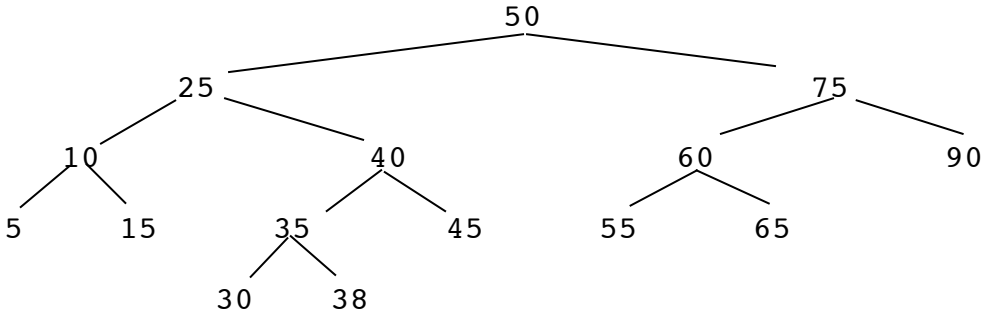


9-14. In each case below, verify for yourself that the tree exhibited is an AVL tree (only the keys are shown). Then show what happens when a node with the indicated key is inserted. You don't need to add all the balance codes to the trees or consult the rules on page 9-6 in the class notes. After performing an insertion, just go up the tree from the inserted node and visually inspect each ancestor for height balance. If/when you come to an ancestor node where height balance is now violated, just figure out what rotation is required and perform it.

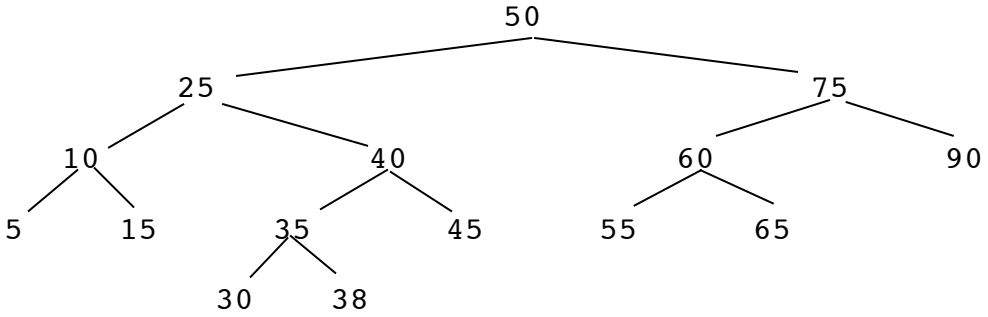
(a) Insert 17 into the AVL tree below.



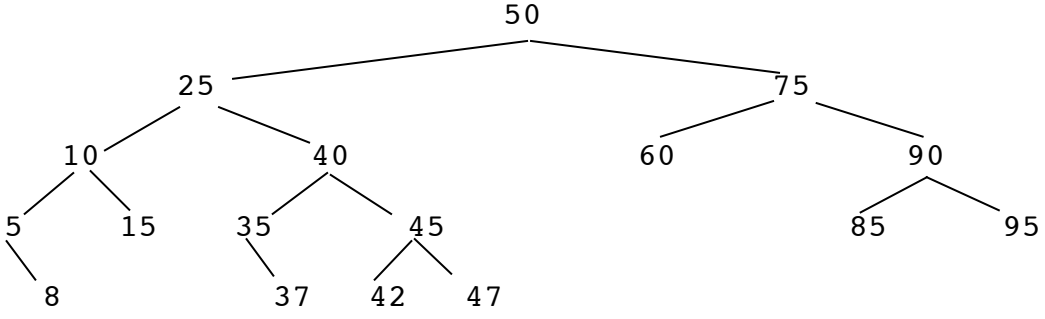
(b) Insert 34 into the AVL tree below.



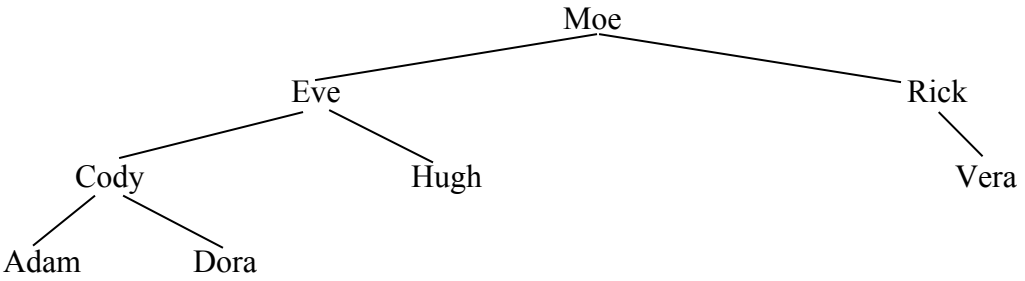
(c) Insert 61 into the AVL tree below.



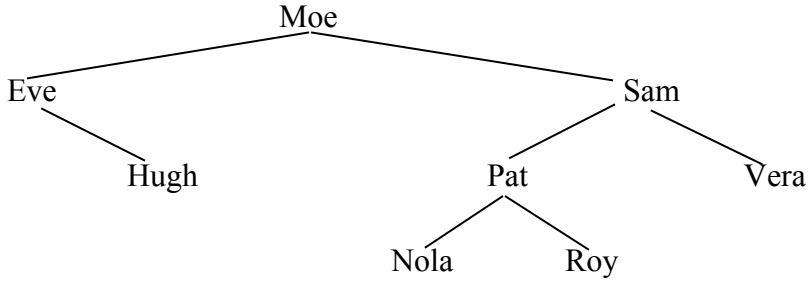
(d) Insert 44 into the AVL tree below.



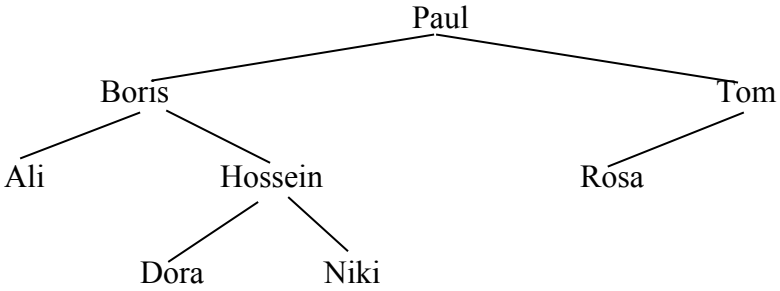
9-17. The tree shown below is an AVL tree. Show what form the tree will have after Rick is removed.



9-18. The tree shown below is an AVL tree. Show what form the tree will have after Eve is removed.

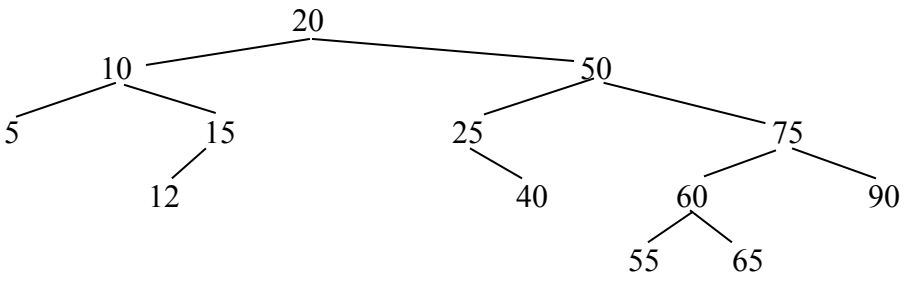


9-20. The tree shown below is an AVL tree. Show what form the tree will have after Tom is removed.

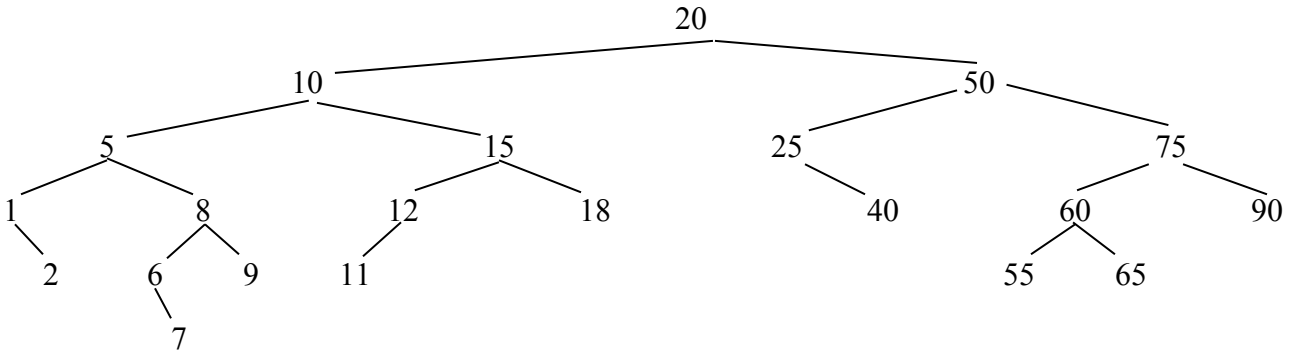


9-21. In each case below, verify for yourself that the tree exhibited is an AVL tree (only the keys are shown). Then show what happens when a node with the indicated key is removed. You don't need to add all the balance codes to the trees. After performing a removal, just go up the tree from the parent of the physically removed node and visually inspect each ancestor for height balance. If/when you come to an ancestor node where height balance is now violated, just figure out what rotation is required and perform it.

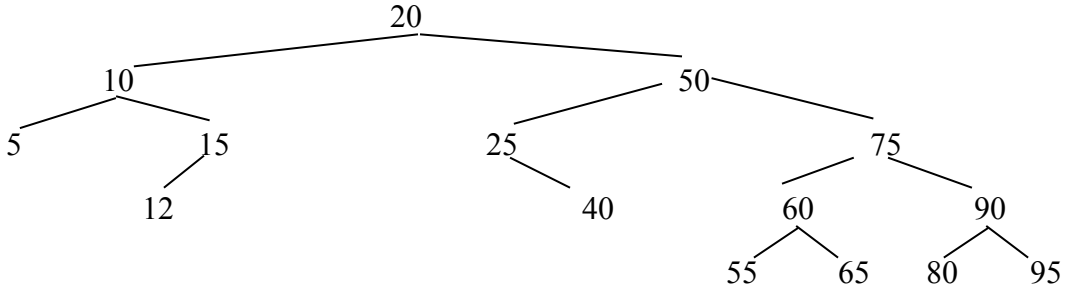
(a) Remove the node with key 12 from the tree below



(b) Remove the node with key 40 from the tree below

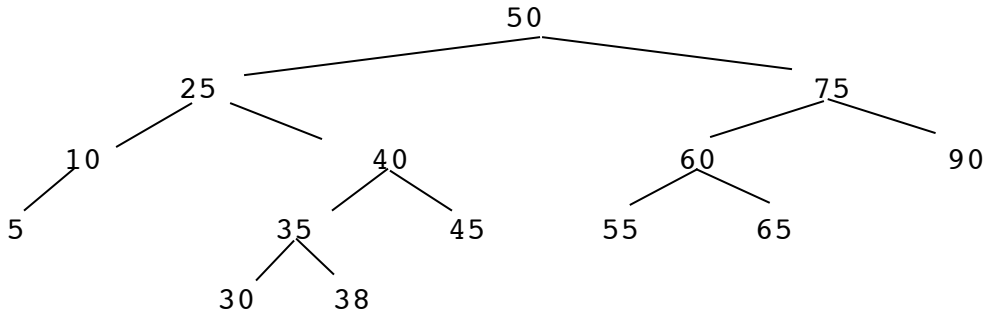


(c) Remove the node with key 40 from the tree below

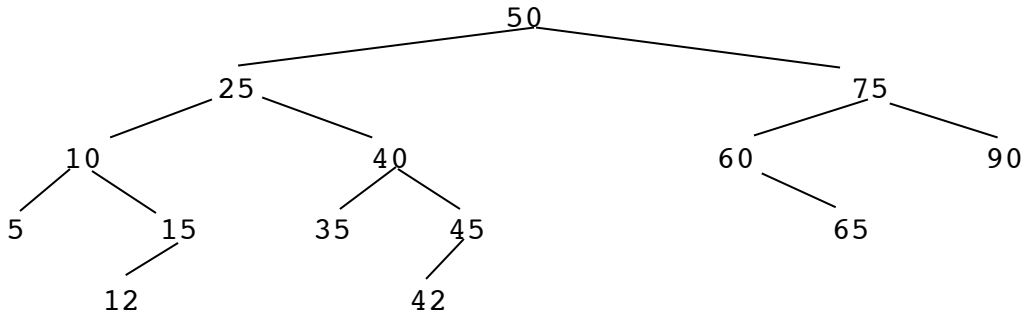


9-22. In each case below, verify for yourself that the tree exhibited is an AVL tree (only the keys are shown). Then show what happens when a node with the indicated key is removed. You don't need to add all the balance codes to the trees. After performing a removal, just go up the tree from the parent of the physically removed node and visually inspect each ancestor for height balance. If/when you come to an ancestor node where height balance is now violated, just figure out what rotation is required and perform it.

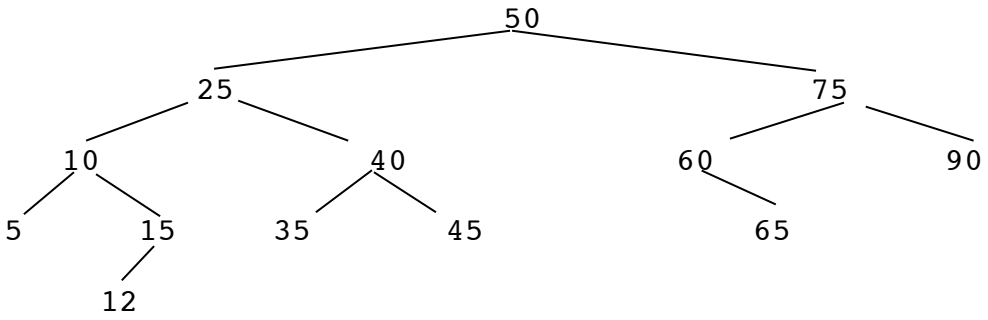
(a) Remove the node with key 5 from the tree below



(b) Remove the node with key 65 from the tree below

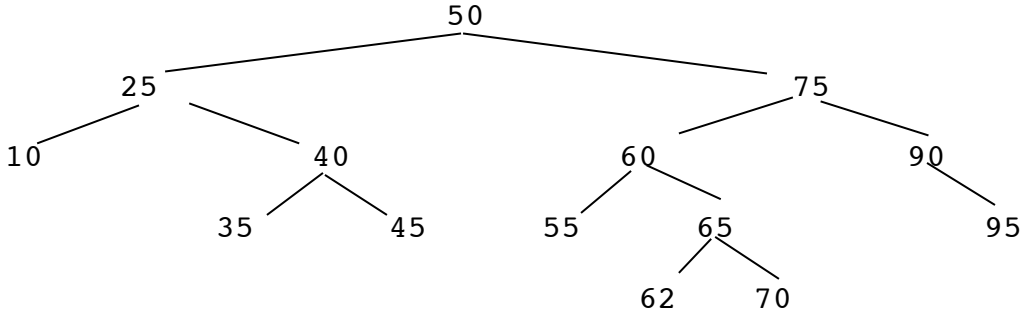


(c) Remove the node with key 90 from the tree below

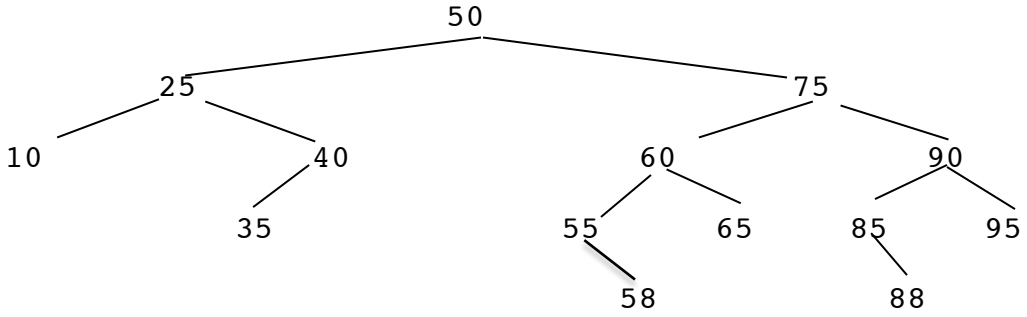


9-24. In each case below, verify for yourself that the tree exhibited is an AVL tree (only the keys are shown). Then show what happens when a node with the indicated key is removed. You don't need to add all the balance codes to the trees. After performing a removal, just go up the tree from the parent of the physically removed node and visually inspect each ancestor for height balance. If/when you come to an ancestor node where height balance is now violated, just figure out what rotation is required and perform it.

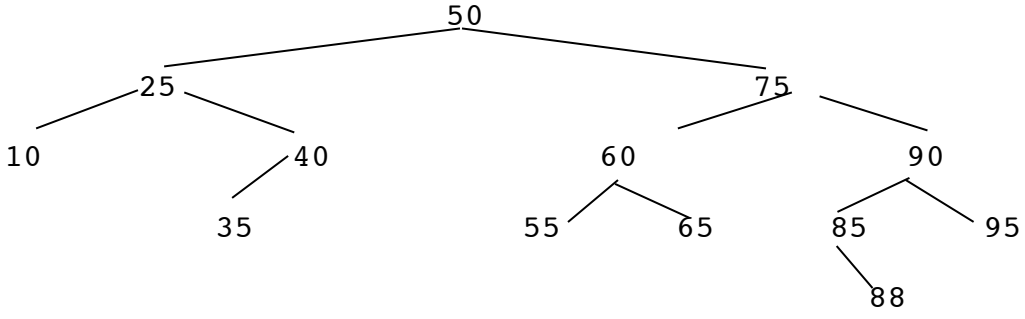
(a) Remove the node with key 95 from the tree below



(b) Remove the node with key 35 from the tree below



(c) Remove the node with key 10 from the tree below



9-26 Assume that we have an AVL tree on which we are performing a sequence of operations. Suppose that a remove has just occurred in the *left* subtree of a node s , causing the left subtree of s to shrink in height by 1 . Let w denote the right child of s and let u denote the left child of w (as in the diagrams on pages 9-8 and 9-9). FILL IN THE BLANKS BELOW:

If the balance at s is '=', change it to '_____' and report {*shrinkage* or *no shrinkage*} to parent.

else if the balance at s is '>', change it to '_____' and report _____ to parent.

else (the balance at s must be '<', so the tree has now become unbalanced):

if the balance at w is '<',

--> perform a single left rotation at s ;

--> change the balance at s to '_____';

--> change the balance at w to '_____';

--> report _____ to parent.

else if the balance condition at w is '>',

--> perform a double left rotation at s ;

--> if the balance condition at u (the new root) is '<',

change the balance at s to '_____';

change the balance at w to '_____';

change the balance at u to '_____';

--> else if the balance condition at u (the new root) is '>',

change the balance at s to '_____';

change the balance at w to '_____';

change the balance at u to '_____';

--> else (the balance condition at u (the new root) must be '=')

change the balance at s to '_____';

change the balance at w to '_____';

change the balance at u to '_____';

--> report _____ to parent.

else (the balance condition at w must be '='):

--> perform a single left rotation at s ;

--> change the balance condition at s to '_____';

--> change the balance condition at w to '_____';

--> report _____ to parent.

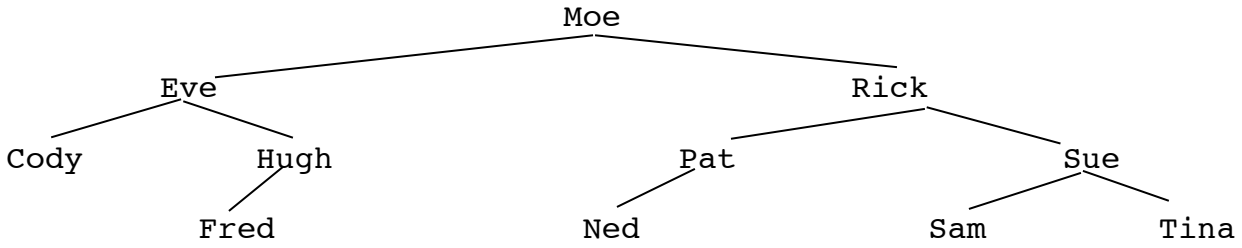
9-28. Starting from an empty AVL tree, perform the following operations. Re-draw the tree after each rotation. [NOTE: these operations are appear in an input file that will be used by your instructor to test the AVL tree operations you have been asked to write. By working out all these steps and then looking at the output from the file, you will be able to test your AVL operations thoroughly.]

Insert (in this order) 36, 48, 74, 62, 86, 70, 54, 40
Remove (in this order) 74, 70, 86, 62, 54
Insert (in this order) 12, 20, 82, 78, 6, 16, 2, 96, 28
Remove (in this order) 6, 2, 82, 48, 96
Insert 90
Remove 12
Insert (in this order) 56, 93, 84, 46, 72, 76
Remove 28
Insert (in this order) 66, 60
Remove (in this order) 20, 36
Insert (in this order) 68, 64
Remove (in this order) 93, 84, 72
Insert 58
Remove (in this order) 64, 46, 16, 68, 90, 76
Insert (in this order) 57, 59
Remove 78

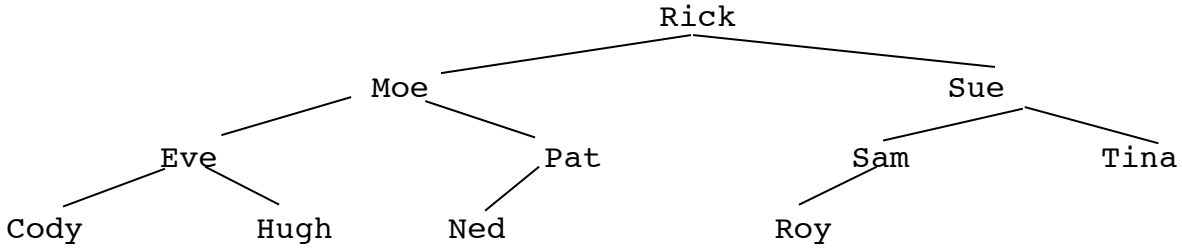
Solutions to Odd-Numbered Exercises

9-1. The tree is not height balanced. The height balance condition fails at the node `Nell`: the left subtree of that node is empty, so it has height -1; the right subtree has height 1. Since these heights differ by more than 1, the height balance condition fails at `Nell`.

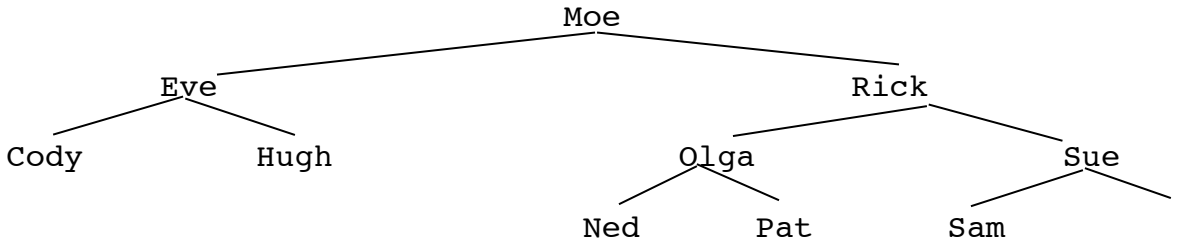
9-9. (a) `Fred` will be inserted as the left child of `Hugh`. This will not cause any height-balance violation, so no rotation will be necessary.



(b) `Roy` will be inserted as the left child of `Sam`. *Going back up the tree*, this will not cause a height-balance violation at Sam, or at Sue, or at Rick. But at `Moe` there will be a height-balance violation. Since both `Moe` and `Rick` have balance condition `<`, a single left rotation at Moe will take care of the problem. The resulting tree is

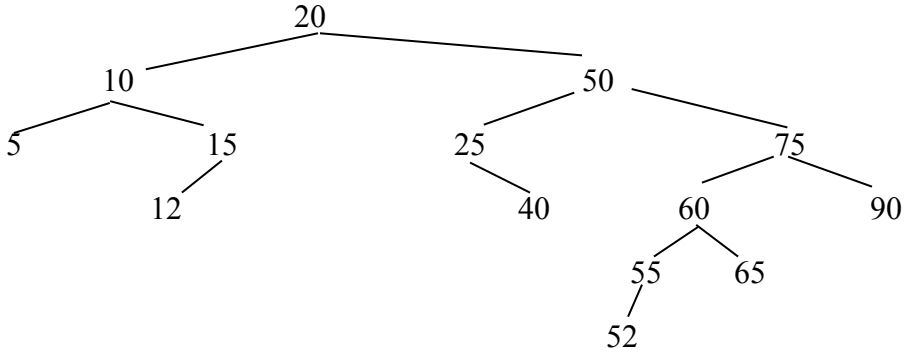


(c) `Olga` will be inserted as the right child of `Ned`. *Going back up the tree*, this will not cause a height-balance violation at `Ned`, but at `Pat` there will be a height-balance violation. Since `Pat` has balance condition $>$ and `Ned` has balance condition $<$, this is the "zig-zag" case and requires a double right rotation at `Pat` to take care of the problem. The resulting tree is

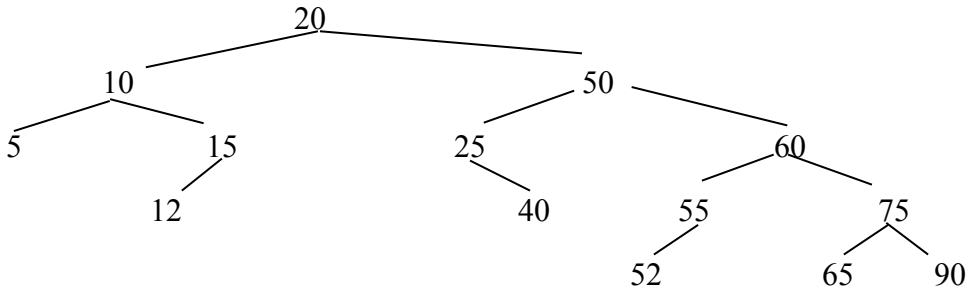


Since the height of the left subtree of Rick has returned to what it was before the insertion, there will be no further changes at any higher point in the tree.

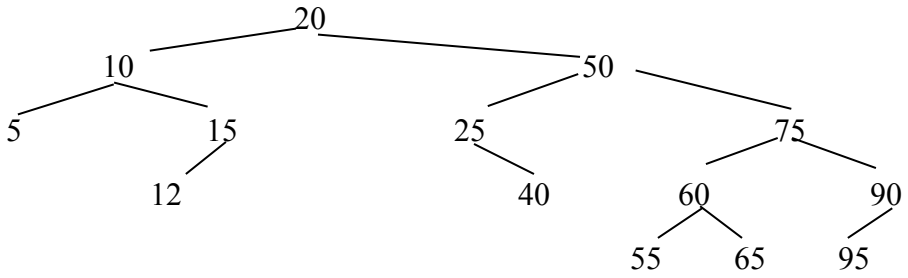
9-13. (a) The key 52 becomes the left child of 55 .



There is no violation of the height balance condition at 55 or 60 , but there is a violation at 75. A single right rotation at 75 rebalances the tree.

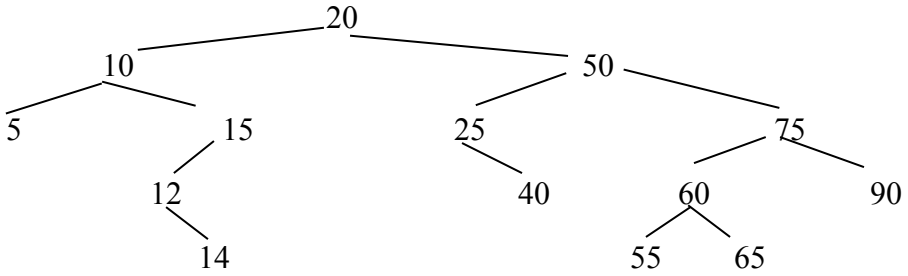


(b) The key 95 becomes the right child of 90 .

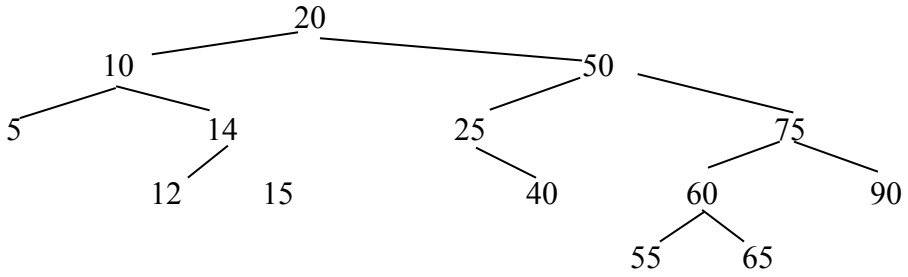


This produces no height balance violation anywhere in the tree.

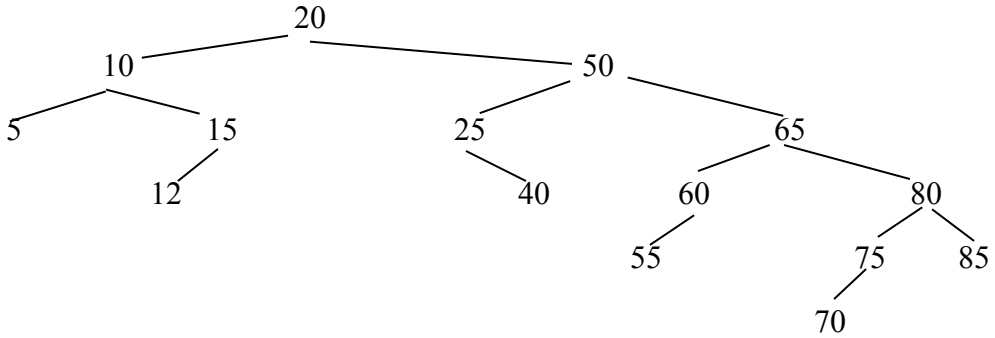
(c) The key 14 becomes the right child of 12 .



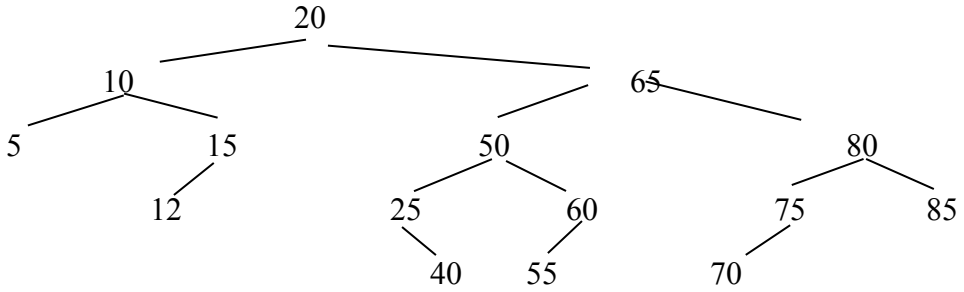
This causes a violation of the height balance condition at 15 . This is the zig-zag case, so a double right rotation is required at 15 to rebalance the tree. (See next page.)



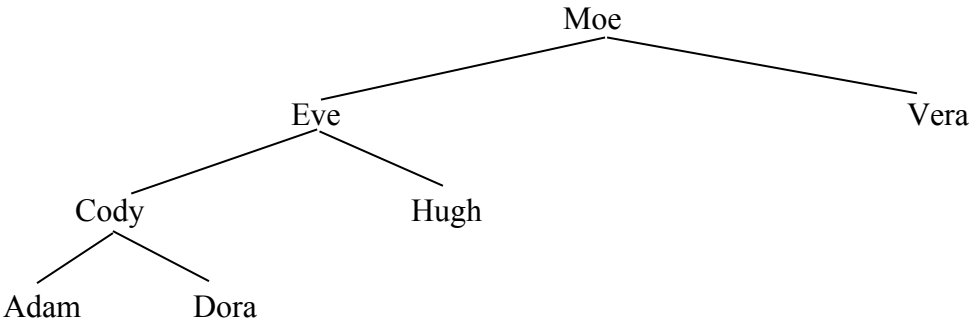
(d) The key 70 becomes the left child of 75 .



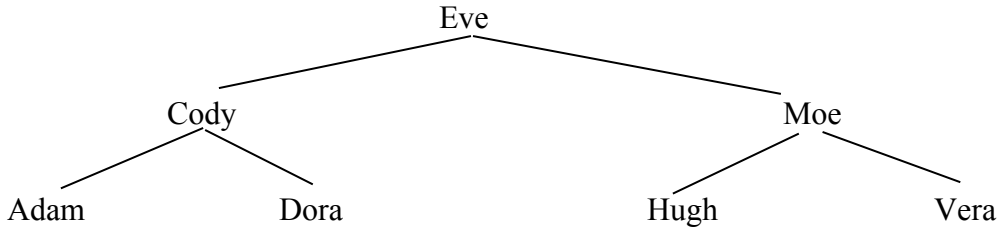
This causes no violation at 75 or at 80 or at 65 , but at 50 there is a violation. This is the "zig-zig" (straight down to the right) case, so a single left rotation at 50 rebalances the tree.



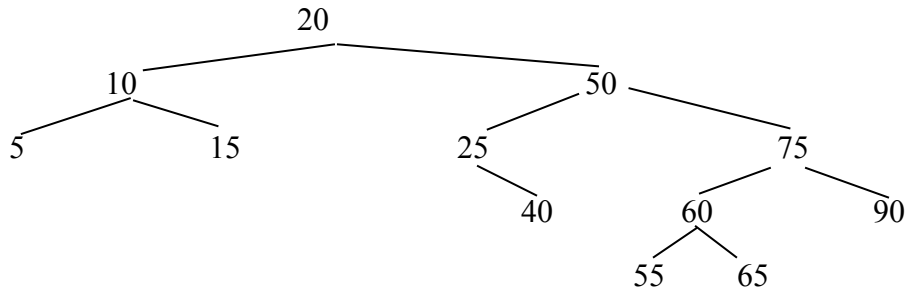
9-17. When Rick is removed, Vera becomes the right child of Moe. There is a height violation at Moe.



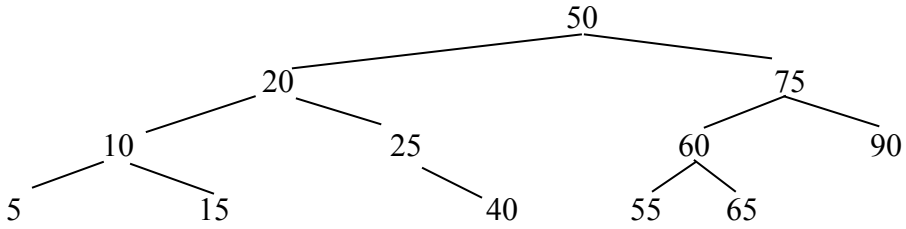
This is the "zig-zig" (straight down to the left) case, so a single right rotation at Moe is needed. The resulting tree is shown on the next page.



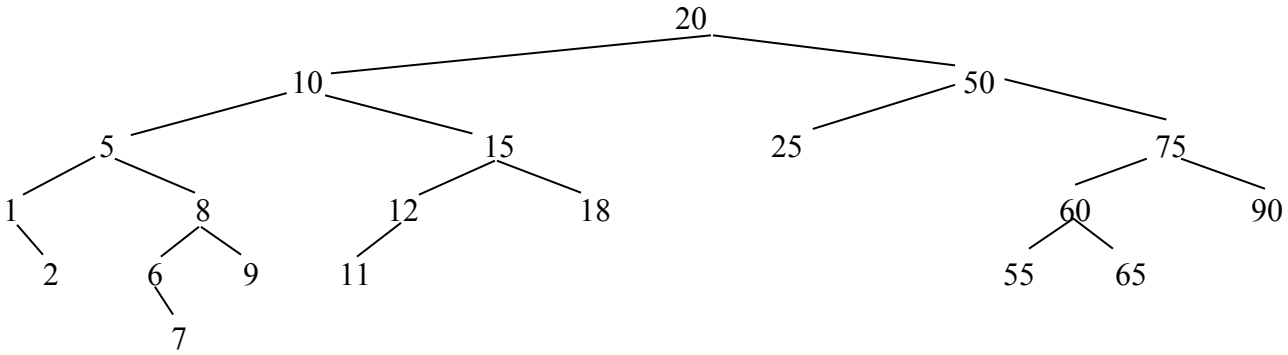
9-21. (a) When 12 is removed from the tree, this produces no violation at 15 or at 10 , but there is a violation at 20 .



A single left rotation at 20 rebalances the tree.

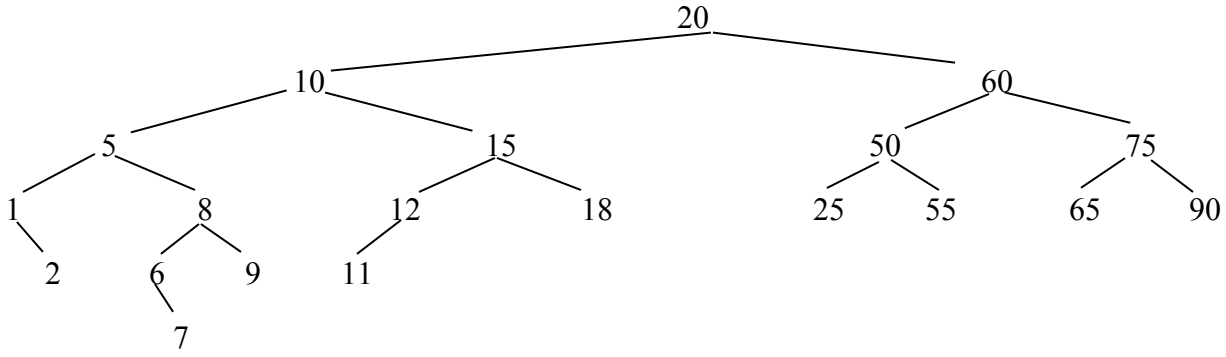


(b) When 40 is removed from the tree, this produces no violation at 25 , but there is a violation at 50 .

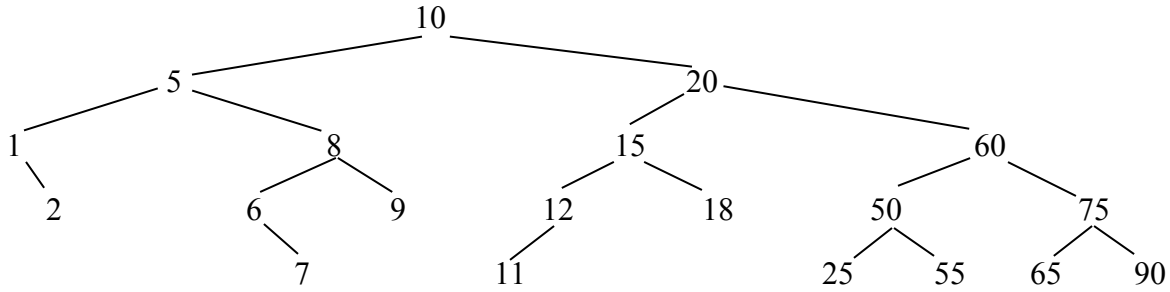


A double left rotation at 50 is required at that node (see next page).

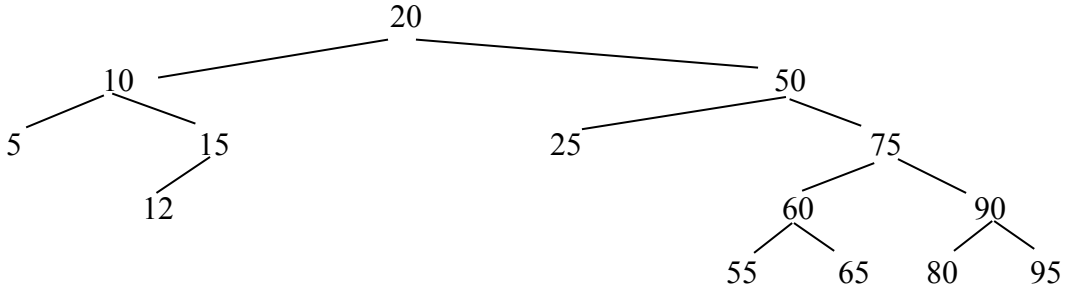
09-24



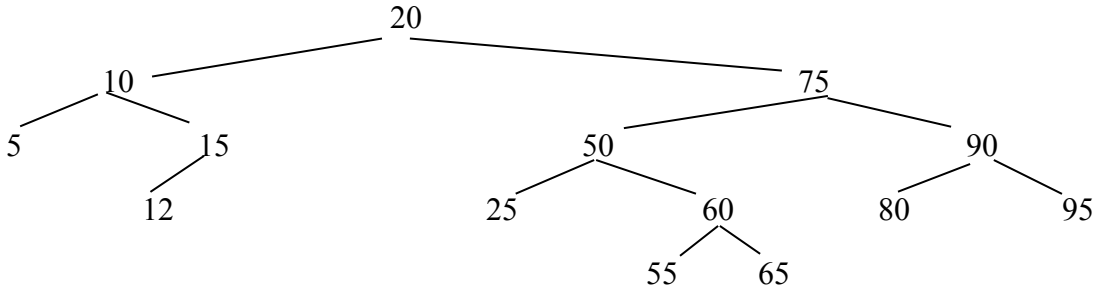
***<u>Shrinkage is now reported up to 20 , where there is another violation</u>*!!!!** This requires a single rotation right at 20 , which produces this tree.



(c) When 40 is removed from the tree, this causes no violation at 25 , but a violation occurs at 50 .



Since the balance condition at 75 is an = , a single left rotation at 50 is needed.



10. The "Priority Queue" ADT

Definition: A **priority queue** is a collection of data objects of any kind, each of which has a "priority" value associated with it (the priority value can be part of the data object, or it may be a value that's assigned to it at the moment when it is placed into the priority queue). The priority values belong to some *ordered* data type, e.g. integers (the most common case), or characters, or strings, or floating point numbers. Objects (and their priorities) can be added to a priority queue in any order, but when a remove operation is performed on the priority queue, the object that's removed must be an object having the "highest" priority among all the objects in the priority queue. Sometimes an additional requirement is imposed on the object being removed: if there are several objects of the same highest priority, then the one that has been in the collection the longest is the one that must be removed. That is, within each priority level, we use the FIFO discipline.

The phrase "highest priority" must be carefully understood. If the priorities that are associated with the objects are positive integers, then in some situations the number 1 would be treated as the highest priority, the number 2 would be treated as having the next highest priority, and so on. Thus "highest" may not always mean largest. This leads to the following supplementary definition.

Definition: A priority queue is called a **priority max-queue** iff the larger priority values are considered to have higher priority than the smaller priority values. A priority queue is called a **priority min-queue** iff the smaller priority values are considered to have higher priority than the larger priority values. The remove operation on a priority max-queue is sometimes called "DELETE-MAX", and the remove on a priority min-queue is sometimes called "DELETE-MIN".

How do priority queues occur in the real world? A typical example is a hospital emergency room. When patients arrive they are assigned some kind of priority (perhaps a somewhat informal one) based on an initial evaluation of the patient. There are many other similar situations in which tasks must be performed in an order that does not necessarily correspond to the order in which the tasks arise. You will also see other examples later in solving certain kinds of operations research problems in finite graph theory (e.g., the shortest path problem, or the maximum flow problem, or the minimum spanning tree problem) where it is useful to be able to access quickly the smallest element (say) in some collection, and then the next smallest, and so on.

The definition of priority queue specified only two operations: insert and remove. These are the primary ones that make the collection a priority queue. Other operations could also be defined as well. Some that come to mind are

(1) a construction operation that creates a priority queue out of a collection of objects in some random order;
(2) a test for emptiness;
(3) an operation to examine -- without removing -- an object of highest priority;
(4) an operation to make a priority queue empty.

10-2

Let's think of some possible implementations of the ADT "priority queue".

arrays?

linked lists?

a small array of _____

binary trees of some kind?

Definition: A **complete binary tree** is a binary tree with the following two properties:
(1) every level of the tree is completely filled, except possibly the bottom level, and
(2) all the nodes in the bottom level are placed as far to the left as possible.

Question: What is the height of a *complete* binary tree with n nodes?

Definition: A **binary max-heap** is a binary tree with the following properties:
(1) each node of the tree contains an object having a priority value in some ordered data type;
(2) at each node, the priority of the object stored at that node is *at least as large* as the priority of all its descendants in the tree; and
(3) the tree is complete.

A **binary min-heap** is a binary tree with properties (1) and (3), but at each node the priority of the object stored there is *at least as small* as the priority of all its descendants in the tree.

It is worth noting that for a tree to have property (2), it is sufficient to require simply that at each node, the priority of the object stored at that node is at least as large as the priority of its children (if any). A similar remark holds for binary min-heaps.

Why are we interested in binary max-heaps and binary min-heaps? The answer is that they give us a good way of implementing the priority queue ADT. To remove an object with highest priority from a binary heap, we simply remove the object at the root of the tree and then repair the damage to the tree in some efficient way. To insert an object into a binary heap, we put it initially into the first open position in the complete binary tree, and then we make it fight its way upward to an appropriate level in the tree. If we implement this properly, the amount of time required for an insert or a remove on a binary heap with N nodes should be at worst proportional to $\log_2(N)$.

It is possible to implement the ADT "binary tree" in an array, without using any "child pointers". We place the root at location 1, the left child of the root at location 2, the right child of the root at location 3, and, in general, for a node stored at location k we place its left child (if any) at location $2k$ and the right child (if any) at location $2k+1$. Cells that do not correspond to nodes in the tree must be marked in some way as empty. This implementation can be very wasteful of space (a binary tree with N nodes can require up to $2^N - 1$ array cells for storage in the worst case). If, however, a binary tree is *complete*, then all N nodes can be stored in just the N locations from 1 to N, which makes this an extremely space-efficient storage structure for a binary tree. It can also be quite time efficient, as we shall now see when we look at the insert and remove operations.

10.1. Insertion into a Binary Heap

Assume that we have a binary heap implemented using an array, and suppose that at some particular time there are N objects in the heap. Suppose we now wish to insert an object x into the heap. We initially insert it at location N+1 in the array, which corresponds to the first "open" position at the bottom level of the tree. (Note that we do *not* have to make a search starting at the root to find that first open position.) Of course, it is fairly likely that x will have a higher priority than its parent (after all, low priority objects are in the lower levels of the tree), so we now compare x with the object in the parent node, which is located at position $\in (N+1)/2\_$. (The "floor" symbols $\in \_$ tell us to "discard any fractional part of the quantity inside.") If we find that x is not in correct position relative to its parent, we can swap x and its parent object, after which we will again need to compare x with its new parent. This continues up the tree until x arrives at the appropriate level. (Some computer scientists refer to this process as "allowing x to percolate up the tree" or "allowing x to sift up the tree".) How much time does an insertion operation require?

CODING NOTE: the algorithm above talks about "swaps". This is a "logical" view of the matter. When we code the algorithm, we keep x in the cell with index 0 and repeatedly compare x with what *would* be its parent if it were in the heap; whenever it has higher priority than its parent, we move the parent down. Only when we find where x actually belongs do we put x into the array.

10.2. Removal from a Binary Heap

Suppose we want to perform a remove operation on a heap with N objects, implemented as an array. We remove (copy out) the object that's in the root position because it has highest priority in the heap, but then we have a "hole" in the tree that probably needs to be repaired. How do we do that?

One way that might occur to you right away is to have a contest between the children of the root to see which one should move up to root position. Then the children of the moved node could contend to see which of them moves up. The problem with this is that at each stage we create a hole in the tree at a new position, and we have no guarantee that the hole will end up where we need it, namely at the right end of the bottom level of the tree. That is, this algorithm may destroy the *completeness* property of the tree.

It turns out that a better way to handle the repair problem is to move the object, call it y , that's in location N of the array into the root position temporarily. This creates a *complete* binary tree with N−1 nodes, which is one property that we want, but it places y at what is almost certainly the wrong level. Now, however, we can let y "sift down" to the level where it belongs in the heap. This is done by allowing y and its children all to compete with each other each time y arrives at a new level. The winner is swapped into the parent position. At all times the tree remains complete. (This process is called "restoring the heap property at y", or "allowing y to percolate down the tree", or "allowing y to sift down the tree.") How much time does a removal operation require?

CODING NOTE: As before, the "swaps" are logical swaps. When coding this algorithm, y should be moved only when it is known where it should end up.

10.3. Initialization of Priority Queues Implemented as Binary Heaps

It is often the case that in situations where the need for a priority queue arises, we will have at hand a large number (say N) of objects that we want start up the priority queue with. That is, we want to get all these objects into the priority queue before we perform even a single remove. The most obvious way to get the priority queue going with these N objects is to create an empty priority queue and then perform the insert operation on each of the N separate objects. If we have implemented the priority queue as a binary heap in an array, then *in the worst case*, the amount of time that this will take is roughly proportional to

$$\log_2(1) + \log_2(2) + \log_2(3) + \ldots + \log_2(N)$$

As we will see later in C455, this sum is $\Theta(N \log(N))$. Thus this "naive" algorithm for initializing a priority queue implemented as a binary heap in an array requires $\Theta(N \log(N))$ time in the worst case. In the best case the running time is $\Theta(N)$ (no percolating up occurs on any insertion).

It turns out that there is a more efficient algorithm that should be used when the number N is large. It works like this: place all N objects in any order into the array that will be used for the binary heap. Then, beginning near the bottom of the tree, "restore the heap property" at every node in the tree. This algorithm is sometimes called "heapifying" the array.

Example. Suppose we have a collection of characters that we want to place initially in a priority queue. In this priority queue, characters with small ASCII values will be considered to have higher priority than characters with large ASCII values, so it will be a priority min-queue. We can implement this priority queue as a binary min-heap. If the characters to be placed initially in the priority queue are

Q, W, E, R, T, Y, U, I, O, P, A, S, D, F, G, H, J, K, L, Z, X, C, V, B

then we can begin by placing them in an array in the order they are given to us:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
Q	W	E	R	T	Y	U	I	O	P	A	S	D	F	G	H	J	K	L	Z	X	C	V	B

Now we begin the heapification algorithm. All the leaves of the tree are already little heaps of size 1, so we can skip immediately to the right-most parent in the array, i.e., the parent of the character in cell 24. This is the 'S' character in cell 12. We compare 'S' with 'B' and see that 'S' is larger, so 'S' must fall to the position of 'B'. Swap them. Since 'S' now has no children, the sift-down operation with 'S' stops.

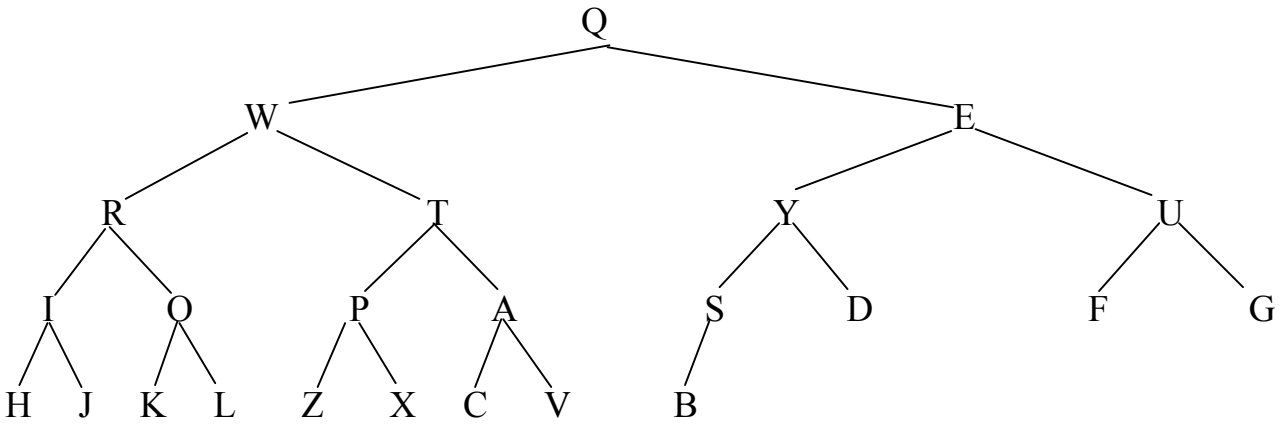
Next we move left to cell 10. Its children are in cells 22 and 23. Since 'C' is smaller than 'V', we compare 'C' with its parent and discover that the parent should not be moved.

Now we consider cell 10. Its children are in cells 20 and 21. Since 'X' is smaller than 'Z', we compare it with the parent 'P' and find that 'P' and 'X' should not be swapped.

Etc.

Theorem: The running time for the heapification algorithm on an array of N objects is $\Theta(N)$.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
Q	W	E	R	T	Y	U	I	O	P	A	S	D	F	G	H	J	K	L	Z	X	C	V	B



Written Exercises

10-1. Suppose we have implemented a priority max-queue as a binary max-heap in an array. Suppose at some point in the life of this priority queue the priorities in the array look like this:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
78	66	73	39	60	58	27	31	34	54	46	22	25	23	16	20

(a) Draw the binary tree that's represented by this array.

(b) What will the *array* look like after objects with priorities 72, 59, and 70 are inserted in that order? Be sure to show the steps in your solution to this problem so that I can follow them and (if necessary) give you partial credit if you make an error.

(c) After the insertions, suppose a remove (i.e., a "delete max") operation is performed on the priority queue. What will the *array* look like after the removal?

(d) Suppose another remove is performed. What will the *array* look like after this removal?

10-2. Suppose we have implemented a priority min-queue as a binary *min*-heap in an array. Suppose at some point in the life of this priority queue the priorities in the array look like this:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
14	34	19	51	40	22	28	75	96	63	81	26	24	37	35	80

(a) Draw the binary tree that's represented by this array.

(b) What will the *array* look like after objects with priorities 46 , 32 , and 10 are inserted in that order? Be sure to show the steps in your solution to this problem so that I can follow them.

(c) After the insertions, suppose a remove (i.e., a "delete min") operation is performed on the priority queue. What will the *array* look like after the removal?

(d) Suppose another remove is performed. What will the *array* look like after this removal?

10-4. Suppose we have implemented a priority min-queue as a binary *min*-heap in an array. Suppose at some point in the life of this priority queue the priorities in the array look like this:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
14	19	34	28	22	40	51	35	37	24	26	81	63	96	75	50

(a) Draw the binary tree that's represented by this array.

(b) What will the *array* look like after objects with priorities 23 , 16 , and 11 are inserted in that order? Be sure to show the steps in your solution to this problem so that I can follow them.

(c) After the insertions, suppose a remove (i.e., a "delete min") operation is performed on the priority queue. What will the *array* look like after the removal?

(d) Suppose another remove is performed. What will the *array* look like after this removal?

10-5. Suppose we fill an array with a collection of objects having integer priorities, and the priorities are in random order in the array as shown below.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
33	26	52	88	14	37	45	71	69	90	25	76	81	12	63	39	94	50	47	19	58	72	35	97	20	44	61

(a) Transform this array into a binary *max*-heap by carrying out the efficient "heapifying" algorithm given in class, which involves moving from right to left across the array and "restoring the heap property" at each node. Show your work as follows: reproduce the array on a sheet of paper. Then by marking out the contents of the cells and placing the new contents beneath the calls, show the stages of the algorithm. For example, if we were converting this array into a binary *min*-heap, then after several steps the display would look like this:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
33	26	52	88	14	37	45	71	69	90	25	76	81	12	63	39	94	50	47	19	58	72	35	97	20	44	61

```
                                        44----------------------------------81
                                    20------------------------------------76
                                19----------------------------90
                            47------------------------69
                        39--------------------71
                    12----------------45
                20----------------37
            39----------88
                        71----------------------88
```

Don't forget that when an object is sifting down the array, it may have to fall farther than just one level. For example, the 88 in cell 4 has to drop to cell 8 and then to cell 16 (as shown above) when forming a *min*-heap.

(b) When you have transformed the array into a max-heap, draw the binary tree represented by the array. Check to make sure that the tree you have drawn is really a binary max-heap.

10-6. Suppose we fill an array with a collection of objects having integer priorities, and the priorities are in random order in the array as shown below.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
33	62	25	88	41	73	54	17	96	39	52	67	18	21	36	93	49	15	74	91	85	27	53

(a) Transform this array into a binary *min*-heap by carrying out the efficient "heapifying" algorithm given in class, which involves moving from right to left across the array and "restoring the heap property" at each node. Show your work as follows: reproduce the array on a sheet of paper. Then by marking out the contents of the cells and placing the new contents beneath the calls, show the stages of the algorithm.

(b) When you have transformed the array into a min-heap, draw the binary tree represented by the array. Check to make sure that the tree you have drawn is really a binary min-heap.

10-8. Suppose we fill an array with a collection of objects having integer priorities, and the priorities are in random order in the array as shown below.

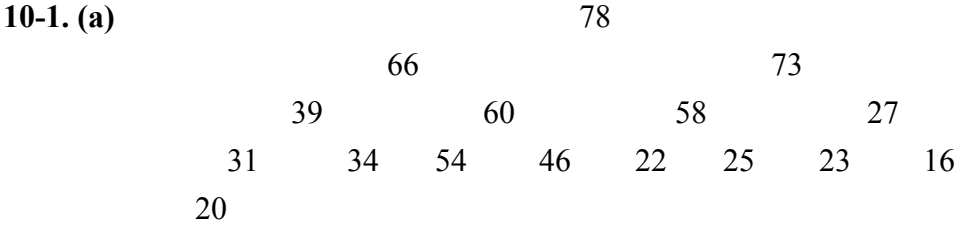
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
53	67	18	21	36	93	49	15	74	91	85	27	62	25	88	41	73	54	17	96	39	52	33

(a) Transform this array into a binary *min*-heap by carrying out the efficient "heapifying" algorithm given in class, which involves moving from right to left across the array and "restoring the heap property" at each node. Show your work as follows: reproduce the array on a sheet of paper. Then by marking out the contents of the cells and placing the new contents beneath the calls, show the stages of the algorithm.

(b) When you have transformed the array into a min-heap, draw the binary tree represented by the array. Check to make sure that the tree you have drawn is really a binary min-heap.

Solutions for Odd-Numbered Exercises

10-1. (a)



(b)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
78	66	73	39	60	58	27	31	34	54	46	22	25	23	16	20

Insert 72; , that is, place it in cell 17 and then let it percolate up as follows:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
78	66	73	39	60	58	27	31	34	54	46	22	25	23	16	20	72

```
                          72----------------------31
              72----------39
    72----66
```

The result after inserting 72 is the following array:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
78	72	73	66	60	58	27	39	34	54	46	22	25	23	16	20	31

Insert 59; that is, place it in cell 18 and then let it percolate up as follows:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
78	72	73	66	60	58	27	39	34	54	46	22	25	23	16	20	31	59

```
                                59----------------------34
```

The result is as follows:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
78	72	73	66	60	58	27	39	59	54	46	22	25	23	16	20	31	34

Insert 70; let it percolate up.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
78	72	73	66	60	58	27	39	59	54	46	22	25	23	16	20	31	34	70

```
                                70----------------------59
                    70----------66
```

The result is as follows:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
78	72	73	70	60	58	27	39	66	54	46	22	25	23	16	20	31	34	59

(c) Now remove max (the object with priority 78); the 59 in the last cell (19) is brought up to the root position and allowed to sift down, as shown below, in the array of 18 objects.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
59	72	73	70	60	58	27	39	66	54	46	22	25	23	16	20	31	34

```
73----59
```

The result is the following:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
73	72	59	70	60	58	27	39	66	54	46	22	25	23	16	20	31	34

(d) Remove max; 34 is brought up to the root position and allowed to sift down; now there are only 17 objects.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
34	72	59	70	60	58	27	39	66	54	46	22	25	23	16	20	31

```
72-34
   70----34
         66-------------34
```

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
72	70	59	66	60	58	27	39	34	54	46	22	25	23	16	20	31

<-- Final answer

10-5. (a) We start at location 27/2 = 13. We allow the 81 to sift down (but it doesn't move because it is larger than both its children). Next we move to location 12 and allow the 76 to sift down. It falls one level. Etc.

Note that when we reach cell 5 , the 14 in that cell falls TWO levels down the tree; that is, first it swaps with the 90 in cell 10 , and then it swaps with the 58 in cell 21 .
Each time you come to a new cell, let its contents fall as many levels down the tree as required to bring it to its correct position in that sub-heap.

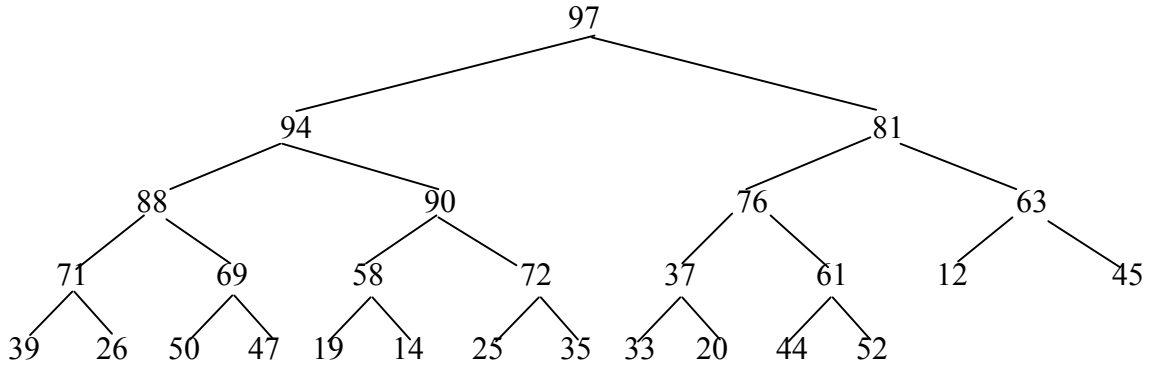
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
33	26	52	88	14	37	45	71	69	90	25	76	81	12	63	39	94	50	47	19	58	72	35	97	20	44	61

```
                                  97----------------------------------76
                              72----------------------------------25
                          94------------------------71
                       63---------------------45
                   97-----------------37
                                  76----------------------------------37
               90-------------14
                              58------------------------------14
           94----------88
       97-------52
                   81--------------------52
                                      61----------------------------------------52
   94----26
           88----------26
                       71------------------------26
97----33
       81-------33
                   76----------------33
                                  37----------------------------------33
```

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
97	94	81	88	90	76	63	71	69	58	72	37	61	12	45	39	26	50	47	19	14	25	35	33	20	44	52

10-12

(b) Here is the binary tree represented by the max-heap in the array above:



11. Sorting Algorithms

We assume that we have a set of "objects", call them x_1 , x_2 , x_3 , . . ., x_n , to be sorted (i.e., arranged in increasing order) on the basis of some property or combination of properties of the objects. In practice, the objects are records of some kind, and we sort them on the basis of one of their fields of type integer, real, or string, or on the basis of a combination of their fields (for example, electronic records for books in a library might be sorted by a primary field such as Author, and then for books having identical authors, the records might be sorted by Title.) Thus in practice it is necessary to overload the various order operators (< , <= , > , >=) so that we can compare objects directly using statements of the form "`if (x1 < x2) ...`".

We make no standing assumption about where the objects are located. They may be in an array and we may sort them by moving them around. Another possibility is that they are in an array and we sort them "exogenously" by creating a set of pointers to (or array subscripts for) the objects and moving the pointers. Still another possibility is that the objects are in the nodes of a linked list, in which case we "move" the objects in the list simply by adjusting pointers.

When we are talking about sorting objects in an array, we shall assume (usually) that the first object x_1 is in the cell with subscript 1 , and so on. The cell with subscript 0 will often used to store a "dummy object" having some useful sentinel value.

We distinguish between **internal sorting algorithms and external sorting algorithms**. An internal sorting algorithm assumes that all the objects are in RAM for fast access. An external sorting algorithm deals with the problem of sorting a list or file of objects that occupy so much space that we cannot hold them all in RAM at the same time.

The problem of space-time trade-offs appears in new forms in the problem of sorting. If we use lots of extra "work space" beyond what is already needed to store the objects, we may be able to speed up a sorting algorithm significantly. In connection with this, we speak of **in-place sorting algorithms**, by which we mean algorithms that use only a few extra auxilliary object locations beyond the memory requirements for holding all the objects. For example, Bubble Sort (to be described) is an in-place sorting algorithm for objects in an array because the only extra memory requirements are a temporary object variable for swaps and two subscript variables. A sorting algorithm is *not* an in-place algorithm if the amount of extra space required by the algorithm increases as n becomes larger.

A sorting algorithm is said to be a **stable sorting algorithm** if and only if it acts on a set with an initial linear arrangement in such a way that if two objects x_1 and x_2 are equal (i.e., the fields on which we base our comparison are identical), and if x_1 precedes x_2 in the initial arrangement (whether in an array or a linked list), then x_1 will again precede x_2 in the final arrangement. This becomes an important property of a sorting algorithm in situations where the objects are already sorted on one field, and you want to sort them on another field, but retain the initial ordering among objects whose primary fields are identical.

11.1. Selection Sort

See selection sort at page 3-5.

11.2. Heap Sort

Def. A *heap* is a perfect binary tree with the following property: every node must contain a value that is larger than all of its descendants. It can easily be proven that the root of the heap contains the highest value. See Fig. 3.1 for an example of a heap.

The *heap sort* arranges the elements of the array into a heap. The largest element is pushed to the root in the process, and can then be moved to the end of the array. The process is repeated by ignoring the last element, and after that all the elements that are already in the right place, until the array is completely sorted.

The operation of turning the array into a heap in the beginning O(n). After removing one element from the heap, the process of rebuilding the structure of the heap takes only O(log n) operations. Thus, the algorithm is of complexity O(n log n) in any case.

Representation. The heap is stored in the array starting with the root at position 0. The children of the root will occupy the positions 1 and 2, and the rest of the nodes follow in order by levels. Figure 3.1 also contains a representation of the heap in an array.

For every index in the array, its children are:

```
inline int leftChild(int i)
{
    return 2*i+1;
} // leftChild()

inline int rightChild(int i)
{
    return 2*i+2;
} // rightChild()
```

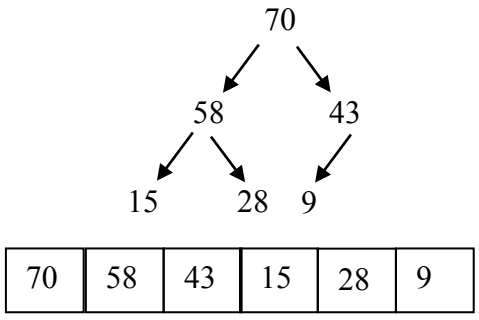


Fig. 3.1 Example of a heap

```
void heapsort(int a[], int size)
{

    // heapify the array
    for (int i=size/2; i>=0; i--)
        percolateDown(a, i, size);

    // move large elements to the back and heapify again
    for (int j=size-1; j>0; j--)
    {
        swap(a[0], a[j]);
        percolateDown(a, 0, j);
    }
} // heapsort()
```

```
void percolateDown(int a[], int i, int n)
{
    int child, temp;

    // move down in the tree while the node i still has a child
    for (temp=a[i]; leftChild(i)<n; i=child)
    {
        child = leftChild(i);

        // find the child with the largest value
        // by comparing the two children
        if (child != n-1 && a[child]<a[child+1])
            child++;

        // move the child's value up if it's larger than the parent
        if (temp < a[child])
            a[i] = a[child]; // now a[child] is a free spot
        else
            break;
    }

    // write the percolated value in the free spot
    a[i] = temp;
} // percolateDown()
```

11.3. Insertion Sorts

The general idea of an Insertion Sort on an array is that throughout the sorting process, there is a sorted portion of the array and an unsorted portion. In this sense it is similar to Selection Sort, which we studied earlier. The difference is this: while in Selection Sort we make a pass over the *entire unsorted* portion of the array to find the next object to be added to one end of the sorted portion, in Insertion Sort we pick an object from the unsorted portion of the array and then make a pass over the sorted portion to see where to insert the object among the sorted items. Generally this insertion requires us to shift a number of items in the sorted part of the array to make room for the object being inserted. By contrast, in Selection Sort we are able to use a single swap to get another object into the sorted portion of the array. Thus Insertion Sort typically makes many more object moves than Selection Sort but only about half as many object comparisons on average.

Here is some C++ code for the simplest version of Insertion Sort. The reason for the name "linear" below will be explained later when we look at another version of Insertion Sort. The algorithm sorts an array `a[1...n]`. It begins by treating the first cell as an ordered subarray of length 1, and the subarray `a[2...n]` as the unsorted portion of the array. Gradually the sorted portion of the array extends farther and farther to the right, and the unsorted portion shrinks away to nothing. In the code below, the subscript `p` marks the left end of the unsorted portion of the array, and the subscript `i` will be used for leftward searches across the sorted portion of the array. Each time through the outer loop, the object in cell `a[p]` is swapped leftward repeatedly until it reaches its correct position in the unsorted portion of the array.

```
template <class otype>
void simpleLinearInsertionSort (otype a[], int n)   // sorts a[1...n]
{
    for (int p = 2; p <= n; ++p)  // p marks the left end of the
    {                             // unsorted portion of the array;
        int i = p - 1;
        while (i >= 1 && a[i] > a[i+1])  // Let a[p] move to the left
        {                                // until it reaches something
            swap (a[i], a[i+1]);         // smaller or equal, or comes
            --i;                         // to the left end of a[1...n]
        }
    }
} // simpleLinearInsertionSort()
```

In the worst case, the algorithm above does the following:

when `p` is 2, the algorithm makes 1 data comparison (`a[i] > a[i+1]`) and 1 swap;
when `p` is 3, the algorithm makes 2 data comparisons (`a[i] > a[i+1]`) and 2 swaps;
when `p` is 4, the algorithm makes 3 data comparisons (`a[i] > a[i+1]`) and 3 swaps;
when `p` is 5, the algorithm makes 4 data comparisons (`a[i] > a[i+1]`) and 4 swaps;
etc.
when `p` is `n`, the algorithm makes n–1 data comparisons (`a[i] > a[i+1]`) and n–1 swaps.

Thus in the worst case the total number of data comparisons is approximately $n^2/2$ and the total number of swaps is the same. This is an "order n^2" sorting algorithm.

In the best case, for each `p` the algorithm performs only one data comparison and no swaps, so the algorithm runs in $\Theta(n)$ time. This occurs when the objects in the array are already in order.

The algorithm can be improved by replacing "swaps" with left shifts of the objects that must be moved. Each time we start the execution of the body of the outer loop, we move `a[p]` to a temporary storage location, which makes a "hole" in cell `a[p]`, and then the inner loop right-shifts each object that must be moved. When the shifts are finished, we move the object from the temporary storage location into its correct position. If we use cell `a[0]` for the temporary storage location, this will simplify the control of the inner loop, because we will no longer have to test whether the subscript variable `i` has gone past the left end of the array; if `i` reaches `0`, the data comparison will stop the loop (see the code at the top of the following page). That is, `a[0]` always holds a "sentinel value".

The worst case and best case analyses of this improved algorithm are essentially the same as the analyses of the simpler version of linear insertion sort. The difference is that we have eliminated all comparisons of the form `"i >= 1"`, and we have replaced each swap, which requires three assignment statements, with a single assignment statement. Thus the improved algorithm runs faster, but its running times grow with `n` at the same *rate* as the simpler version: order n^2 in the worst case.

It may have occurred to you that in the algorithm above, each search for the place to insert `a[p]` is conducted on the *sorted* portion of the array. Why not use *binary* search instead of *linear* search? That observation leads to the significantly faster Binary Insertion Sort algorithm. The code is given on the following page. It makes calls to the `binarySearch` function that we analyzed early in C243 (page 3-9 in the class notes). Go there to read the documentation for the function.

How much better is Binary Insertion Sort than our two Linear Insertion Sorts? A little thought shows that if all the objects are distinct then the number of object moves (assignment statements of one cell to another) will be exactly the same as with the improved version of Linear Insertion Sort. After all, for each new object `a[p]` that gets inserted into the sorted portion of the array, we must move `a[p]` into a temporary location, then make the same shifts to open up a cell for the insertion, and then copy the object into the hole. What has changed is that we usually make far fewer object comparisons because of the use of binary search. As we saw earlier, in the worst case a binary search on an array of `n` objects requires about $2 \log_2(n)$ object comparisons (the body of the loop is executed only about $\log_2(n)$ times, but the body of the loop may make `1` or `2` comparisons). Thus an upper bound for the total number of object comparisons made by Binary Insertion Sort on an array of length `n` is about $2n \log_2(n)$. When `n` is large, this is much better than the $n^2/2$ (roughly) object comparisons made by the Linear Insertion Sorts in the worst case. Adding the time required by the object comparisons in the worst case to the time required by the object moves in the worst case, we see that in the worst case the running time for Binary Insertion Sort is $\Theta(n^2)$ because of the object moves.

```
void improvedLinearInsertionSort (otype a[], int n)  //sorts a[1...n]
{
    for (int p = 2; p <= n; ++p)    // p marks the left end of the
    {                               // unsorted portion of the array.
        a[0] = a[p]; //Sentinel value in a[0] stops leftward searches
        int i = p - 1;
        while (a[i] > a[0])
        {
            a[i+1] = a[i];  // Right-shift each object that's larger
            --i;            // than a[0].
        }
        a[i+1] = a[0];   // a[i] is <= a[0], so put a[0] to its right
    }
} // improvedLinearInsertionSort()

void binaryInsertionSort (otype a[], int n)      // sorts a[1...n]
{
    for (int p = 2; p <= n; ++p)  // p marks the left end of the
                                  // unsorted portion of the array.
    {
        bool found;  // will be ignored, but must be passed
        int  k;      // will get the subscript of the insertion cell
        binarySearch (a, a[p], 1, p-1, found, k);
        a[0] = a[p];
        for (int i = p-1; i >= k; --i)
            a[i+1] = a[i];  // Right-shift objects in  a[k...p-1]
        a[k] = a[0];
    }
} // binaryInsertionSort()
```

What is the best case for binary search? As it turns out, the best case occurs when we reduce the number of right shifts to 0 on each "insertion". In order for this to occur, each binary search must end up at the right end of the sorted portion of the array. Since a binary search on a sorted array of length k requires time approximately proportional to $\log_2(k)$, we can see that altogether these binary searches require time approximately proportional to

$$\log_2(1) + \log_2(2) + \log_2(3) + \ldots + \log_2(n-1),$$

which can be shown to be $\Theta(n \log(n))$. This dominates the number of object moves (in the algorithm shown above there would be $2(n-1)$ object moves). Thus in the best case the running time for Binary Insertion Sort is $\Theta(n \log(n))$.

Curiously, when given an array that is already sorted, the Binary Insertion Sort runs significantly more slowly than either version of our Linear Insertion Sorts. Can you see why?

Is Binary Insertion Sort stable?

Is Binary Insertion Sort an in-place sorting algorithm?

Is Binary Insertion Sort an internal or an external sorting algorithm?

Can Binary Insertion Sort be written recursively?

11.4. Bubble Sorts

Below is the code for a very popular sorting algorithm. It is an "exchange sort" in which objects that are discovered to be in the wrong order with respect to each other are swapped.

```
template <class otype>
void brainlessBubbleSort (otype a[], int n)  // sorts a[1...n]
{
    for (int p = n; p >= 2; --p)      // p marks the right end of
    for (int i = 1; i <= p - 1; ++i)  // the unsorted portion of
        if ( a[i] > a[i+1] )          // the array
            swap (a[i], a[i+1]);
} // brainlessBubbleSort()
```

Each pass over the shrinking array a[1...p] moves the largest object in the unsorted portion of the array to the right-most position. In that sense it is like Selection Sort, but less efficient because of all the many swaps.

Is the algorithm shown above a *stable* sorting algorithm?

When Brainless Bubble Sort is applied to an array of length n it always makes exactly

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

object comparisons. In the best case it makes no object moves (no swaps), which occurs when the array is already sorted. In the worst case it makes the same number of swaps as object comparisons, and thus makes about $\frac{3}{2} n^2$ object moves. This occurs when the array is originally in reverse order. Thus the running time for Brainless Bubble Sort is always $\Theta(n^2)$ because of the comparisons.

It is possible to improve the brainless version of Bubble Sort by being alert on each pass to information that becomes available about which items are out of order. Suppose, for example, that on some pass across 500 objects in the unsorted portion of the array, the last 100 comparisons produce no swaps (these would be the comparisons 400vs401, 401vs402, . . ., 499vs500). Then the last 101 of those objects (in locations 400 through 500) are already in order with respect to each other. They are also larger than or equal to all the preceding 399 objects, because the pass across the first 400 objects moved the largest of those objects to position 400, and then that object is smaller than all those that follow it. Thus we can eliminate cells 400 through 500 from further consideration. This idea is exploited in the improved version of Bubble Sort shown at the top of the next page.

```
template <class otype>
void alertBubbleSort (otype a[], int n)  // sorts a[1...n]
{
    int p = n;             // p marks the right end of the
    int lastSwap;          // unsorted portion of the array

    while (p >= 2)
    {
        lastSwap = 0;

        for (int i = 1; i <= p - 1; ++i)
            if ( a[i] > a[i+1] )
            {
                swap (a[i], a[i+1]);
                lastSwap = i;
            }

        p = lastSwap;
    }
} // alertBubbleSort()
```

The worst case for the algorithm above is exactly the same as the worst case for the Brainless Bubble Sort: $\Theta(n^2)$. The best case, however, occurs when the array is already sorted. In this case the variable `lastSwap` remains at 0 throughout the inner "`for`" loop, and thus the outer loop terminates after executing just once. The running time in the best case is therefore $\Theta(n)$.

Is Alert Bubble Sort a *stable* sorting algorithm? Is it an *in-place* sorting algorithm?

The algorithm above can be improved even further by alternating left-to-right passes with right-to-left passes. On a right-to-left pass the smallest object in the array is picked up during the pass and transported all the way to the left end of the unsorted portion of the array. Thus the unsorted portion of the array is sandwiched in between two growing sorted subarrays, one at the left end and one at the right end of the array. This improved version of Bubble Sort has been dubbed the "Cocktail Shaker Sort", or simply the Shaker Sort. It performs quite well on arrays in which most of the items initially lie close to where they belong. In the best case it is a $\Theta(n)$ algorithm, but in the worst case (the objects are initially in reverse order) it is a $\Theta(n^2)$ algorithm.

Bubble Sorts tend to be slow in the worst case because they are restricted to swapping items that lie adjacent to each other. Thus no item can ever "jump" a long distance toward its final position. The next algorithm we will study (Quicksort) is an "exchange sort", like Bubble Sort, but it compares items that are far apart and allows them to be swapped when they are out of place.

11.5. Quicksort

One of the best sorting methods known was invented in the early 1960s by C.A.R. Hoare, who published his algorithm and a careful analysis of it in a research paper in 1962. He called his algorithm "Quicksort". Since that time there have been a number of small improvements on the algorithm, but the ideas are still substantially unchanged from Hoare's landmark paper.

Assume that we are given a (sub)array `a[first..last]` to sort. In the simple version of Quicksort that we will consider, the first object in the array is used to "partition" the remaining objects into those that are smaller than the first object and those that are larger than the first object. Those that are smaller are kept in the left part of the array, while those that are larger are kept in the right part of the array. Then these portions of the array are sorted recursively.

```
// The following code assumes that before the function is called, an
// object at least as large as all the objects in  a[first..last]
// is placed in cell  a[last+1] as a sentinel value to stop searches.

template <class data>
void quicksort (data a[], int first, int last)
{
    if (last <= first) // base case;  arrays of length <= 1 are sorted
        return;

    int i = first + 1,  j = last; // Initialize the search subscripts.
    while (a[i] < a[first] && i <= j) // Move i to the right until finding
        ++i;                  // an object at least as large as a[first].
                              // It's possible that i ends up at last+1.

    while (a[j] > a[first]) // Move j to the left until finding an
        --j;                // object at least as small as a[first].
                            // It's possible that j ends up at first.
    while (i < j)
    {
        swap (a[i], a[j]);   // Move the large object to the right part
                             // of the array and the small to the left.

        do                        // Resume the search for an object at
            ++i;                  // least as large as  a[first].
        while (a[i] < a[first]);

        do                       // Resume the search for an object at
            --j;                 // least as small as  a[first].
        while (a[j] > a[first]);
    }

    swap (a[first], a[j]);  // Place the partitioning object between
                            // objects less than or equal to it and
                            // objects greater than or equal to it.

    quicksort (a, first, j-1);  // Recursively sort the two subarrays
    quicksort (a, j+1, last);   // of objects smaller and larger than
                                // the partition object in cell j.

} // quicksort()
```

Example:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
35	16	52	84	33	20	35	15	23	60	31	59	48				

Is Quicksort a stable sorting algorithm?

Is Quicksort an in-place sorting algorithm?

What happens in Quicksort if all the objects are distinct and in increasing order already? That is, what is the running time for Quicksort when it is applied to an array that has already been sorted?

11.6. Merging Two Sorted Arrays

Consider the following ***merging problem***. Suppose we are given two arrays (or subarrays) whose cells contain objects of the same data type, and suppose the objects in each of the arrays are arranged in increasing order. (This assumes that the objects in the array cells are of a data type on which an order relation "<" and its associated relation "_" are defined.) Let `a[afirst..alast]` be the first array and `b[bfirst..blast]` be the second. The objects from these two arrays are to be merged into a single, ordered list in a third array or subarray, call it `c[cfirst..clast]`.

An algorithm for solving this problem is quite easy to invent. We can begin by comparing the first object in array `a` with the first object in array `b` and copying the smaller into the first cell of array `c`. We then increment a place-holding index on the array from which the object was copied and a place-holding index on the array `c`. This continues until one of the two arrays is exhausted. Then the remaining objects in the other array are copied to array `c`.

```
/********************  A R R A Y   M E R G E  ********************
This function assumes that the (sub)arrays a[afirst..alast] and
b[bfirst..blast] contain objects of a data type on which the binary
comparison operator "<=" is defined. It also assumes that the objects in
each array are arranged in increasing order.  It begins by making sure
that the (sub)array c[cfirst..clast] is large enough to hold all the
objects from the other two arrays;  if that is not the case, it returns
the value false).  If it determines that the array c is big enough, it
copies all the objects from array  a  and  b  into array c in such a way
that they form a single ordered list in  c .  Coded by W. Knight, using a
standard elementary merging algorithm. */

template <class otype>
bool mergeArrays (const otype a[], int afirst, int alast,
                  const otype b[], int bfirst, int blast,
                        otype c[], int cfirst, int clast)
{
    if (clast - cfirst + 1 < (alast - afirst + 1) + (blast - bfirst + 1))
        return false;
    while (afirst <= alast && bfirst <= blast)  // While a and b both
        if (a[afirst] <= b[bfirst])             // have objects that
            c[cfirst++] = a[afirst++];          // are not yet copied,
        else                                    // merge them into
            c[cfirst++] = b[bfirst++];          // the array c.

    // When the loop above terminates, at most one of the following
    // two loops will have its loop condition true.

    while (afirst <= alast)          // Copy the remaining objects from
        c[cfirst++] = a[afirst++];   // array a into array c.

    while (bfirst <= blast)          // Copy the remaining objects from
        c[cfirst++] = b[bfirst++];   // array b into array c.

    return true;
} // mergeArrays()
```

11.7. Merge Sort

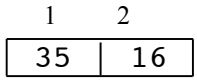
This sorting algorithm uses a divide-and-conquer strategy. The idea of the algorithm is to break the array to be sorted into two subarrays of approximately equal size, to sort these recursively, and then to merge the two sorted subarrays. Here is some C++ code to implement this idea. Note that the fourth parameter has a default value. When this function is called initially to sort some array, it should be called with just three arguments; no argument should be supplied initially for the fourth parameter.

```
template <class otype>
void mergeSort (otype a[], int first, int last, otype *aux = NULL)
{
    if (last <= first) // a[first..last] has size 1 or less. If size
        return;        // is 1 then the array is already sorted.
                       // This is the base case for the recursion.

    bool initialCall = !(aux);    // Set to true if and only if aux
                                  // is NULL.
    if (initialCall)
    {
        aux = new otype[last - first + 1]; //Allocate auxiliary space
        testAllocation (aux);
    }
    int mid = (first + last) / 2;

    mergeSort (a, first, mid, aux);     // Sort left subarray.
    mergeSort (a, mid+1, last, aux);    // Sort right subarray.

    // Merge the two sorted subarrays in "a" into the aux array.
    // Ignore the boolean value returned by the merge function.

    mergeArrays(a, first, mid, a, mid+1, last, aux, 0, last-first);

    // Copy the sorted objects from aux back to "a".

    for (int i = first, j = 0;  i <= last;  ++i, ++j)
        a[i] = aux[j];

    if (initialCall)    // then the array pointed to by aux was
        delete [] aux;  // allocated during this call. Deallocate it.
} // mergeSort()
```

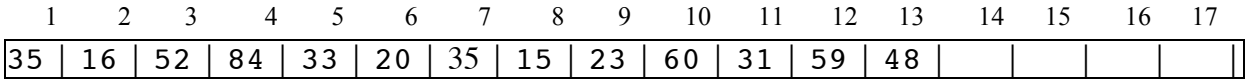
The amount of memory space used by this algorithm can be quite substantial. In addition to the memory space needed for the two addresses and two integers of the parameter list, the stack frame will also have to hold four local variables (a pointer and three integers). Moreover, the recursions produce a maximum of about $\log_2(n)$ stack frames. But even that's a piddling amount compared to the memory space required to be allocated for the auxiliary array `aux`, which has as many cells are there are objects to be sorted. All the objects get merged into the auxiliary array and then copied back to the original array. This is definitely *not* an in-place sorting algorithm.

Examples:

How would Merge Sort sort the following array of length 2 ?



How would Merge Sort sort the following array of length 13 ?



Like Quicksort, the Merge Sort algorithm is a "divide and conquer" algorithm. Unlike Quicksort, however, the "divide" operation cuts each array or subarray almost exactly in half. It can be proved (you will see the proof in C455) that the running time for this algorithm is $\Theta(n \log(n))$. That is, in both the best and worst cases (whatever those may be), this algorithm requires an amount of time approximately proportional to $n \log(n)$. This algorithm therefore comes with a guarantee: it can never take time proportional to n^2 . In this sense it is better than Quicksort. Experiments have shown, however, that the improved versions of Quicksort run faster than Merge Sort on almost all arrays, just as was true for Heap Sort.

Quicksort and Heap Sort run fast, but they are not stable sorting algorithms. Merge Sort also runs fast. Is it a stable sorting algorithm?

11.8. Distribution Sorting Algorithms

Problem: Suppose we want to sort a collection of integers. Suppose that we know for certain that there are no duplicates among the integers to be sorted. Finally, suppose we know that all the integers lie in some range 0 . . . MAX. Then we can sort them as follows. Create an array of 1+MAX boolean cells (cells that can hold `true` or `false`) and initialize all the cells to `false`. Then for each integer k in the collection to be sorted, change the `false` in cell k to `true`. Finally, make a pass over the array and for each cell that contains `true`, print (or store elsewhere) the subscript of that cell. This will print out (or store) in increasing order all the integers in the collection.

The running time for this will be Θ(MAX) because we must make two complete passes over the array of length MAX+1, and we must also handle each integer in the collection exactly once; since there are no duplicates in the collection, the number of integers in the collection is at most MAX+1.

Problem: Suppose we want to sort a collection of integers, where the collection may contain duplicate integers. Suppose we know that all the integers lie in some range 0 . . . MAX. Then we can sort them as follows. Create an array of 1+MAX integers and initialize all the cells to 0. The k-th cell in the array will be used to count the number of occurrences of k in the collection to be sorted. For each integer k in the collection, increment the count in cell k by 1. Finally, make a pass over the array and for each cell that contains a non-zero count c, print (or store elsewhere) c copies of the subscript of that cell. This will print out (or store) in increasing order all the integers in the collection.

One problem with the algorithms described above is that they may require an array of tremendous size. For example, if integers up to 32 bits in length can be members of the collection to be sorted, then we will require 2^{32} cells in the arrays. This is more than 4 billion cells. It is not likely that we will have so much memory space available to us in the near future. In such a case, we can break the integers down into pieces and sort on the various pieces. This is the notion of **Bucket Sort** (also called **Radix Sort**).

Suppose, for concreteness, we are sorting a collection of unsigned integers (non-negative integers) that are stored in 32-bit "words" of memory. Then each integer can be broken down into 4 parts, say, where each part is 8 bits long. The integers can then be sorted on each of the 4 parts separately, starting with the least significant part and moving to the most significant part. In this case, the 8 bits of each part can store $2^8 = 256$ bit patterns, each of which can be thought of as an unsigned integer. So we create 256 empty queues over which we will distribute the integers to be sorted.

On the first pass, the integers are enqueued on the basis of the least significant 8 bits in each. When this pass is complete, the queues are concatenated with each other to form one long "master list" of integers that are in sorted order on the basis of their least significant 8 bits.

On the second pass, we dequeue the integers from the master list and distribute them over the queues on the basis of the next most significant 8 bits. When this pass is complete, the queues are concatenated with each other to form a "master list" of integers that are in sorted order on the basis of their least significant 16 bits.

The third and fourth passes are similar. The fourth pass sorts on the basis of the most significant 8 bits of the integers, and when it is complete we will have a single list of all the integers arranged in increasing order.

There is nothing special from the logical point of view about 8 bits. We could choose to sort on 8 parts, each of size 4 bits. This would require 8 passes but only $2^4 = 16$ queues. Alternatively, we could have 2 parts, each of size 16 bits, which would require 2 passes but $2^{16} = 65,536$ queues. Or we could break each integer into 3 parts, two of size 11 bits and one of size 10 bits, which would require $2^{11} = 2048$ queues.

What is the running time? Suppose that we plan to break each integer into p parts, with approximately $b = 32/p$ bits in each part. Then we will need to make p passes over the n integers to be sorted, and each pass involves the following:

- initialize the 2^b queues to empty;
- throw each of the n integers into the appropriate queue based on the bits in the part we are sorting on;
- concatenate the 2^b queues into a single list.

Assuming that 2^b is smaller than n (it is usually much smaller in cases where you would want to use this sort), the running time on each pass will be dominated by the time required to distribute the n integers over the queues. Thus the total running time should be roughly proportional to pn. Since p can be regarded as a constant (no matter what n is we'll break each integer into p parts), we have what appears to be a linear time sorting algorithm: the time required is proportional to n; that is, the running time is $\Theta(n)$. This is better than any of the other methods we have seen so far. Note however that this generally works well only with integer keys or with strings of approximately equal length.

Example: Suppose we want to sort the following integers using the Bucket Sort algorithm, but with each decimal digit treated as a separate part of the integers.

258	736	908	521	745	96	539	806	738	265	6	540	761	939	725

11.9. External Sorting

Suppose $x_1, x_2, x_3, \ldots, x_n$ are objects in a file to be sorted, and suppose **n** is so large that only a fraction of the objects can be held in RAM at any one time. Let **m** denote the maximum number of objects from the data set that we can have in RAM at any one time. In this case, our sorting strategy will be to sort the file piecemeal, internally, to produce ordered subfiles called ***runs*** that we'll store externally and then merge.

An important fact to keep in mind is that reading an object from a file or writing one to a file takes anywhere from 10 to 1000 times as long as moving an object from one location to another in RAM. Thus, when judging competing strategies, the number of *reads* and *writes* is likely to be the main consideration, not what happens to the objects internally.

How can we produce the initial runs? The naive way is to read in m objects, sort them, write them out to a file, call it "Run1". Repeat with the next m objects, writing them out to Run2. Etc. There will be $\cup n/m\_$ runs produced. Each object will be read once and written once, so there will be n read/writes. When finished creating the runs, start merging the run files, as many as you can at a time.

Example: Suppose $n = 680{,}000$ and $m = 30{,}000$. The naive method gives

Suppose further that we can merge 4 runs at a time (an artificially small number). How many rounds of merges would be required to get all the objects into one sorted run?

Each round of merges would require that every object be read and written once, so a total of 3n read/writes The total number of read/writes for the entire sorting operation would therefore be 4n .

The example above assumes tacitly that the runs are placed in separate files on a hard disk so that there is "random access" to each separate file whenever we want it. The situation is worse when all the data about the objects is on tape and all the runs must be placed on tape. Suppose, for example, that we have 5 tape drives and all the data is originally on one tape. We place that tape on one of the drives and then start reading the data off it. Each time we create a run we write that run out to a tape on one of the other four drives, distributing the runs as evenly as possible over those 4 tapes. Then in the example above, 6 runs will go onto each of 3 tapes, and 5 runs will go onto the fourth tape. Then we could remove our original tape (keep it safely stored in case anything goes wrong with the sorting process), put on a new blank tape (call it Tape 0), rewind the 4 tapes with the 23 runs spread over them, and start reading from those 4 tapes, merging the runs into 6 longer runs that could be written onto Tape 0. Then we could rewind all the tapes, distribute the 6 runs from Tape 0 over the other four tapes, rewind those tapes, read the runs off of them and merge the 6 runs into 2 runs, which would be written onto Tape 0. We could then rewind all the tapes, distribute the 2 runs from Tape 0 onto two of the other tapes, rewind all the tapes, and merge the two runs into a single sorted data set on Tape 0. How many times would each object be read and written?

The situation might be improved by writing the original 23 runs onto just two of the other 4 tapes, call them A1 and A2, and then merging two runs at a time onto the other two tapes, call them B1 and B2. Then we could rewind all the tapes and merge the 12 runs from the B tapes onto the A tapes, creating 6 runs distributed over the A tapes. Then we could rewind again and merge runs to get 3 runs on the B tapes. Etc. How many times would each object be read and written?

The running time for an external sort is heavily dependent on the number of times each object must be read and written. For that reason, anything we can do to cut down that number will improve the running time markedly. One way to do that is to make the initial runs longer.

11.10. Replacement Selection Algorithm (for producing initial runs)

Again, assume we have **n** objects in a file to be sorted, and we can hold only **m** in RAM. The following algorithm for producing initial runs (sorted subfiles) is called the Replacement Selection Algorithm.

(1) Read the first m objects from the file into an array.

(2) Heapify the array, making it into a min heap.

(3) Create a new file for an initial run.

(4) Repeat the next two steps until the heap size is reduced to 0 :

(a) Remove the smallest object from the array and write it out to the run (the file opened in step (3)).

(b) Read in the next object from the original file. If that object is larger than or equal to the object that went out, it can be part of the same run as the object that just went out, so insert it at the top of the heap and let it sift down to its correct position. If, instead, the new object is smaller than the object that just went out, it will have to go into a later run, so put the object that's in the last heap position up to the top and replace it with the new object. Decrement by 1 the variable that holds the heap size, and let the top object sift down in the smaller heap.

(5) Return to step (2), then step (3) (create a file for the next initial run), etc.

Stop when all objects from the original file have been read and written out to some initial run.

Example: Suppose the original file contains 16 objects in the following order:

38 62 25 51 80 44 39 32 65 20 88 60 14 48 91 75

Suppose the internal sorting array has just 6 places (m = 6).

Written Exercises

11-2. Rewrite the code for Selection Sort (page 3-5) so that instead of using a loop it uses recursion. (The recursion in this case will be tail recursion.)

11-3. Below is an array in random order. Suppose we apply the Heap Sort algorithm to this array. Show what it will look like at the end of Phase I (the heapification phase). Then show what it will look like after the largest value has been moved to the right end. Finally, show what it will look like after the next-largest value has been moved to the right end.

1	2	3	4	5	6	7	8	9	10	11	12	13
42	27	55	38	61	19	76	84	29	52	90	36	67

11-4. Below is an array in random order. Suppose we apply the Heap Sort algorithm to this array. Show what it will look like at the end of Phase I (the heapification phase). Then show what it will look like after the largest value has been moved to the right end. Finally, show what it will look like after the next-largest value has been moved to the right end.

1	2	3	4	5	6	7	8	9	10	11	12	13
24	72	55	83	16	91	67	48	92	25	9	63	76

11-6. Below is an array in random order. Suppose we apply the Heap Sort algorithm to this array. Show what it will look like at the end of Phase I (the heapification phase). Then show what it will look like after the largest value has been moved to the right end. Finally, show what it will look like after the next-largest value has been moved to the right end.

1	2	3	4	5	6	7	8	9	10	11	12	13
76	28	45	17	84	9	33	52	8	75	91	37	24

11-7. Suppose we are given the following array to be sorted by one of our Insertion Sorts.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
a		45	71	62	45	80	53	28	59	16	77	45	60	45

After the outer loop body has been executed 9 times in any of the three Insertion Sorts, the array will have been changed to look like this:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
a		16	28	45	45	53	59	62	71	77	80	45	60	45

sorted unsorted

At this stage, the loop variable `p` will be incremented to 11 , and it will be time to insert the 45 in cell `a[11]` into its correct position in the sorted portion of the array.

(a) If the Simple Linear Insertion Sort (page 11-3) is used, how many swaps will be made to accomplish this? Since every swap involves three object moves, how many object moves will be made? (In this example, the objects are integers.)

(b) If the Improved Linear Insertion Sort (page 11-5) is used to insert the 45 in cell `a[11]` into correct position, how many ints will be moved during the process? (Be sure to count the move of 45 into `a[0]` and the later move of 45 to its correct position.)

(c) If the Binary Insertion Sort (page 11-5) is used, which cells will be examined during the search of the subarray `a[1..10]`? (Use the algorithm on page 3-9.) The 45 that started in `a[11]` will end up in which cell? Is this different from what happens in the Linear Insertion Sorts?

11-8. Suppose we are given the following array to be sorted by one of our Insertion Sorts.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
a		61	19	35	19	70	45	19	4	19	83	54	19	22

After the outer loop body has been executed 10 times in any of the three Insertion Sorts, the array will have been changed to look like this:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
a		4	19	19	19	19	35	45	54	61	70	83	19	22

sorted unsorted

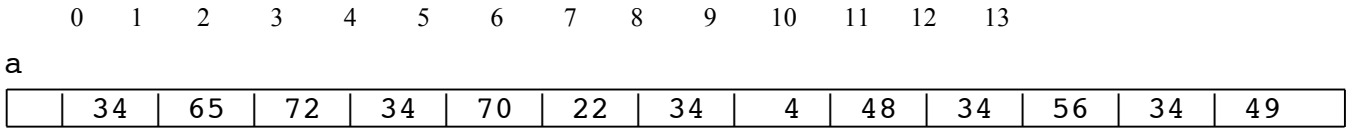
At this stage, the loop variable `p` will be incremented to 12 , and it will be time to insert the 19 in cell `a[12]` into its correct position in the sorted portion of the array.

(a) If the Simple Linear Insertion Sort (page 11-3) is used, how many swaps will be made to accomplish this? Since every swap involves three object moves, how many object moves will be made? (In this example, the objects are integers.)

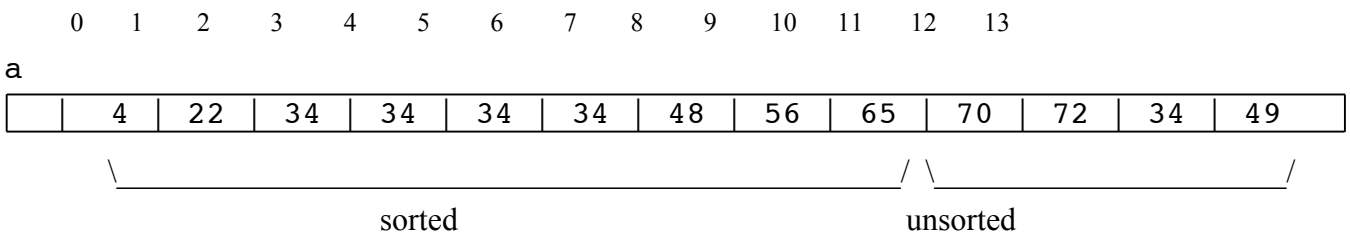
(b) If the Improved Linear Insertion Sort (page 11-5) is used to insert the 19 in cell `a[12]` into its correct position, how many ints will be moved during the process? (Be sure to count the move of 19 into `a[0]` and the later move of 19 to its correct position.)

(c) If the Binary Insertion Sort (page 11-5) is used, which cells will be examined during the search of the subarray `a[1..11]`? (Use the algorithm on page 3-9.) The 19 that started in `a[12]` will end up in which cell? Is this different from what happens in the Linear Insertion Sorts?

11-10. Suppose we are given the following array to be sorted by one of our Insertion Sorts.



After the outer loop body has been executed 10 times in any of the three Insertion Sorts, the array will have been changed to look like this:



At this stage, the loop variable `p` will be incremented to 12 , and it will be time to move the 34 from cell `a[12]` into its correct position in the sorted portion of the array.

(a) If the Simple Linear Insertion Sort (page 11-3) is used, how many swaps will be made to accomplish this? Since every swap involves three object moves, how many object moves will be made? (In this example, the objects are integers.)

(b) If the Improved Linear Insertion Sort (page 11-5) is used to insert the 34 in cell `a[12]` into its correct position, how many ints will be moved during the process? (Be sure to count the move of 34 into `a[0]` and the later move of 34 to its correct position.)

(c) If the Binary Insertion Sort (page 11-5) is used, which cells will be examined during the search of the subarray `a[1..11]`? (Use the algorithm on page 3-9.) The 34 that started in `a[12]` will end up in which cell? Is this different from what happens in the Linear Insertion Sorts?

11-12. Rewrite the code for Simple Linear Insertion Sort (page 11-3) so that instead of using a loop it uses recursion. (The recursion in this case will *not* be tail recursion.)

11-13. Show how to eliminate the local variable `p` from the code for the `brainlessBubbleSort` function given on page 11-6. [Hint: the variable `n` is passed by value, so it can be changed.] Warning: do not eliminate the variable `p` just by changing its name. The idea is to reduce (very slightly) the amount of space required by the function by eliminating an unnecessary local variable.

11-16. Rewrite the code for Brainless Bubble Sort (page 11-6) so that instead of using a loop it uses recursion. (The recursion in this case will be tail recursion.)

11-22. Write a template function that accepts three objects and returns their median. Assume that all four of the operators > , >= , < , <= are defined on pairs of the objects.

11-23. (a) Below is an array in random order. Suppose we apply the naive `quicksort` function (the one given in the class notes on page 11-8) to this array. What the array will look like after the first pass over the array; that is, what will it look like just before the two recursive calls? Show your work.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
37	44	20	97	33	78	45	71	69	90	25	76	81	71	63	39	94	50	47	19	58	12	14	88	52	26	56

(b) Starting from the same array as the one you *started* with in part (a) (i.e., the array shown just above), make one pass over the array using the improved version of `quicksort` that implements the median-of-three strategy as described in class. Compare the "split" you obtained this way with the split you obtained in part (a). [Warning: be sure to calculate correctly the subscript of the "mid" cell of the array.]

11-24. (a) Below is an array in random order. Suppose we apply the naive `quicksort` function (the one given in the class notes on page 11-8) to this array. What the array will look like after the first pass over the array; that is, what will it look like just before the two recursive calls? Show your work.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
62	60	19	31	52	34	88	45	27	43	96	22	50	71	36	12

(b) Starting from the same array as the one you *started* with in part (a) (i.e., the array shown just above), make one pass over the array using the improved version of `quicksort` that implements the median-of-three strategy as described in class. Compare the "split" you obtained this way with the split you obtained in part (a).

11-26. (a) Below is an array in random order. Suppose we apply the naive `quicksort` function (the one given in the class notes on page 11-8) to this array. What the array will look like after the first pass over the array; that is, what will it look like just before the two recursive calls? Show your work.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
38	40	81	69	48	66	12	55	73	57	4	78	50	29	64	88

(b) Starting from the same array as the one you *started* with in part (a) (i.e., the array shown just above), make one pass over the array using the improved version of `quicksort` that implements the median-of-three strategy as described in class. Compare the "split" you obtained this way with the split you obtained in part (a).

11-27. What would you conjecture is the running time for Quicksort on an array in which all the objects had the same value (e.g., an array of n cells, each containing the integer 43)? The way to answer this is to investigate whether the "splits" will be good or bad.

11-29. Suppose we want to be able to sort very long arrays as quickly as possible. Naturally we will want to use Quicksort with the median-of-three strategy. However, as was described in class, when the recursive calls in Quicksort are made on arrays of size 15 or smaller, the code in Quicksort is so complicated that other simpler sorting techniques are actually faster. Describe carefully the way in which we can combine Quicksort with Linear Insertion Sort to produce an especially fast sorting algorithm.

11-33. Show in detail how the Merge Sort Algorithm sorts the following array.

1	2	3	4	5	6	7	8
71	35	52	84	63	37	90	25

11-35. Show how Bucket Sort would alphabetize the following four-letter words, all of which are made up of lower case letters in the range from `a` to `k` (the first 11 letters of the alphabet). Make 4 passes, each time sorting on a different position in the word. Use 11 queues.

```
iced  aide  back  heed  bide  deck  edge  jack  fade  hack
bead  head  gage  aged  kick  cake  each  bike  cafe  deed
```

11-36. Show how Bucket Sort would alphabetize the following three-letter words, all of which are made up of lower case letters in the range from `a` to `k` (the first 11 letters of the alphabet).

```
hid  bib  dad  ice  cad  die  ebb  kid  gab  bad  eke  fig
dab  aid  fed  egg  bid  ace  fad  bee  had  gag  age  bag
```

11-38. Show how Bucket Sort would alphabetize the following three-letter words, all of which are made up of lower case letters in the range from `a` to `l` (the first 12 letters of the alphabet).

```
eel  big  add  led  gel  fee  lie  elk  dig  lab  ilk  fie
did  ale  lad  beg  cab  gad  ail  fib  ill  bah  gal  lid
```

11-39. Suppose a file contains the following 25 integers to be sorted.

```
12 63 39 94 50 47 19 58 72 35 97 20 33 26 52 88 14 37 45 71 69 90 25 76 81
```

Suppose the program that must sort these numbers can spare only 5 array cells in which to form runs for an external sort. The cells will be numbered from 1 to 5 (ignore the question of whether there has to be a cell numbered 0). The naive method of producing runs reads 5 integers at a time from the file into the array, sorts them, and writes out runs of size 5; this would produce 5 initial runs:

```
12 39 50 63 94
19 35 49 58 72
20 26 33 52 97
14 37 45 71 88
25 69 76 81 90
```

Show how the initial runs would be formed if, instead of this naive method, we use the Replacement Selection Algorithm for generating runs. Show *in complete detail* how this is done up to the point at which two objects have been written out to the first run. After that point, don't show all the binary heap operations; just show the array with its 5 cells and use "marking out" to keep track at all times of what is in the priority queue. You can do the delete-min operations in your head.

11-40. Suppose a file contains the following 25 integers to be sorted.

```
52 88 14 37 45 71 69 90 25 76 81 12 63 39 94 50 47 19 58 72 35 97 20 33 26
```

Suppose the program that must sort these numbers can spare only 5 array cells in which to form runs for an external sort. The cells will be numbered from 1 to 5 (ignore the question of whether there has to be a cell numbered 0).

(a) If the naive method of producing initial runs is used, how many runs will be formed and what will they look like?

(b) Show how the initial runs would be formed if, instead of the naive method, we use the Replacement Selection Algorithm for generating runs. Show *in complete detail* how this is done up to the point at which two objects have been written out to the first run. After that point, don't show all the binary heap operations. Just show the array with its 5 cells and keep track by "marking out".

11-42. Suppose a file contains the following 25 integers to be sorted.

```
58 37 61 12 50 53 81 42 28 65 13 80 67 74 48 19 86 63 55 29 31 10 75 24 18
```

Suppose the program that must sort these numbers can spare only 5 array cells in which to form runs for an external sort. The cells will be numbered from 1 to 5 (ignore the question of whether there has to be a cell numbered 0).

(a) If the naive method of producing initial runs is used, how many runs will be formed and what will they look like?

(b) Show how the initial runs would be formed if, instead of the naive method, we use the Replacement Selection Algorithm for generating runs. Show *in complete detail* how this is done up to the point at which two objects have been written out to the first run. After that point, don't show all the binary heap operations. Just show the array with its 5 cells and keep track by "marking out".

Solutions for Odd-Numbered Exercises

11-3. Phase I of Heap Sort is to form the array objects into a *max*-heap. The heapification starts at cell 6 and moves to the left, allowing each object to sift as far down the tree as it can.

```
 1  2  3  4  5  6  7  8  9 10 11 12 13
|42|27|55|38|61|19|76|84|29|52|90|36|67|

               67 - - - - - - - - - - 19

            90 - - - - - - - -  61

         84 - - - -  38

      76 - - - -  55

   90 - - - 27                          (Note that 27 must fall 2 levels,
            61 - - - - - - - -  27       not just one.)

90 42                                   (The 33 must fall 2 levels, but need not
   84 -- 42                              fall to its lowest possible level.)
```

```
 1  2  3  4  5  6  7  8  9 10 11 12 13
|90|84|76|42|61|67|55|38|29|52|27|36|19|
```

This is what is produced by Phase I.

The first step in Phase II is to swap the first and last items in the heap, decrease the heap size to 12 , and let the item at the root sift down. After we swap the 90 with the 19 , the 19 falls to cell 2 , then to cell 5 , then to cell 10 . The result of these operations is

```
 1  2  3  4  5  6  7  8  9 10 11 12 13
|84|61|76|42|52|67|55|38|29|19|27|36|90|
```

Now we swap the 84 with the 36 , reduce the heap size to 11 , and let the 36 sift down through cell 3 to cell 6 . The result is

```
 1  2  3  4  5  6  7  8  9 10 11 12 13
|76|61|67|42|52|36|55|38|29|19|27|84|90|
```

The Heap Sort algorithm would continue in this way until the heap size was reduced to 1. At that point, the array would be sorted.

11-7.

```
                0  1  2  3  4  5  6  7  8  9 10 11 12 13
                   a
|  | 16 | 28 | 45 | 45 | 53 | 59 | 62 | 71 | 77 | 80 | 45 | 60 | 45 |
   \_____________________________________________________/ \_________/
                        sorted                              unsorted
```

(a) If we use Simple Linear Insertion Sort, then the 45 in cell 11 will be swapped with 80, then with 77, then with 71, then with 62, then with 59 , then with 53 . It will NOT be swapped with the 45 in cell 4 . Thus 6 swaps will be made, which means that 18 integer moves will be made.

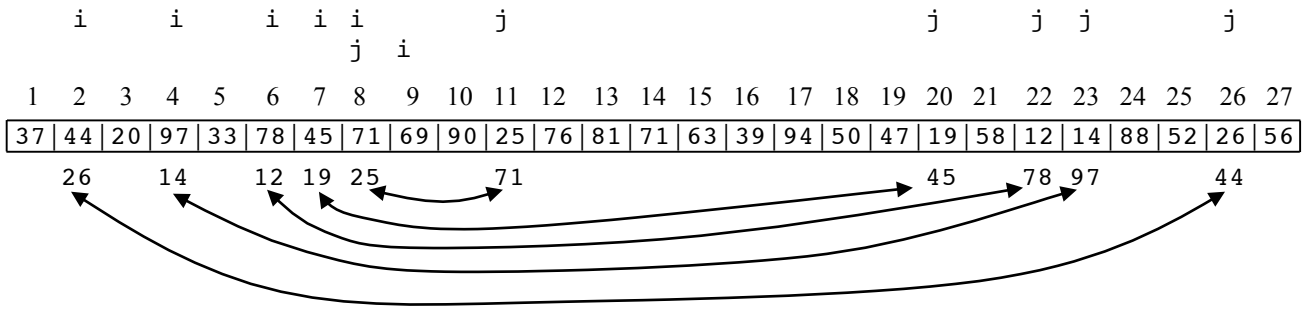
(b) First 45 will be moved from cell 11 into cell 0 . Then on the basis of comparisons, 80 will be shifted right by one cell, then 77 , then 71, then 62, then 59, then 53 . Then 45 will be moved from cell 0 into cell 5 . Altogether, 8 integer moves will be made.

(c) When `binarySearch` is called on the subarray `a[1..10]` with target 45, it first examines cell 5, then cell 2, then cell 3, where it finds another 45. The `binarySearch` function will return the subscript 3. Thus the 45 that started in `a[11]` will end up in cell 3. This is different from the Linear Insertion Sorts, which move that 45 from cell 11 to cell 5.

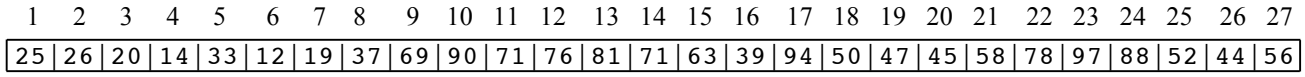
11-13.

```
template <class otype>
void brainlessBubbleSort (otype a[], int n)  // sorts a[1...n]
{
    while (n >= 2)
    {
        for (int i = 1; i <= n - 1; ++i)
            if ( a[i] > a[i+1] )
                swap (a[i], a[i+1]);
        --n;
    }
} // brainlessBubbleSort()
```

11-23. (a) We use the 37 in cell 1 as the partitioning object.

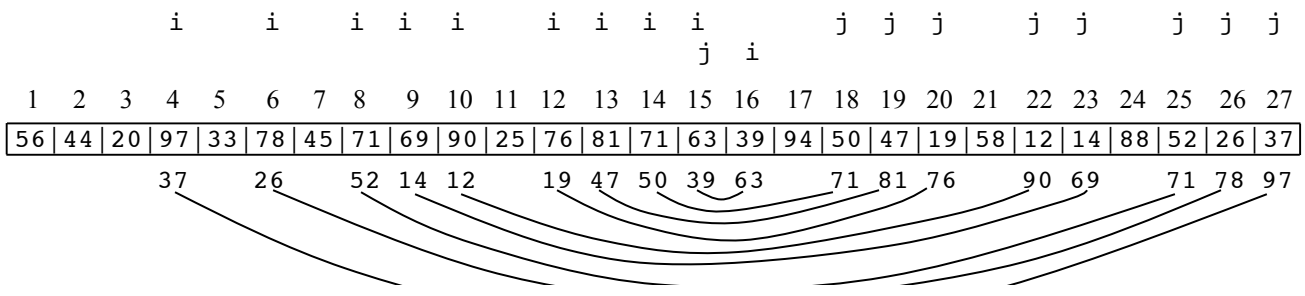


The index `j` stops for the last time at cell 8, so the partitioning object 37 is swapped with 25.



Now recursive calls will be made on the subarrays `a[1..7]` and `a[9..27]`. This is not a particularly good split.

(b) This time we begin by examining the objects in cells 1, 27, and 14 (= (1 + 27) / 2). They are 37, 71, and 56. The median of those three numbers is 56, so we swap it into first position (cell 1) so it can serve as the partitioning object.



The index `j` stops for the last time at cell 15, so the partitioning object 56 is swapped with 39 .

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
39	44	20	37	33	26	45	52	14	12	25	19	47	50	56	63	94	71	81	76	58	90	69	88	71	78	97

This time recursive calls will be made on the subarrays `a[1..14]` and `a[16..27]` . This is a much better split than the split obtained by using the naive version without median-of-three.

11-27. The subscript `i` would stop at each cell as it moved to the right, and the subscipt `j` would stop at each cell as it moved to the left. They would meet at the middle of the array, so we would make about n/2 swaps, and the partitioning object would end up at about the center of the array. This means we always get a "good split". The same will be true in every recursive call, so the running time will be $\Theta(n \log(n))$.

11-29. The algorithm should be structured this way:

```
template <class otype>          // sorts a[first...last]
void quicksort (otype a[], int first, int last)
{
    incompleteQuicksort (a, first, last);

    simpleLinearInsertionSort (a, first, last);
    // This would be a slightly more general version of the code
    // found on page 11-3.
} // quicksort()


template <class otype>          // "nearly" sorts a[first...last]
void incompleteQuicksort (otype a[], int first, int last)
{
    if (last - first + 1 <= 15)   // or simply  (last - first < 15)
        return;                   // This prevents the use of the
                                  // quicksort "machinery" on small
    else                          // on small arrays.
    {
        "move the median-of-three into the first cell"

        The rest of the code is the same as in the "naive" quicksort on page 11-8,
        except that the recursive calls are to incompleteQuicksort.
    }
} // incompleteQuicksort ()
```

When the initial call to `incompleteQuicksort` is finished, the array `a[first...last]` will not be completely sorted. Instead, it will consist of a number of segments of size 15 or less, each of which is unsorted but in correct position relative to all the other segments. The one call to a Simple Linear Insertion Sort will complete the sorting of the entire array. Since no object is farther than 15 cells from where it belongs, Simple Linear Insertion Sort will very quickly finish off the sorting of the entire array.

11-28

11-33. The following explanation is an adaptation of a solution to a similar problem.
The solution was given by a C243 student, Brendan Bellina, in the fall semester of 1998.

Split the array

1	2	3	4	5	6	7	8
71	35	52	84	63	37	90	25

into two pieces:

71	35	52	84

63	37	90	25

Sort the first subarray recursively as shown below. Then sort the second, as shown below.

Split the subarray into two pieces.

71	35

52	84

Split the subarray into two pieces.

63	37

90	25

Sort the first as below: Sort the second as below.

Sort the first (below). Sort the second:

Split into two. Split into two.

Split into two. Split into two:

71

35

52

84

63

37

90

25

Sort first (base case). Sort first (base case).

Sort the first (base). Sort the 1st (base).

Sort second (base case). Sort second (base case).

Sort the 2nd (base). Sort the 2nd (base).

Merge them to produce

35	71

Merge them to produce

52	84

Merge to produce

37	63

Merge to produce

25	90

Now merge the two subarrays into a single array.

35	52	71	84

Now merge these two into one subarray.

25	37	63	90

Now merge these two subarrays into a single (auxiliary) array.

25	35	37	52	63	71	84	90

Copy these back into the original array.

11-35.

```
iced  aide  back  heed  bide  deck  edge  jack  fade
hack
bead  head  gage  aged  kick  cake  each  bike  cafe
deed
```

First we put each object in a "bucket" based on its last (4th) character.

```
a ->
b ->
c ->
d -> iced -> heed -> bead -> head -> aged -> deed
e -> aide -> bide -> edge -> fade -> gage -> cake -> bike -> cafe
f ->
g ->
h -> each
i ->
j ->
k -> back -> deck -> jack -> hack -> kick
```

Concatenating these queues in order gives the new list

```
iced, heed, bead, head, aged, deed, aide, bide, edge, fade
gage, cake, bike, cafe, each, back, deck, jack, hack, kick
```

Now we put each object in a "bucket" based on its third character.

```
a -> bead -> head
b ->
c -> each -> back -> deck -> jack -> hack -> kick
d -> aide -> bide -> fade
e -> iced -> heed -> aged -> deed
f -> cafe
g -> edge -> gage
h ->
i ->
j ->
k -> cake -> bike
```

Concatenating these queues in order gives the new list

```
bead  head  each  back  deck  jack  hack  kick  aide  bide
fade  iced  heed  aged  deed  cafe  edge  gage  cake  bike
```

Now we put each object in a "bucket" based on its second character.

```
a -> each -> back -> jack -> hack -> fade -> cafe -> gage -> cake
b ->
c -> iced
d -> edge
e -> bead -> head -> deck -> heed -> deed
f ->
g -> aged
h ->
i -> kick -> aide -> bide -> bike
j ->
k ->
```

Concatenating these queues in order gives the new list

```
each  back  jack  hack  fade  cafe  gage  cake  iced  edge
bead  head  deck  heed  deed  aged  kick  aide  bide  bike
```

Now we put each object in a "bucket" based on its first character.

```
a -> aged -> aide
b -> back -> bead -> bide -> bike
c -> cafe -> cake
d -> deck -> deed
e -> each -> edge
f -> fade
g -> gage
h -> hack -> head -> heed
i -> iced
j -> jack
k -> kick
```

Concatenating these queues in order gives the final sorted list:

```
aged  aide  back  bead  bide  bike  cafe  cake  deck  deed
each  edge  fade  gage  hack  head  heed  iced  jack  kick
```

11-39. Read 5 integers into the array and form them into a binary min-heap:

```
  1   2   3   4   5
| 12| 63| 39| 94| 50|   <--  Here are the integers read from the file.  Heapify it to a min-heap.

    50-------63

  1   2   3   4   5
| 12| 50| 39| 94| 63|   <--  Now we have a min-heap.
```

The smallest object is 12 . Write out 12 to the 1st run. Read in the integer 47 from the file to be sorted. Since 47 is larger than 12, it can join the 1st run. Insert 47 into the heap by putting it in cell 1 and letting it sift down.

```
  1   2   3   4   5
| 47| 50| 39| 94| 63|

39----47

  1   2   3   4   5
| 39| 50| 47| 94| 63|   <--  Now we have a new min-heap.
```

Write out the smallest integer 39 to the 1st run. Read in the integer 19 from the file to be sorted. Since 19 is smaller than 39, it cannot join the 1st run. Move 63 from cell 5 to cell 1, and then put 19 into cell 5 . **Now the heap consists of only 4 cells; cell 5 is no longer part of the heap. Let 63 sift down in this min-heap of 4 cells.**

```
  1   2   3   4       5
| 63| 50| 47| 94|    19|

47----63

  1   2   3   4       5
| 47| 50| 63| 94|    19|   <--  Now we have a new min-heap;  it has  4  cells.
```

From this point on, I will not show the sift downs. I will just pretend that the array contains a min-heap, and I'll visually select the smallest from the "heap" at each step.

The array now contains the following objects.

```
| 47| 50| 63| 94|    19|
```

Write out the smallest integer 47 to the 1st run. Read in the integer 58 from the file to be sorted. Since 58 is larger than 47, it can join the 1st run. Put 58 into the heap.

Write out the smallest integer 50 to the 1st run. Read in the integer 72 from the file to be sorted. Since 72 is larger than 58, it can join the 1st run. Put 72 into the heap.

Write out the smallest integer 58 to the 1st run. Read in the integer 35 from the file to be sorted. Since 35 is smaller than 58, it cannot join the 1st run. Put 35 in cell number 4.

Now the heap consists of only 3 cells; cells 4 and 5 are no longer part of the heap.

1	2	3	4	5
63	72	94	35	19

Write out the smallest integer 63 to the 1st run. Read in the integer 97 from the file to be sorted. Since 97 is larger than 63, it can join the 1st run. Put 97 into the heap.

Write out the integer 72 to the 1st run. Read in the integer 20 from the file to be sorted. Since 20 is smaller than 72, it cannot join the 1st run. Put 20 in cell number 3 . **Now the heap consists of only 2 cells.**

1	2	3	4	5
94	97	20	35	19

Write out the integer 94 to the 1st run. Read in the integer 33 from the file to be sorted. Since 33 is smaller than 94, it cannot join the 1st run. Put 33 in cell number 2 . **Now the heap consists of only 1 cell.**

1	2	3	4	5
97	33	20	35	19

Write out the integer 97 to the 1st run. Read in the integer 26 from the file to be sorted. Since 26 is smaller than 97, it cannot join the 1st run. Put 26 in cell number 1 and **declare the binary heap to be extinct**.

The first run has ended. It contains **12, 39, 47, 50, 58, 63, 72, 94, 97.**

The array now looks like this:

1	2	3	4	5
26	33	20	35	19

We now heapify this into a new min-heap and start over, this time producing run 2.

Write out 19 and read in 52, which can join the 2nd run.

Write out 20 and read in 88, which can join the 2nd run.

Write out 26 and read in 14, which cannot join the 2nd run, so put it in cell 5 . Heap has 4 cells.

Write out 33 and read in 37, which can join the 2nd run.

Write out 35 and read in 45, which can join the 2nd run.

Write out 37 and read in 71, which can join the 2nd run.

Write out 45 and read in 69, which can join the 2nd run.

Write out 52 and read in 90, which can join the 2nd run.

Write out 69 and read in 25, which cannot join the 2nd run, so put it in cell 4 . Heap has 3 cells.

Write out 71 and read in 76, which can join the 2nd run.

Write out 76 and read in 81, which can join the 2nd run.

Write out 81. Nothing more can be read in because the input file is exhausted.

Write out 88.

Write out 90. This exhausts the binary min-heap.

The **second run** contains **19, 20, 25, 26, 33, 37, 45, 52, 69, 71, 76, 81, 88, 90**.

Finally, the **third run** will contain **14, 25** .

Thus instead of 5 initial runs (which would be produced by the naive method), we produce 3 initial runs.

12. Finite Graphs and Their Applications

In this chapter we are going to study various algorithms for solving certain classical problems in graph theory. Before we can do that we need to have a precise definition of the concept of a finite graph, and we need a good way of implementing graphs within computer programs.

Intuitively, graphs are data structures of type container representing collections of objects together with a binary relationship between these objects. The objects stored in a graph are called **vertices**. A binary relationship is usually defined as a connection between couples of objects from the set. Thus, any two objects in the graph may or may not be connected by this relationship. Such connections are called **edges** in the graph.

Remark: The word ***vertex*** is the singular of the plural form ***vertices***.

Finite graphs can be used to represent such things as

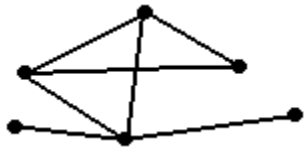
- power line connections in a community;
- road systems in a state;
- flows of information (data);
- dependencies among functions in programs;
- finite automata (e.g. lexical analyzers);
- circuits on a microchip;
- course prerequisites in an academic setting;

and so on. The variety of applications is limitless.

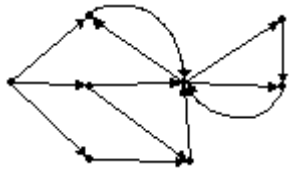
If the connections between vertices can go both ways, then we call the graph *undirected*. If the connections are not symmetrical, such that each edge has a source and a destination, then the graph is called *directed* or a *digraph*.

The easiest way to handle graphs is to use a graphical representation of them where the vertices are drawn as circles or dots and the edges are drawn as line segments between them. We can see below the representation of a directed and undirected graph.

Here is a typical **undirected finite graph**. It has 6 **vertices** (represented by dots) and 7 **edges**.



Here is a typical **directed finite graph (digraph)**. It has 8 vertices and 13 edges.



Let's give a formal definition of these notions.

Definition A ***finite graph***, call it G, consists of a finite set, call it V, of ***vertices***, and a finite set, call it E (the ***edge set*** of G), of pairs of vertices from V. We call each pair an ***edge*** of the graph G.

If each edge in E is an *ordered* pair (i.e. there is definite first vertex of the pair, and a definite second vertex), then we say that the graph is a ***directed graph***, or ***digraph***. If all the edges in E are unordered pairs (i.e. sets of size 2), then we say that G is ***non-directed***, or ***undirected***.

If each edge in G is assigned a numerical value, then we call G a ***weighted graph***, and the numbers on the edges are the ***weights*** in G.

The number of vertices in V is denoted by $|V|$, and the number of edges in E is denoted by $|E|$.

If v_1 and v_2 are vertices in V, then we say that there is ***an edge from*** v_1 ***to*** v_2 iff the ordered pair (v_1, v_2) is in the edge set E (in the case of a digraph) or the unordered pair $\{v_1, v_2\}$ is in the edge set E (in the case of an undirected graph); in an undirected graph we also speak of an edge being ***between*** v_1 and v_2. A vertex v_2 is said to be ***adjacent to*** a vertex v_1 iff there is an edge from v_1 to v_2. We will assume that for any two distinct vertices v_1 and v_2 in a graph there can be at most one edge from v_1 to v_2. We will also assume that a finite graph *never* contains an edge from a vertex to the same vertex (such an edge is sometimes called a "loop").

Remark: Let us denote by n the number of vertices in the graph (or $|V|$), and by m the number of edges (or $|E|$). The *minimum* possible value for the number m in a finite graph is zero. A graph with no edges whatsoever is not a very interesting graph, since it is "totally disconnected". The *maximum* possible value of m in a *digraph* can be computed by noting that for *each* of the vertices, there can be at most $n - 1$ edges *from* that vertex *to* the other vertices. Altogether then there can be at most $n(n - 1)$ edges in a digraph and $n(n-1)/2$ in an undirected graph because we have counted each edge twice. A finite graph that contains all possible edges between its vertices is called ***complete***. What is the value of $|E|$ in a *complete, undirected* graph?

We see that for finite graphs, $|E| = O(|V|^2)$.

Example 1. Let G be the *digraph* whose vertex set is $V = \{ v_1 , v_2 , v_3 , v_4 \}$ and whose edge set is $E = \{ (v_1 , v_2), (v_1 , v_3), (v_3 , v_1), (v_3 , v_2), (v_2 , v_4), (v_4 , v_3) \}$. Let's draw this graph.

In this graph, v_4 is adjacent to v_2 ,
but v_2 is *not* adjacent to v_4.

Example 2: Let G be the *undirected* graph with vertex set $V = \{ v_1 , v_2 , v_3 , v_4 , v_5 \}$ and edge set $E = \{ \{v_1 , v_3\} , \{v_2 , v_3\} , \{v_2 , v_5\} , \{v_5 , v_4\} , \{v_2 , v_4\} \}$. Let's draw this graph.

In this graph, v4 is adjacent to v2 ,

and v_2 *is* adjacent to v_4 .

Suppose we assign a weight to each of the edges in the digraph G of Example 1 as follows:

$W(v_1 , v_2) = 27,\ W(v_1 , v_3) = 11\ W(v_3 , v_1) = 8$

$W(v_3 , v_2) = 14,\ W(v_2 , v_4) = 20,\ W(v_4 , v_3) = 9$.

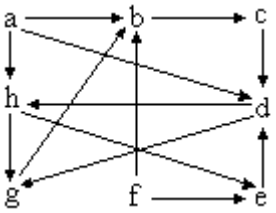
(Here W stands for "weight of".) Suppose each of these weights is regarded as the cost of traveling along the corresponding edge in the direction of the arrow. What is the least costly way to travel from v_1 to v_4 ? (This is an example of the classical Least Costly Path Problem.)

Definition: Let G be a finite graph, and let v and w be vertices in this graph. A ***path*** in G ***from*** v ***to*** w is a finite sequence of edges, say (v_0 , v_1) , (v_1 , v_2) , (v_2 , v_3) , . . . , (v_{n-1} , v_n) , in a digraph, or $\{v_0 , v_1\}$, $\{v_1 , v_2\}$, $\{v_2 , v_3\}$, . . . , $\{v_{n-1} , v_n\}$, in an undirected graph, such that $v_0 = v$ and $v_n = w$. We call such a sequence the ***edge sequence*** of the path. It is easy to see that we can equally well specify a path by specifying the ***vertex sequence*** of the path, i.e., the list of vertices that are successively encountered along the path: $v_0 , v_1 , v_2 , \ldots , v_n$. The ***length of the path*** is the number of *edges* that make up the path. For any vertex v , we'll consider the empty set of edges to constitute a path of length 0 from v to itself.

Example 3: In the digraph shown at right, the edges

(a, d) , (d, g) , (g, b) , (b, c) , (c, d) , (d, h) , (h, e)

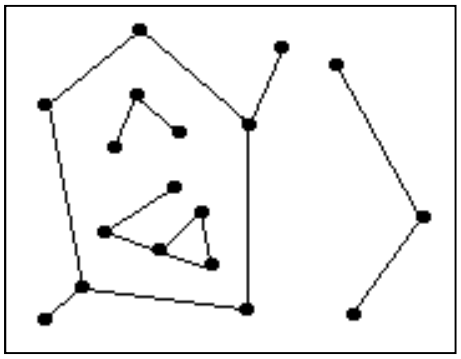
form a path from a to e . The length of the path is 7 .



Definition: A path in a finite graph is called a ***simple path*** iff no vertex is repeated in the vertex sequence of the path. A path is called a ***cycle*** (or ***circuit***) iff the first and last vertices in the vertex sequence are the same. A ***simple cycle*** (or ***simple circuit***) is a cycle such that when the last vertex is removed from the vertex sequence, the remaining sequence is a simple path. A graph is called ***acyclic*** iff it contains no cycles. [The prefix a- on words of Greek origin often means something like "without": amoral (without morals), asymmetric (lacking symmetry), amorphous (formless), atheistic (godless), arrhythmic (out of rhythm).]

In the digraph in Example 3 above, the path given there is not simple, nor is it a cycle. The path specified by the vertex sequence b, e, c, h, g is simple. The path specified by the vertex sequence a, d, g, b, c, d, h, e, a is a cycle, but not a simple cycle. The path specified by the vertex sequence b, e, c, h, g, b is a simple cycle.

Definition: An undirected graph is called ***connected*** iff for every pair of vertices x and y in the graph there is a path from x to y (and since the graph is undirected, the "reverse" path would be a path from y to x). A directed graph is called ***strongly connected*** iff for every pair of vertices x and y in the graph there is a path from x to y and also a path from y to x. A directed graph is called ***weakly connected*** iff the undirected graph derived from it by ignoring the directions on the edges is connected.



The undirected graph in Example 2 is connected. The digraph in Example 1 is strongly connected. The digraph in Example 3 is weakly connected but not strongly connected. The undirected graph at right is not connected, but it has four "connected components". We can define this notion in terms of *equivalence relations*.

Suppose we create a binary relation, call it C, on the vertex set V of any finite graph G as follows: $x \, C \, y$ if and only if there exists a path in G from x to y. (The letter 'C' was chosen so that it could be read "can be connected to".) It is easy to verify that *if* G *is undirected*, then the relation C is an equivalence relation on V.

Definition: Let $G = (V, E)$ be an undirected graph. Let C denote the equivalence relation defined on V as follows: $x \, C \, y$ if and only if there exists a path in G from x to y. Then the equivalence classes of the relation C are called the ***connected components*** of the graph G.

It is possible to define the notion of connected component of an undirected graph in a different way, as a "maximum connected subgraph" of G. The following definition and theorem provide this alternative way of looking at connected components.

Definition: A graph H with vertex set W and edge set F is a ***subgraph*** of a graph G with vertex set V and edge set E iff $W \subseteq V$ and $F \subseteq E$.

Theorem: A graph H is a ***connected component*** of a graph G iff (1) H is a subgraph of G, and (2) H is connected, and (3) there is no other *connected subgraph* K of G such that H is a subgraph of K.

Proof. The proof is not deep, but it's not very interesting to check all the details.

Here are some of the kinds of finite graph problems that people often need to solve.

Connectedness problems. Is a graph connected?

Tours. In a connected graph, find a path that moves along each edge exactly once. Alternatively, in a connected graph, find a path that visits each vertex exactly once.

Shortest route problems. Given a weighted graph in which the weights are interpreted as "distances", and given a starting vertex and a goal vertex, find a shortest path from start to goal, provided such a path exists.

Traveling salesman problems. Given a weighted graph in which the weights are interpreted as distances, find a shortest path that visits every vertex.

Maximal flow problems. Given a weighted graph in which each weight is interpreted as a "capacity" for that edge, and given a "source" vertex and a "sink" vertex, find a way of assigning "flows" to the edges to maximize the throughput from source to sink.

Task sequencing problems. Given a directed graph in which each node represents a task to be done and each edge (v, w) represents a restriction that task v must be completed before w can be started, find a way of listing all the tasks to be done in such a way that all the restrictions are met.

Minimal spanning tree problems. Given a weighted graph in which each weight is interpreted as a "cost", find a minimal-cost set of edges connect (directly or indirectly) all the vertices.

Isomorphism problems. Given two graphs, determine whether they have identical structure.

["Iso-", same; "morph", form.]

We will look at algorithms for solving most of these problems. We will practice these algorithms by carrying out the computations on rather small graphs. You should keep in mind, however, that when graph problems are solved in real world applications, the graphs may very well have thousand of vertices and tens or hundreds of thousands of edges. Any *practical* algorithm for solving a problem in graph theory will have to run efficiently on very large graphs. An algorithm that may seem straightforward when used on a small graph may not be as efficient, and therefore not as useful, as a somewhat more complicated algorithm.

Exercises

12-1 In each case below, draw the graph whose vertex set and edge set are given.

(a) V = {M, N, P, Q, R, S, T, U} ,

E = { (P, N) , (R, T) , (Q, M) , (T, P) , (R, U) , (S, P) , (U, N) , (M, S) , (T, Q) }

(b) V = {M, N, P, Q, R, S, T, U} ,

E = { {P, N} , {R, T} , {Q, M} , {R, Q} , {U, P} , {N, U} , {M, S} , {R, S} }

12-2 In each case below, draw the graph whose vertex set and edge set are given :

(a) V = {A, B, C, D, E} ,

E = { (A, B), (A, D), (A, E), (B, C), (D, A), (D, B), (E, C), (E, D) }

(b) V = {A, B, C, D, E} ,

E = { {A, B}, {A, D}, {A, E}, {B, C}, {D, A}, {D, B}, {E, C}, {E, D} }

12-3 Find all paths from R to N in the graph in Exercise 1G(a). Is that graph strongly connected? Is it weakly connected? Does the graph contain any cycles, or is it acyclic?

12-4 Find 3 different paths from A to C in the graph in Exercise 2G(a). How many simple paths are there from A to C? How many paths are there from A to C? Is that graph strongly connected? Is it weakly connected? Does the graph contain any cycles, or is it acyclic?

12-5 Find all simple paths from T to S in the graph in Exercise 1G(b). Is that graph connected? If not, identify its connected components. Does the graph contain any cycles, or is it acyclic?

12-7 Draw the digraph whose vertices are the nineteen integers from 2 to 20 and whose edge set is { (k, n) : k is a divisor of n } . For example, the graph contains an edge from 5 to 15, but no edge from 4 to 14 .

12-8 Draw the digraph whose vertices are the nineteen integers from 2 to 20 and whose edge set is { (k, n) : $k = \lfloor n/3 \rfloor$ } . For example, the graph contains an edge from 5 to 17, but no edge from 3 to 12 .

12-9 Draw the undirected graph whose vertices are the eleven integers from 20 to 30 inclusive, and whose edge set is { {m, n} : $m \neq n \ \wedge$ m and n have at least one common prime divisor } . For example, the graph will contain an edge between 21 and 28 because those vertices have 7 as a

common prime divisor, but the graph will not contain an edge between 22 and 25 because they are "relatively prime", i.e, their only common divisor is the non-prime integer 1 .

12-11 For any finite graph G , a binary relation C can be defined on the vertex set V of G in the following way: for all vertices x and y , $x \, C \, y$ if and only if there is a path from x to y in G . That is, $C = \{ (x, y) : x \in V , y \in V ,$ there exists a path in G from x to $y \}$.

(a) Prove that if G is undirected, then C is an *equivalence relation* on V .

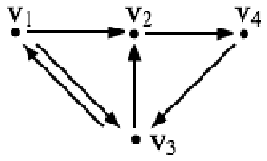
(b) Prove that every *equivalence class* of the relation C is a *connected component* of G , and vice versa.

12.1 Computer Representations of Finite Graphs

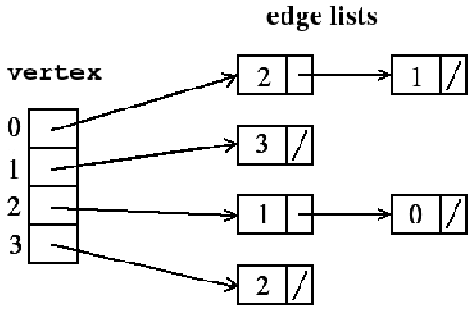
Now that we know the mathematical definitions for finite graphs, let's look at two closely related questions about implementation of graphs in computer programs. The first deals with methods by which a program can acquire the data that defines a graph. The second deals with how the data can be used by the program to build a data structure that represents the graph and supports efficient algorithms for solving typical graph theory problems (connectedness, shortest paths, etc.).

There are two major graph representations that can be found in most programs that require graph algorithms: using an *adjacency list* and using an *adjacency matrix*.

Let's start with a very simple example. Suppose we have a digraph (directed graph) with vertex set {0, 1, 2, 3} and edge set {(0, 1), (0, 2), (2, 0), (2, 1), (1, 3), (3, 2)}. If we label the vertices $\{v_1, v_2, v_3, v_4\}$, then a diagram for this graph would look like this:



In an *adjacency list*, we store for each vertex a linked list of the indexes of all the adjacent vertices, meaning the ones that edges from the vertex point to. We call it an *edge list*. Such a structure for the graph above is shown below. We can see that for example, the list attached to the vertex of index 2 (v3) contains the index 1 (for v2) and 0 (for v1), which is consistent with the edges (v3, v2) and (v3, v1) in the graph. Note that the list of adjacent vertices can be in any order. Also note that for an undirected graph, an edge {A, B} must be represented both as the index for A being in the edge list for B and the index for B being in the edge list for A.



Variations of the adjacency list for directed graph could store two edge lists for each graph if necessary for the application: the incoming edge list and the outgoing edge lists. For weighted graphs, the structure of the list node will be expanded to include the information about the weights.

In an *adjacency matrix*, we store the graph in a matrix containing values of 0 or 1, where each row represents the first vertex on an edge, and each column represents the second vertex. Thus, an edge (A, B) of indexes (*i*, *j*) will be represented as the value 1 in the cell on row *i* and column *j*. A matrix for an undirected graph will be symmetrical with respect to the first diagonal. The adjacency matrix for the graph above can be seen on the next page.

	0	1	2	3
0	0	1	1	0
1	0	0	0	1
2	1	1	0	0
3	0	0	1	0

In the case of a weighted graph, we would store the weights directly in the cells instead of the value 1. We might need a special value to represent the absence of an edge if the value 0 is a possible weight.

Which representation is better or more efficient? It depends on the application.

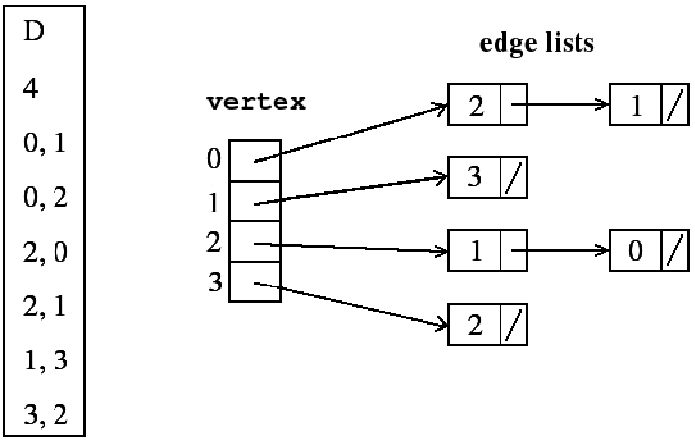
It terms of simple memory use, assuming that an integer requires the same number of bytes as a pointer (which is the case for some platforms), then the adjacency list would require $2m$ sizeof(int) storage space, where m is the number of edges. An adjacency matrix requires n^2 sizeof(short int) where n is the number of vertices. Thus, if the vertices are on average adjacent to less than half of the vertices in the graph, then an adjacency matrix will take more space in the memory.

Let us suppose that we need to know in our application if two vertices of indexes i and j are connected by an edge. In an adjacency list we would have to do a linear search in the edge list for i for the target value j. This operation is $O(n)$. For an adjacency matrix, we can directly access the cell (i, j) to find out if it's a 0 or a 1, operation which is $\Theta(1)$.

Now let us suppose that we need to perform an operation for all the vertices adjacent to a particular one of index i. In an adjacency list, we would use a list traversal loop for which the number of iterations will be equal to the number of vertices adjacent to i. In an adjacency matrix, we would have no choice but to traverse the entire row i to see which values are equal to 1, operation that will always require n iterations.

12.2 File Representation

Now let us consider the problem of representing a graph in a file. For the graph shown on the previous page, the figure at the top of the next page show the edge list on the right and a possible content of a text input file that could be created to unambiguously specify this graph on the left. The letter D in the first line of the input file indicates that the graph contained in the file is a digraph. The number 4 on the second line specifies that the graph contains exactly four vertices, which then are assumed to be 0, 1, 2, 3. The remaining lines list the edges of the graph. From the letter D we know that these are all ordered pairs. (The letter U will be used to say the pairs are unordered, i.e., the graph is undirected).



A program can open this file and begin reading it and building an adjacency list for the graph. The array named `vertex` has a cell for each vertex, and for each vertex `i` the cell `vertex[i]` contains a pointer to a linked list of all the vertices adjacent to vertex `i` . For example, the pointer in `vertex[2]` points to a list containing the vertices 1 and 0, indicating that there is an edge from 2 to 1 and a separate edge from 2 to 0. (Caution: nothing in that list should be interpreted as saying there is an edge from 1 to 0 in the graph.)

Here is some pseudo-code that describes how a program could build the adjacency structure while it is reading a graph input file. It is assumed that the vertex set is {0, 1, 2, . . . n – 1} for some integer n.

1. Read the first line of the input file into a character variable `ch`.
   `ch` will be either D or U, for ”directed” or ”undirected”
2. Read the second line into an integer variable `n`.
3. Allocate an array, call it `vertex` , of `n` empty linked lists of integers.
4. While not at the end of the input file,
5. read the two integers, call them `i` and `j`, from the next line of the file;
6. insert the integer `j` at the front of the list `vertex[i]`;
7. if the character `ch` is a ’U’ then
8. *there is another task to be done; this will be explained farther along.*

Now let us consider the case where the vertices in the graph have labels that are strings and not just integers. Then the file containing the graph description should also list those labels. To make the description more complete, let’s also assume that the graph is weighted. Then the lines specifying the edges will have to provide the weight information. The file could have the following format:

(1) the first line of the file contains either the letter D or the letter U , to tell us whether the graph is ***d***irected or ***u***ndirected;

(2) the second line of the file contains the number of distinct vertices that make up the graph;

(3) the names of the vertices are listed next, one name per line;

(4) the remaining lines of the file contain descriptions of the edges; each line contains the names (character strings) of two vertices, separated by a comma and a blank that immediately follows the first name; a comma and blank also follow the second name; these are followed by an integer that represents the weight of the edge.

Here is an example of such a file:

```
U
4121
Frankfurt
Tehran
Mexico City
Paris
...
South Bend, Paris, 4225
Nairobi, South Bend, 7806
Boston, Calcutta, 13410
Sidney, Calcutta, 8007
...
```

It is not hard to write a program that can read this file and construct some internal representation of the graph so that various algorithms can be carried out on the graph. That internal representation, however, will almost certainly require that the vertices be used as indexes into one or more arrays. This immediately presents us with a problem. If the vertices are given as character strings, they cannot be used as indexes into an array. To overcome this problem, almost all programs that implement finite graph algorithms begin by assigning ***sequence numbers*** to the vertex names as they are read from the input file, and then the sequence numbers are used internally to represent the vertices. When the information about the edges of the graph is read from the file, it will be necessary to be able to convert vertex names to sequence numbers rapidly, and so the vertex names are usually stored in a hash table along with their sequence numbers. In the example above, the number 4121 near the top of the file gives the program a way of knowing what a suitable size for the hash table array will be, and then an array of that size can be dynamically allocated. For a hash table that uses the method of linear probing, an array that contains about two and a half times as many cells as the number of vertices should give good performance. In our example, that would be a little over 10,000. Each cell in the array would consist of a place to store the sequence number of a vertex and a place to store a pointer to a dynamically allocated character array for the corresponding vertex name.

After space for the hash table has been allocated, the program can begin reading the list of vertices from the input file. Each time it reads a vertex name, it attempts to insert that name into the hash table along with the sequence number for that name. If the insertion attempt fails for any reason (such as the name already being in the table), this should cause processing to halt with an error message. The sequence numbers are placed in the hash table along with the vertex names because when the edges are read from the input file, it will be necessary to hash each of the two vertex names of the endpoints of the edge in order to retrieve the sequence numbers for those vertices.

When the program has finished executing whatever algorithm it has been designed to perform, it will be necessary to have the results printed out using the external vertex names that were

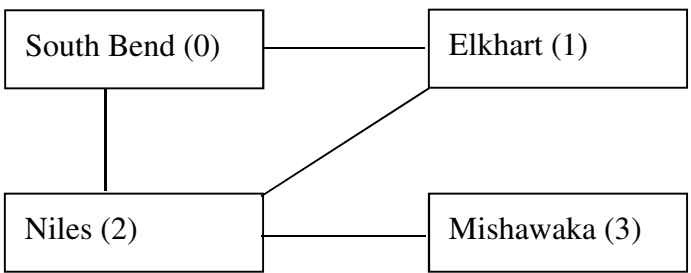
in the input file. In order to be able to recover the vertex names rapidly from the sequence numbers, a "name retrieval array" should be created at the same time that the hash table is set up, and each name inserted into the hash table should have a pointer to it placed in the "name retrieval array" at the cell indexed by the sequence number.

Let's take as an example the input file shown on the preceding page. We'll read the first vertex name, "Frankfurt", into a buffer and assign it the sequence number 0 . That is, we set up a "record" with a sequence number field and a vertex name field, and we put 0 in the sequence number field and in the vertex name field we put a pointer to the buffer holding the vertex name. Then we call a hash table insertion function to insert a copy of this record into the hash table. If the insertion succeeds, then we put a pointer to the hash table's copy of the string "Frankfurt" in cell 0 of the name retrieval array. Note: the insertion function for the hash table can be made to return a pointer to the vertex name that's stored in the hash table.

The second vertex name in the file is "Tehran". We read this into our buffer (overwriting "Frankfurt") and assign it the sequence number 1 by creating a record containing 1 and a pointer to "Tehran" in the buffer. Then we insert a copy of this record into the hash table and put a pointer to the hash table's copy of the string "Tehran" into cell 1 of the name retrieval array. Etc.

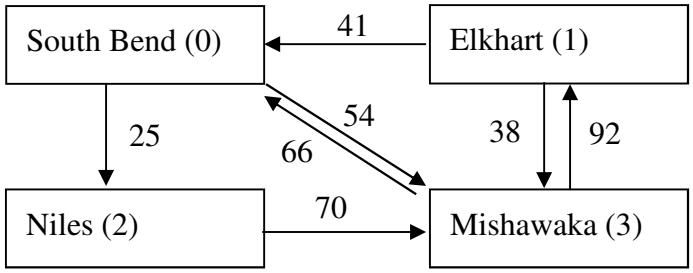
After the vertex names have been read from the input file and stored in the program, the edge information is read and placed into some internal representation of the graph. This brings us to an important question: how shall we represent a graph within the computer program? We could adopt the plan of trying to mimic as nearly as possible the actual structure of the graph by creating "nodes" that represent vertices of the graph and placing pointers in these nodes to represent the edges of the graph. The problem with this is that some nodes may need space for many pointers, while others may need space for only a few, or even none. This non-uniformity does not prevent us from using this method if we are determined to do so. A little reflection, however, reveals that the construction of this mass of nodes and pointers is not easy to do in an *efficient* way during the process of reading the input file.

One much simpler possibility is to use an ***adjacency matrix***. This is a two-dimensional array, indexed on each of two sides by the sequence numbers of the vertices. For an unweighted graph, a 1 is used in row i column j to indicate the presence of an edge in the graph from the vertex represented by i to the vertex represented by j . A 0 is used to indicate the absence of an edge. If the graph is weighted, then the weights can be placed in the cells of the array, with a special "sentinel" value used to indicate the absence of an edge. Below are two examples. The first gives the adjacency matrix representation of an undirected, unweighted graph. The second gives the matrix representation of a weighted digraph (the number -1 has been used as the sentinel value).




```
    0  1  2  3
0 [ 0  1  1  0 ]
```

```
1 | 1   0   1   0 |
2 | 1   1   0   1 |
3 | 0   0   1   0 |
```



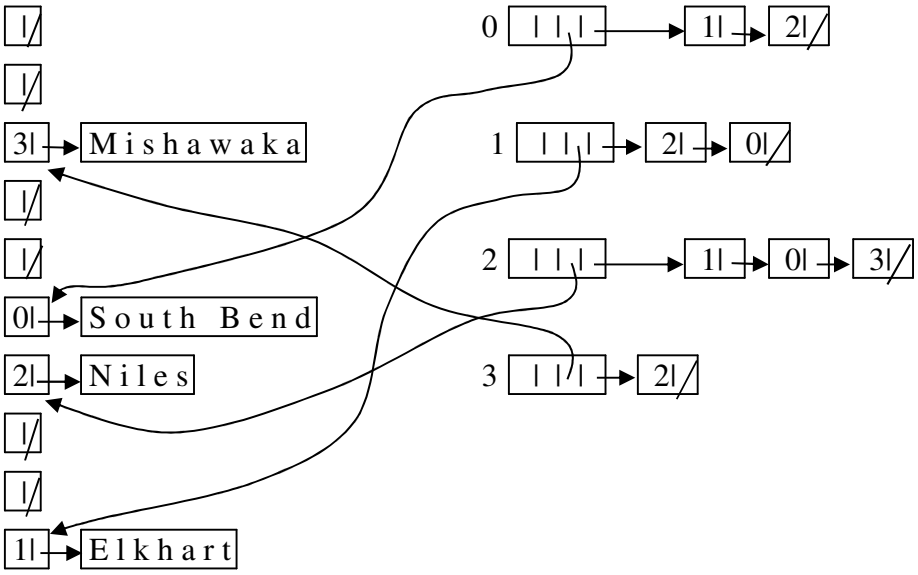

```
     0    1    2    3
0 | -1   -1   25   54 |
1 | 41   -1   -1   38 |
2 | -1   -1   -1   70 |
3 | 66   92   -1   -1 |
```

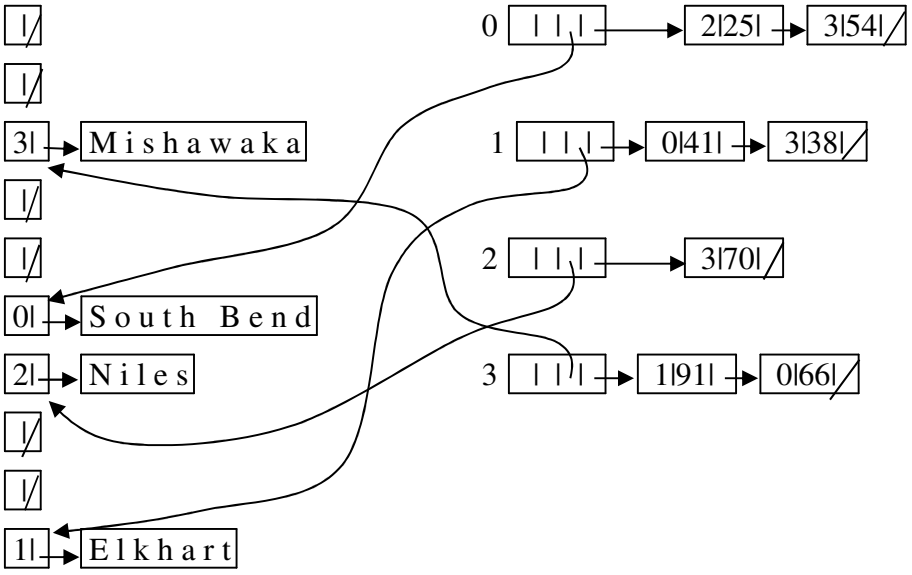
Note that the elements on the diagonal running from upper left to lower right always indicate no edge, because the graphs we are considering have no self-loops. Also notice that the adjacency matrix for an undirected graph must be symmetrical across that diagonal, because if there's an edge from i to j, then the same edge is considered to be an edge from j to i.

Adjacency matrices tend to work well if the graphs on which we will be performing operations are ***dense***, by which we mean that the number of edges is a significant fraction of the maximum possible number of edges. If, however, the number of vertices is large and the graph is ***sparse***, by which we mean that most vertices are connected by edges to relatively few other vertices, then the adjacency matrix is huge, and most of the entries are just indicators of no edge. This is a waste of precious memory space. To represent a sparse graph (which is what most graphs with many vertices are), we normally use a dynamically allocated ***adjacency structure*** consisting of an array, called the ***vertex array***, indexed by the sequence numbers of the vertices. The cells of the vertex array contain pointers to ***edge lists*** . This vertex array can also be used to hold the vertex name pointers, so that a separate "name retrieval array" is not necessary.

On the next page are diagrams showing the adjacency structure representations of the two graphs shown above.

It is important to understand that in this representation the pointers in the edge lists do NOT represent edges. They are simply the links in the lists of sequence numbers of vertices to which there is an edge in the graph from the vertex whose sequence number is the array subscript.





Note that in the adjacency structure for undirected graph (the first structure shown above) there are two list nodes representing each individual edge. This is because we must keep track of both directions along which we can "travel" along an edge in an undirected graph. Thus there are 8 list nodes to represent the 4 edges of the graph. In the adjacency structure of the directed graph there is only one list node for each directed edge: 7 edges are represented by 7 list nodes.

When we begin looking at algorithms for solving various graph problems, we will see that the cells that make up the vertex array in an adjacency structure must sometimes carry more information than just a pointer. In some cases, each cell of the vertex array will be a structure with several data members. This suggests that when the file is read and the adjacency structure is built, the program that does this work will generally be a special purpose program for solving a particular problem, or closely related problems.

12.3 The Data Structure and A Few Functions

The class shown below is using the C++ STL classes **list**, **vector**, and **iterator**. A few class methods are shown on the next pages. To use the C++ STL **list** class, we'll define the following types:

```
typedef list<int> ListInt;
typedef ListInt::iterator ListIntIter;
```

As a reminder, the class list from the STL provides functions **push_front** and **push_back** to insert elements in a list, **clear** to erase the content of the list. We'll also assume that we have a function with the prototype

```
bool isInList(int i);
```

returning true if the integer **i** is in the list.

Furthermore, we'll assume that we have a class called **HashTable** storing pairs of a string and an integer. The string represents a vertex name, while the integer is an index in the vector of vertices. The class might have the following definition:

```
class HashTable
{
protected:
    struct TableRecord     // Data type of the objects to be stored in
    {                      // the hash table based on their name.
        int  index;        // The "index" need not have any obvious
        char *name;        // relationship to the "name".
    };
    vector<TableRecord> hashArray;
public:
    HashTable(int size);   // Constructor.  Allocates the hash array and
                           // defines a hash function, both of which are
                           // derived from the "size" parameter.
    ~HashTable();          // Destructor. Deletes the hash array.
    void insert(int i, char *s); // Creates a TableRecord and containing i
                                 // and s and inserts it into the table.
    int index (char *s);  // Returns the index associated with "s" if
                          // "s" is in the table.  Otherwise returns -1.
    void MakeEmpty();     // Deallocates memory that's been allocated by
                          // the HashTable methods;
      ...
};  // Code will not be supplied for the HashTable methods.
```

We will assume that the implementation follows the ideas in Chapter 5. The closest class to a hash table in the STL is an unordered map.

```
class Graph
{
protected:
    struct vertex
    {
        ListInt edgeList;
        string name;
    };
    int numVertices, numEdges;
    bool directed;
    vector<vertex> vertices;
    HashTable namesIndexes;
    void addEdge(const int index1, const int index2);
public:
    void makeEmpty();
    Graph();
    ~Graph();
    void addVertex(const string theName);
    void addEdge(const string name1, const string name2);
    void print(); // output the data in the graph
    void read(const char *filename); // Reads the graph from a file
};

// Default constructor
Graph::Graph()
    : vertices(0), namesIndexes()  // calling the constructors for the
                                   // vector and the hash table
{
    numVertices = 0;
    numEdges = 0;
    directed = false; // making the graph undirected by default.
} // Graph::Graph()

// Destructor: we only need to call the function make empty.
Graph::~Graph()
{
    makeEmpty();
} // Graph::~Graph()
```

In the function **makeEmpty**, we must delete all the memory that has been allocated dynamically. This means that we must first empty all the edge lists for all the vertices. We can use the function clear defined in the class list for this purpose. After this is done, we also need to delete the content of the vector of vertices, because each vertex has been added to it individually. The function clear from the class vector accomplishes this. Finally, we must also erase the data from the hash table, for which purpose we employ the function that we assumed to be defined in this class.

```
// A function that deletes the entire content of the graph object.
void Graph::makeEmpty()
{
    if (numVertices)
    {
        // erase the data from all the edge lists
        for (int i = 0; i < numVertices; ++i)
            vertices[i].neighbors.clear();
        // then erase the vector itself
        vertices.clear();
        // erase the data from the hash table of names indexes
        namesIndexes.makeEmpty();
        numVertices = 0; // set these values properly
        numEdges = 0;    // in case the graph is reused
        directed = false;
    }
} // Graph::makeEmpty()
```

The next set of functions can be used to add vertices and edges dynamically to the graph. The vector STL class allows us to add more elements to the vector with the function **`push_back`**. The list STL class also provides a function by the same name. In the function **`addVertex`**, we also add the name to the hash table.

```
// Adding a vertex based on its name. The vertex is simply added at the
// back of the vector of vertices.
void Graph::addVertex(const string theName)
{
    vertex newVertex;               // create a new vertex
    newVertex.name = theName;       // store the name in it
    vertices.push_back(newVertex); // then insert it in the vector
    namesIndexes.insert(theName, numVertices);  // also add the name to
    ++numVertices;                              // the hash table
} // Graph::addVertex()

// A function returning the index of a name in the vertices vector. If the
// name doesn't exist in the table, it is added to it.
int Graph::getIndex(const string theName)
{
    int idx = namesIndexes.index(theName);
    if (idx < 0)            // was the name found in the table?
    {
        addVertex(theName); // if not, add it
        idx = numVertices-1; // we know it's the last one now
    }
    return idx;
} // Graph::getIndex()
```

For the functions **addEdge**, we have two functions by the same name. The first one receives the names of the two vertices as strings, and has to identify their indexes in the vertices array before adding the edge. In the second one, we assume that the indexes of the two vertices are already known, so we only need to perform the insertion operation on the appropriate edge list. The second function is protected because it is used internally in the class.

```
// Add an edge with vertices specified by their names. The function first
// determines the position of each name in the vertex vector, then it
// calls the function addEdge that takes two integers as parameters.
void Graph::addEdge(const string name1, const string name2)
{
    int index1 = getIndex(name1);
    int index2 = getIndex(name2);
    addEdge(index1, index2); // Call to the overloaded function below
} // Graph::addEdge()

// Add an edge with vertices specified by index.
void Graph::addEdge(const int index1, const int index2)
{
    vertices[index1].edgeList.push_back(index2);
    if (!directed)
        vertices[index2].edgeList.push_back(index1);
    ++numEdges;
} // Graph::addEdge()
```

The last function discussed here is a function printing the content of the graph by simply traversing all the edge lists. The standard template library list class doesn't give direct access to nodes. Instead, we have to use an iterator, which is an object encapsulating a node pointer. Iterators have two important functions to be noticed in this function. First, the operator ++ advances the iterator from one node of the list to the next. Second, the operator * retrieves the content of the node.

The class list provides two functions also used here, called begin and end, that return an iterator containing a pointer to the first and the last nodes of the list respectively. The STL list is a doubly- linked circular list. The last element is there for the purpose of the list structure and should neither be dereferenced, nor used to store any data.

One last remark here: the function inserting a vertex in the graph does not check if the name has been used in the graph before. Typically, a function insert defined in a hash table would return a true or false value to indicate if the insertion was successful or not. An unsuccessful outcome is an indication of a record with the given key (the name in our case) already existing in the table, or a memory allocation failure if the records are stored externally. We leave it as an exercise to add a proper failure check to this function.

The following function prints the entire content of the graph to the console. In this function we need a loop browsing the vector of vertices, and also for each vertex, a loop browsing its edge list. The latter is implemented using iterators. This function is a good example of simple graph traversal.

```
// Print the content of the graph.
void Graph::print()
{
    int i;
    if (directed)
        cout << "The graph is directed" << endl;
    else
        cout << "The graph is undirected" << endl;

    cout << "The graph contains " << numVertices
         << " vertices and " << numEdges<< " edges" << endl;

    // print all the names
    cout << "The vertex names are: ";
    for (i = 0; i < numVertices; ++i)
        cout << vertices[i].name << ' ';

    // print all the adjacency lists
    cout << endl << "The adjacency lists are:" << endl;
    for (i = 0; i < numVertices; ++i) // for each vertex
    {
        cout << i << " --> ";
        ListIntIter iter;
        iter = vertices[i].edgeList.begin(); // first node in the
                                             // edge list
        while (iter != vertices[i].edgeList.end()) // we have not reached
                                                   // the end yet
        {
            // here we have an edge from i to *iter.
            cout << *iter << ' '; // output the content of the node
            ++iter;               // move to the next node
        }
        cout << endl;
    }
} // Graph::print()
```

Finally, let's take a look at the function reading a graph from a file. We assume that this is a file such as described in section 12.2 and that the graph is not weighted.

```
// Reads the graph from a file
void Graph::read(const char *filename)
{
    ifstream fin(filename);
    char buffer[20], buffer1[20];
    int nrv;
    fin >> buffer; // read the letter U or D
    if (buffer[0] == 'd' || buffer[0] == 'D')
        directed=true;
    fin >> nrv; // read the number of vertices

    for (int i=0; i < nrv; i++) // read all the names of vertices
    {
        fin >> buffer; // read the vertex name
        addVertex(string(buffer)); // this will also update numVertices
    }

    // read all the edges until the end of the file
    while (!fin.eof() && fin.good()) // we have not reached the end
    {                                // of the file
        fin >> buffer;
        if (strlen(buffer) && fin.good()) // was the first name read
        {                                 // correctly?
            fin >> buffer1;
            if (strlen(buffer) && strlen(buffer1)) // how about the
            {                                      // second?
                // we know both names are valid here
                buffer[strlen(buffer)-1]='\0'; // Deleting the ','
                addEdge(string(buffer), string(buffer1));
            }
        }
    }
    fin.close();
} // Graph::read()
```

Exercises

12-13 Suppose a file in the format described on page 6 looks like this:

```
D
4
IUSB
Purdue
Bloomington
Ball State
IUSB, Purdue, 34
Bloomington, Ball State, 76
Purdue, Bloomington, 29
Ball State, IUSB, 85
Bloomington, IUSB, 51
Purdue, Ball State, 27
Bloomington, Purdue 10
```

(a) Draw the digraph represented by the information in this file.

(b) Suppose that a program that reads this file allocates a hash table of size 11 (indices 0 through 10) for the sequence number and name records. It assigns the sequence number 0 to "IUSB" and hashes the string to address 4. It assigns the sequence number 1 to "Purdue" and hashes the string to address 9. It assigns the sequence number 2 to "Bloomington" and hashes the string to address 6. It assigns the sequence number 3 to "Ball State" and hashes the string to address 9 , but then uses linear probing to go on to address 10. Draw a picture of the hash table.

(c) Draw the adjacency matrix for this graph, using the sequence numbers assigned in part (b). Try to imagine how a program would construct this matrix as it reads through the file.

(d) Draw the adjacency structure that would result from constructing the edge lists by adding each new edge node to the *end* of the list.

(e) Draw the adjacency structure that would result from constructing the edge lists by adding each new edge node to the *front* of the list.

12-15 Consider the graph in Exercise 13. Write out the names of the vertices along the path whose vertex list is 1, 3, 0, 2 (these are the vertex sequence numbers of the vertices). Pretend that you are doing this problem the way a *program* would do it, using the "name pointers" in the vertex array and the names in the hash table given in the answer for Exercise 13.

12-17 Suppose a file contains specifications for a graph in the following format. The first line is U or D depending on whether the graph is undirected or directed. The second line gives the number of vertices. A list of the vertices follows on succeeding lines, with some "weight" or "score" or "value" assigned to each vertex. Then the edges are listed on the remaining lines, but the edges do not have weights. Suppose a file in this format looks like this (the vertex weights are rough population figures, in millions):

```
U
```

```
5
Texas 20
California 43
Nevada 2
Pennsylvania 14
Indiana 9
Texas, California
California, Nevada
California, Pennsylvania
Pennsylvania, Indiana
```

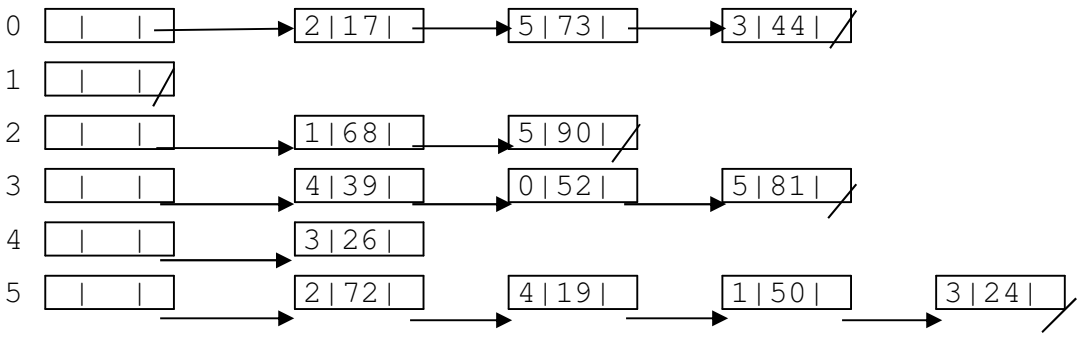
(a) Draw the graph represented by the information in this file.

(b) Suppose that sequence numbers are assigned (as they would be) in the order in which the vertices are listed. Suppose that the hash table for names consists of 13 locations (addresses 0 through 12), and suppose the hash function is the rather crude one of counting the letters in the string, multiplying the length by 7, dividing the product by 13, and using the remainder as the hash table address. (Example: Texas hashes to address 9). Build the hash table that records the sequence number assignments. Also draw the adjacency structure that would result from constructing the edge lists by adding each new edge node to the front of the list. Each cell in the vertex array should contain the weight of the vertex as well as a pointer to the edge list. Think about how a program would build this adjacency structure while it was reading the input file.

12-19 Draw the undirected graph represented by the following adjacency matrix. Use the sequence numbers to label the vertices of the graph.

	0	1	2	3	4	5
0	0	0	1	0	1	1
1	0	0	0	1	1	0
2	1	0	0	1	1	1
3	0	1	1	0	0	0
4	1	1	1	0	0	1
5	1	0	1	0	1	0

12-21 Draw the weighted digraph represented by the following adjacency structure. Use the sequence numbers to label the vertices of the graph.



Breadth First Search and Depth First Search

Most algorithms that we perform on finite graphs involve searching the graphs in a systematic manner and processing the information that we find as we search. Almost all graph searches employ one of the following two search strategies: ***breadth first search*** and ***depth first search***.

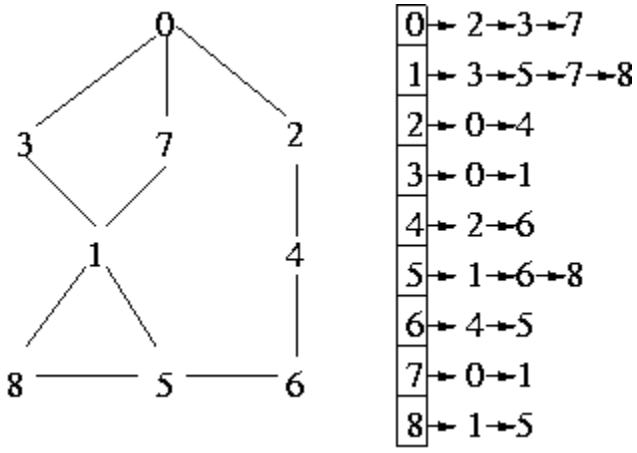
In a *breadth first search* we start at some vertex v, process it in whatever way is appropriate, and mark it as having been "reached". Then we take each of its neighbors in turn, process the edges from v to those neighbors (if edges need to be processed), process the neighbors themselves, and mark all the neighbors as having been reached. Then for each of those neighbors, taken one at a time, we process the edges out of them, and process and mark the neighbors at the ends of those edges. We continue in this way until all vertices that can be reached have been marked. To put it differently, a breadth first search strategy processes some initial vertex v, then processes all vertices to which there is a path of length 1 from v, then processes all vertices to which there is a path of length 2 from v, and so on. The collection of processed vertices spreads out like an ink blot around that initial vertex v.

To implement a breadth first search, we have to keep a list of which vertices need to be processed. The list also helps us process them in the correct order. The kind of list we must keep is a FIFO list, which is to say, a standard ***queue***.

Here is some pseudo-code for a breadth first search in a graph G starting from a prescribed vertex v.

```
void Graph::breadthFirstSearch (int v)  // v is a vertex number
{
    // initialize all vertices as "unmarked";
    bool mark[numVertices];
    for (int i = 0; i < numVertices; ++i)
        mark[i] = false; //initialize all vertices as "not reached".
    queue<int> Q;
    mark[v] = true;
    perform preliminary processing on v;
    Q.enqueue(v);
    while (Q is not empty)
    {
        x = Q.dequeue();
        for each neighbor y of x in vertices[x].edgeList
        {
            process the edge from x to y if necessary;
            if (!mark[y])
            {
                mark[y] = true;
                perform preliminary processing on y;
                Q.enqueue(y);
            }
        }
        perform final processing on vertex x;
    }
} // Graph::breadthFirstSearch()
```

Let's look at a breadth first search on a graph the way it would be carried out in a program in which the graph is implemented by an adjacency structure. A sample undirected graph and one possible adjacency structure for it are shown below.



Suppose we start the search at vertex 0. Our first step would be to mark vertex 0 as "reached" (or "visited") and then to place the number 0 in an empty queue Q. Then in the main "while" loop we

- dequeue 0 into x ;
- set an iterator p to the head of the edge-list for vertex 0; this will allow us to traverse the edge list, thereby implementing the loop indicated by "for each neighbor y of x in G";
  - process the edge from 0 to 7 if necessary;
  - mark vertex 7, perform preliminary processing on 7, and put 7 into Q;
  - advance the iterator p to the next edge node;
  - process the edge from 0 to 2 if necessary;
  - mark vertex 2, perform preliminary processing on 2, and put 2 into Q;
  - advance iterator p to next edge node;
  - process the edge from 0 to 3 if necessary;
  - mark vertex 3, perform preliminary processing on 3, and put 3 into Q;
  - advance iterator p and discover we've reached the end of the edge-list;
- perform final processing on vertex 0;
- dequeue 7 into x;
- set an iterator p to the head of the edge-list for vertex 7;
  - process the edge from 7 to 0 if necessary;
  - since 0 is already marked, advance the pointer p to the next edge node;
  - process the edge from 7 to 1;
  - mark vertex 1, perform preliminary processing on 1, and put 1 into Q;
  - advance iterator p and discover we've reached the end of the edge-list;
- perform final processing on vertex 7; ETC.

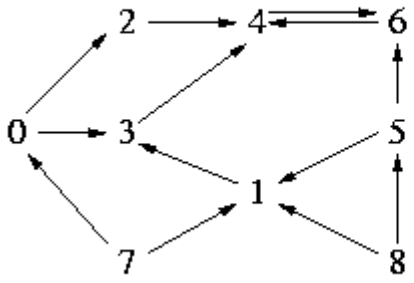
To summarize the previous consideration, here is a table of the program execution:

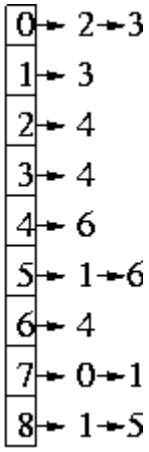
Reached	x	y	Q
none			empty
0			0
0	0		empty
0, 2		2	2
0, 2, 3		3	2 3
0, 2, 3, 7		7	2 3 7
0, 2, 3, 7	2		3 7
0, 2, 3, 7		0	3 7
0, 2, 3, 4, 7		4	3 7 4
etc.			

Here are the values taken by Q from the beginning to the end of the function execution:

Q: (empty) –⟩ (0) –⟩ empty –⟩ (2) –⟩ (2 3) –⟩ (2 3 7) –⟩ (3 7) –⟩ (3 7 4) –⟩ (7 4) –⟩ (7 4 1) –⟩ (4 1) –⟩ (1) –⟩ (1 6) –⟩ (6) –⟩ (6 5) –⟩ (6 5 8) –⟩ (5 8) –⟩ (8) –⟩ empty

What would happen if we made a breadth first search of the graph shown below, starting at vertex 0 ? (The fact that the graph is directed makes no difference in the strategy.)





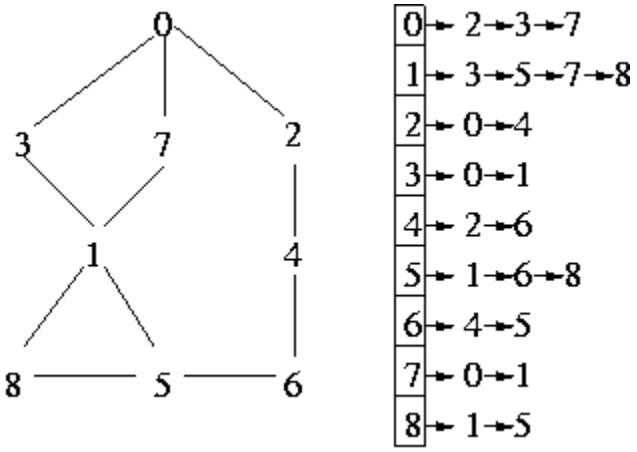
As you can see, not every vertex of the graph would end up marked "reached". If we wanted to process all edges and vertices of the graph, we would have to scan down the array of the adjacency structure until we came to an unmarked vertex. Then we could perform another breadth first search starting at that vertex. When we finished, we would again continue our scan down the array. In this way we would eventually conduct enough breadth first searches to completely process all parts of the graph.

Now let's look at the other search strategy: ***depth first search***. We start at some vertex v, process it in whatever way is appropriate, and mark it as having been reached. Then we select one of its neighbors, call it w, process the edge from v to w, and mark w as having been reached. Then, instead of processing another neighbor of v, we select an unmarked neighbor of w, call it x, process the edge from w to x, and mark x as reached. Then we select an unmarked neighbor of x, call it y, process the edge from x to y, and mark y as having been reached. We keep on going "deeper and deeper" into the graph until we cannot reach any more unmarked nodes. Then we ***backtrack*** until we get back to a node that has an unmarked neighbor, and we start down that path. We keep on going as far as possible through unmarked nodes, and then we backtrack to a vertex that has an unmarked neighbor. Again we go as far as possible, then backtrack, and continue this process until finally we exhaust all possible paths away from the original vertex v.

How can we keep track of what path we had followed during a "forward search" so that we will be able backtrack *easily* whenever that becomes necessary?

Here is some pseudo-code for a depth-first search on a graph G starting from a prescribed vertex v. A slightly more efficient version of the algorithm is described in Exercise 12-29.

```
Graph::depthFirstSearch (int v)  // v is a vertex number
{
    bool mark[numVertices];
    for (int i=0; i< numVertices; i++)
        mark[i]=false; //initialize all vertices as "not reached".
    stack<(int, vertex*)> S;
    mark[v]=true; // mark the vertex v as "reached";
    perform preliminary processing on v;
    ListIntIter p;
    p = vertices[v].edgeList.begin();
    S.push(v,p);
    while (S is not empty)
    {
        // p points to some node in the edge-list of vertex v
        (v,p) = S.pop();
        perform intermediate processing on v if necessary;
        while (p != vertices[v].edgeList.end())
        {
            w = *p; // the neighbor of v pointed to by p;
            process the edge from v to w if necessary;
            p++; // advance p by one node in the edge-list of v;
            if (!mark[w]) { // w has not been reached
                S.push(v,p);
            v = w;
            mark[v] = true; // mark v as "reached";
            perform preliminary processing on v;
            //make p point to the edge-list of v;
            p = vertices[v].edgeList.begin();
        }
        perform final processing on vertex v;
    }
} // Graph::depthFirstSearch()
```



The graph that was first given on page 14 is repeated above. Let's see how a depth first search starting at vertex 0 would process the adjacency structure of the graph.

First we initialize all "reached" fields as FALSE. Then we create an empty stack S. Then we mark vertex 0 as "reached" and perform preliminary processing on 0. Then we set a pointer p to the edge-list of 0 and push (0, p) onto S. Now S contains a pair consisting of 0 and a pointer to the node containing 7 in the edgelist of 0. Since S is not empty, we do the following:

- pop the pair (0, pointer to 7) from S into v and p; v = 0 ; p points to 7 in edgelist of 0
- perform intermediate processing on 0 if necessary;
- since p ≠ NULL,
  - w = 7;
  - process the edge from 0 to 7 if necessary;
  - advance p so that it points to 2 in the edgelist of 0;
  - since 7 has not been reached,
    - push (v, p) onto stack S; now S contains (0, pointer to 2 in edgelist of 0);
    - v = 7 ;
    - mark 7 as reached and do preliminary processing on 7;
    - make p point to the node 0 in the edgelist of 7;
- since p ≠ NULL,
  - w = 0;
  - process the edge from 7 to 0 if necessary;
  - advance p so that it points to 1 in the edgelist of 7;
  - since 0 has been reached, do nothing;
- since p ≠ NULL,
  - w = 1;
  - process the edge from 7 to 1 if necessary;
  - advance p so that it points to NULL (conceptually it's at the end of the edgelist of 0);
  - since 1 has not been reached,
    - push (v, p) onto stack S; now S contains (7, NULL)
    - (0, pointer to 2 in edgelist of 0);
    - v = 1 ;

    - mark 1 and do preliminary processing on 1;
    - make p point to the node 3 in the edgelist of 1;
- since p ≠ NULL,
    - w = 3;
    - process the edge from 1 to 3 if necessary;
    - advance p so that it points to 7 in the edgelist of 1;
    - since 3 has not been reached,
        - push (v, p) onto stack S; now S contains (1, pointer to 7 in edgelist of 1)
        - (7, NULL)
        - (0, pointer to 2 in edgelist of 0);
        - v = 3 ;
        - mark 3 as reached and do preliminary processing on 3;
        - make p point to the node 1 in the edgelist of 3;
- since p ≠ NULL,
    - w = 1;
    - process the edge from 3 to 1 if necessary;
    - advance p so that it points to 0 in the edgelist of 3;
    - since 1 has been reached, do nothing;
- since p ≠ NULL,
    - w = 0;
    - process the edge from 3 to 0 if necessary;
    - advance p so that it points to NULL (conceptually it's at the end of the edgelist of 3);
    - since 0 has been reached, do nothing;
- since p is now NULL, escape from that loop;
    - perform final processing of vertex 3;
- since the stack S is not empty, pop the pair (1, pointer to 7 in edgelist of 1) into (p, v) ; now v = 1 ; p points to 7 in edgelist of 1 ; **WE HAVE BACKTRACKED TO VERTEX 1**.
    - perform intermediate processing of vertex 1 ;
- since p ≠ NULL,
    - w = 7;
    - process the edge from 1 to 7 if necessary;
    - advance p so that it points to 5 in the edgelist of 1;
    - since 7 has been reached, do nothing;
- since p ≠ NULL,
    - w = 5;
    - process the edge from 1 to 5 if necessary;
    - advance p so that it points to 8 in the edgelist of 1;
    - since 5 has not been reached,
        - push (v, p) onto stack S; now S contains (1, pointer to 8 in edgelist of 1)
        - (7, NULL)
        - (0, pointer to 2 in edgelist of 0)
        - v = 5 ;
        - mark 5 as reached and perform preliminary processing on 5;
        - make p point to the edgelist of 5;
- since p ≠ NULL,
    - w = 1;

- process the edge from 5 to 1 if necessary;
- advance p so that it points to 8 in the edgelist of 5;
- since 1 has been reached, do nothing;

- since p ≠ NULL,
  - w = 8;

ETC.

We can draw a table of the program execution in this case:

Reached	**(v,*p*)**	**w**	**S**
none	(0,*2*)		empty
0	(0,*2*)		(0,*2*)
0	(0,*2*)		empty
0, 2		2	
0, 2	(0,*3*)		(0,*3*)
0, 2	(2,*0*)		(0,*3*)
0, 2		0	(0,*3*)
0, 2	(2,*4*)		(0,*3*)
0, 2, 4		4	(0,*3*)
0, 2, 4	(4,*2*)		(0,*3*),(2,*4*)
0, 2, 4	(4,*2*)	2	(0,*3*),(2,*4*)
0, 2, 4	(4,*6*)		(0,*3*),(2,*4*)
0, 2, 4, 6		6	(0,*3*),(2,*4*),(4,*6*)
etc.			

The following lists show the content of S when we reach the deepest we can go into the graph and we have to backtrack to a previous position:

(0,*3*),(2,*4*),(4,*6*),(6,*5*),(5,*1*),(1,*3*),(3,_)
(0,*3*),(2,*4*),(4,*6*),(6,*5*),(5,*1*),(1,7),(7,_)
(0,*3*),(2,*4*),(4,*6*),(6,*5*),(5,*1*),(1,*8*),(8,_)

We can see that after the last version of the list S, we have marked all the nodes and the algorithm stops (the backtrack doesn't find any more nodes with unmarked neighbors).

Although we can work our way through the algorithm as we have just been doing, it remains somewhat unclear why it should work. There is a better way to write the algorithm. It takes advantage of the fact that after processing the first vertex and moving on to the second, we are essentially starting over on a depth first search in the graph. This suggests that the algorithm can be written *recursively*, and that is indeed the case. Some pseudo-code for the recursive version is given below. If you work through the example at the top of page 17-Graphs, starting at vertex v = 0 and tracing through the recursive calls and their returns, you will find that the operations on the adjacency structure are carried out in exactly the same order as they are when we use the non-recursive version with its explicit stack.

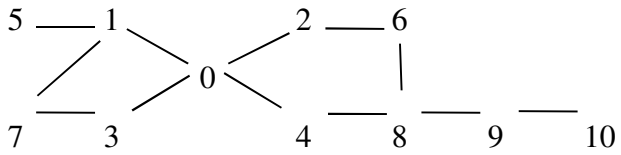
Question: how does the recursive version do its backtracking if it does not contain any reference to an explicit stack?

```
bool mark[numVertices];
Graph::depthFirstSearch(int v)
{
    for (int i=0; i< numVertices; i++)
        mark[i]=false;
    recursiveDepthFirstSearch(v);
} // Graph::depthFirstSearch()

Graph::recursiveDepthFirstSearch (int v) // v is a vertex number
{
    mark[v] = true; // mark vertex v as "reached";
    perform preliminary processing on v;

    // set an iterator p referencing the edgeList of v;
    ListIntIter p;
    p = vertices[v].edgeList.begin();

    // go through the entire list of neighbors
    while (p != vertices[v].edgeList.end())
    {
        perform intermediate processing on v if necessary;
        w = *p; // the neighbor of v pointed to by p;
        process the edge from v to w if necessary;
        p++; // advance p by one node in the edge-list of v;

        // call the function recursively from any non-visited neighbor
        if (!mark[w]) { // w has not been reached
            recursiveDepthFirstSearch (w); // RECURSIVE CALL
    }

    perform final processing of vertex v;
    return from the call;
} // Graph::recursiveDepthFirstSearch()
```

Exercises

12-25 Consider the digraph whose adjacency structure is shown in Exercise 20G. If we use this adjacency structure to conduct a breadth first search, starting at vertex 0 , in what order will the vertices be reached? Will the search mark all of them? In what order will the vertices be reached if we make a depth first search, starting at vertex 0 ? For the depth first search, show the stack that you used to keep track of where you were in the graph during the search.

12-27 Create an adjacency structure for the graph shown below. Suppose a breadth first search is made on this graph, starting at vertex 0 . In what order will the vertices be reached? In what order will the vertices be reached if we make a depth first search, starting at 0 ?



12-29 The algorithm given on page 17 can be modified so as to make it more efficient. The reason the less efficient version was given is that it fairly faithfully mimics the way the recursive version of depth first search works. Here is the more efficient version.

```
Graph::efficientDepthFirstSearch (graph G, int v) // v is a vertex number
{
    create an empty stack S;
    mark vertex v as "reached";
    perform preliminary processing on v;
    push (v, S);
    while (S is not empty)
    {
        x = pop(S);
        make an iterator p point to the edgeList of x;
        while (p != end of the list)
        {
            let y be the neighbor of x pointed to by p;
            process the edge from x to y;
            if (y has not been reached)
            {
                mark y and perform preliminary processing on y;
                push (y, S);
            }
            advance p along the edgeList of x;
        }
        perform final processing on x;
    }
} // Graph::efficientDepthFirstSearch()
```

Work part way through the example give on page 23, using this more efficient depth first search. Does it mark the nodes in the same order? Are there other differences?

Numbering the Connected Components of an Undirected Graph

Suppose we are given an adjacency structure for an undirected graph G and asked to determine whether the graph is connected. This is an easy problem to solve. We simply make a breadth first search of the graph, starting at any vertex, and see whether this marks all the vertices. If it does, the graph is connected. If it doesn't, the graph is not connected. A depth first search will work just well: either type of search will mark all the vertices that can be reached from the vertex you started with.

Now let's make the problem more interesting. When an undirected graph is NOT connected, then it has more than one *connected component*. Is there an algorithm that "sorts" the vertices into their respective components? More precisely, can we find an algorithm that "sorts" the numerical values to the vertices such that all vertices having the same assigned number belong to the same connected component?

This is actually not very hard at all. Here is some pseudo-code to do the job. It assumes that in the adjacency structure for a graph G, each cell of the vertex array contains a field (i.e., member) named `compNumber`, as well as a pointer to an `edgeList`.

```
Graph::numberTheComponents()
{
    // give all "compNumbers" in array
    for (int v = 0; v < numVertices; ++v)
        vertices[v].compNumber = 0; // a sentinel value: " not visited"

    int compCount = 0;

    for (v = 0; v < numVertices; ++v)
        if (vertices[v].compNumber == 0) // v has not been visited
        {
            ++compCount;  //move to a new component number;

            // perform a breadth first search and use the value of
            // compCount to mark the vertices reached.

            breadthFirstSearch(G, v, compCount);
        }
} // Graph::numberTheComponents()
```

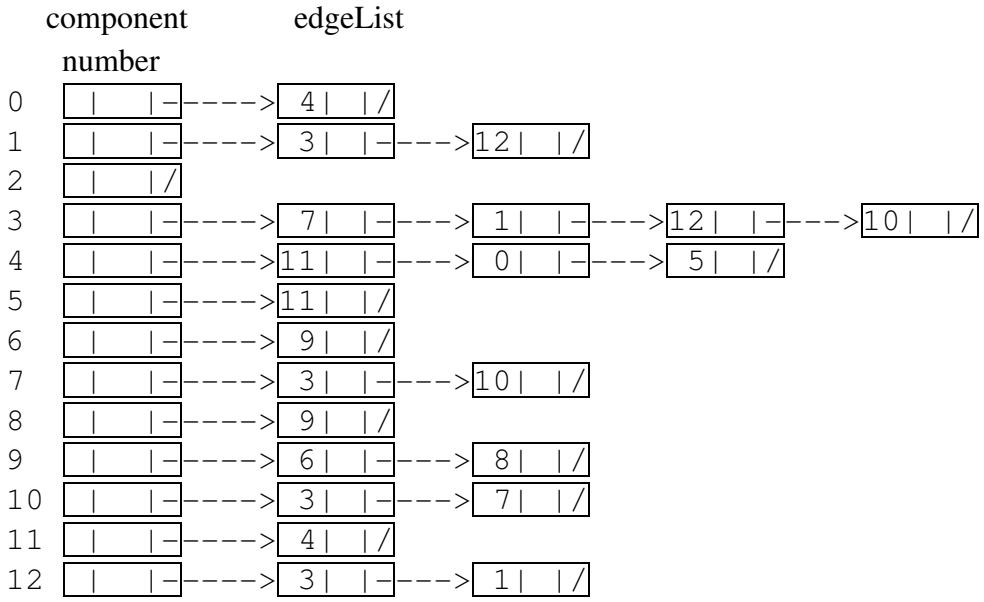
Although breadth first searches are specified in the algorithm above, depth first searches work just as well.

In the breadthFirstSearch function above, the change to the function shown on page 16 is that the part “perform preliminary processing on v” translates as

```
    vertices[v].compNumber = compCount;
```

Exercise

12-33 Below is the adjacency structure for an undirected graph. *Without actually drawing the graph*, perform the algorithm given on the preceding page and determine how many components the graph has and which vertices belong together in which components.



Shortest Path Problems

Let's begin with an easy version of the classical shortest path problem. Suppose we are given a finite graph G (directed or undirected) and two vertices, one of which is specified as an "origin" and the other of which is specified as a "destination". How can we find a path in G from the origin to the destination with the *fewest possible edges*? Of course, if it turns out that there is no path at all in G from origin to destination, we'll want to discover that also.

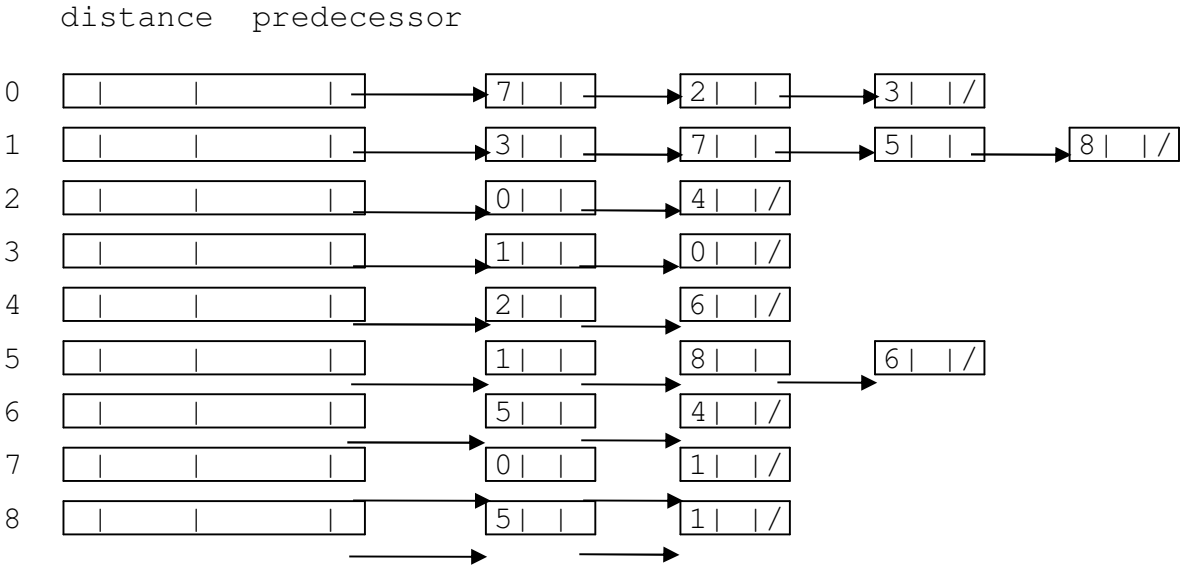
While it would be possible to solve this problem by repeated depth first searches in G starting from the origin, that is not likely to be an efficient way to find the shortest path. A little thought about how a breadth first search works will convince you that you are likely to find a shortest path more quickly by using a breadth first search, which first "marks" all the vertices separated by 1 edge from the origin, then all vertices separated by 2 edges from the origin, and so on.

```
Graph::shortestPath (int origin, int destination)
{
    for (int i = 0; i < numVertices; ++i)
        vertices[i].distance = -1;  // A sentinel value to show vertex i
                                    // has not been reached during search
    queue<int> Q;
    vertices[origin].distance = 0;
    Q.enqueue (origin);
    while (Q is not empty)
    {
        x = Q.dequeue();
        for each vertex y that can be reached from x  //in edge list
            if (vertices[y].distance == -1) // y has not yet been reached
            {
                vertices[y].distance = 1 + vertices[x].distance;
                vertices[y].predecessor = x;
                if (y == destination) // we can quit searching and
                    return;             // EXIT from the routine
                Q.enqueue (y);
            }
    }
} // Graph::shortestPath()

// Now print out the vertices along a shortest path (backward) from
// destination to origin.
if (vertices[destination].distance != -1) // if destination was reached
{
    int i = destination;
    do
    {
        cout << i << endl;
        i = vertices[i].predecessor;
    }
    while (i != origin);
}
```

When setting up the adjacency structure for the graph in preparation for solving a shortest edge-distance problem, each cell in the vertex array should have a "distance" member and also a "predecessor" member. The "distance" member for each vertex v is used to store the number of edges on a shortest path from the origin to v , and the "predecessor" member for each vertex v will be used to store the vertex from which v can be reached on a shortest path from the origin. On the following page is some pseudo-code for an efficient "fewest edges" algorithm. It assumes that in the adjacency structure representing the graph, the array that's indexed by the vertex sequence numbers and that contains the pointers to the edges has been given the name "vertex".

If we are willing to expend some extra effort (say by putting the vertex numbers on a stack instead of printing them), then we can print the vertices in order from origin to destination, which might be considered preferable, especially if the graph is directed.

Here is an example. The adjacency structure shown below represents an undirected graph (the same one that was shown on page 14). Let's find the shortest path from vertex 6 to vertex 3.



Now let's make the problem more challenging. Suppose now that we are considering a *weighted* graph, that is, a graph G in which each edge has been assigned a non-negative number, called the "cost" of that edge. In many applications, each cost is simply the distance along some road or air route from one place to another. The problem to be considered is this: given a vertex specified as the "origin" and another specified as the "destination", find a path in G from the origin to the destination such that the sum of the costs of the edges along the path is as small as possible; if there is no path in G from the origin to the destination, discover and report this fact.

The best known algorithm for solving this ***Least Costly Path Problem*** was invented in 1959 by the famous Dutch computer scientist Edsger Dijkstra[1]. It employs a modification of the breadth-first search method that we have just examined for unweighted graphs. It is not exactly a breadth-first search, however, because it is a "greedy algorithm": it attempts to follow low cost paths before pursuing higher cost paths. In order to accomplish this, it uses a ***priority queue*** in place of the simple FIFO queue.

Let's review briefly the concept of a priority queue, which can be thought of as an ADT (abstract data type). Each of the objects to be placed in a priority queue, call it PQ, must have some numerical *priority value* associated with it. Objects can be inserted into PQ in any order, but when a "dequeue" operation is performed on PQ , the object that's removed from PQ is one with "highest" priority among the objects in PQ . The word "highest" here requires some interpretation. The object(s) with highest priority may be those with the smallest priority number (consistent with common usage, where we refer to tasks of "first priority", "second priority", and so on), or they may be those with the largest priority number. A ***min-queue*** is a priority queue in which elements with the smallest priority values are considered to have the highest priority; a ***max-queue*** is one in which larger priority values have higher priority. There are various ways to implement a priority queue. The most common method uses the notion of a ***binary heap***: a (logical) binary tree is stored in an array, with high priority items at the top of the tree; each node of the tree has priority at least as high as all the nodes below it in the tree.

Now let's return to the Dijkstra algorithm for finding least cost paths. This algorithm requires us to have extra "bookkeeping" members in the cells of the vertex array of the graph's adjacency structure. Here is a list of these members and a description of how each one is used during the algorithm. Assume that we are discussing the cell representing a particular vertex v .

(1) We need a "**total cost**" member to record the sum of the costs along the best path known "so far" to v from the origin. Of course, when the algorithm begins, there is no "best path known so far", so we'll initialize the total cost member to "plus infinity", which should be a number so large that it exceeds the sum of all the costs in the graph.

(2) We need a "**predecessor**" member to record the vertex from which v can be reached on the best path known so far from the origin. Initially there is nothing sensible to record in this member.

(3) We need a boolean "**finalized**" member to tell us whether the best path *known so fa*r to v from the origin is actually the best possible path. Initially this member will be marked FALSE.

---

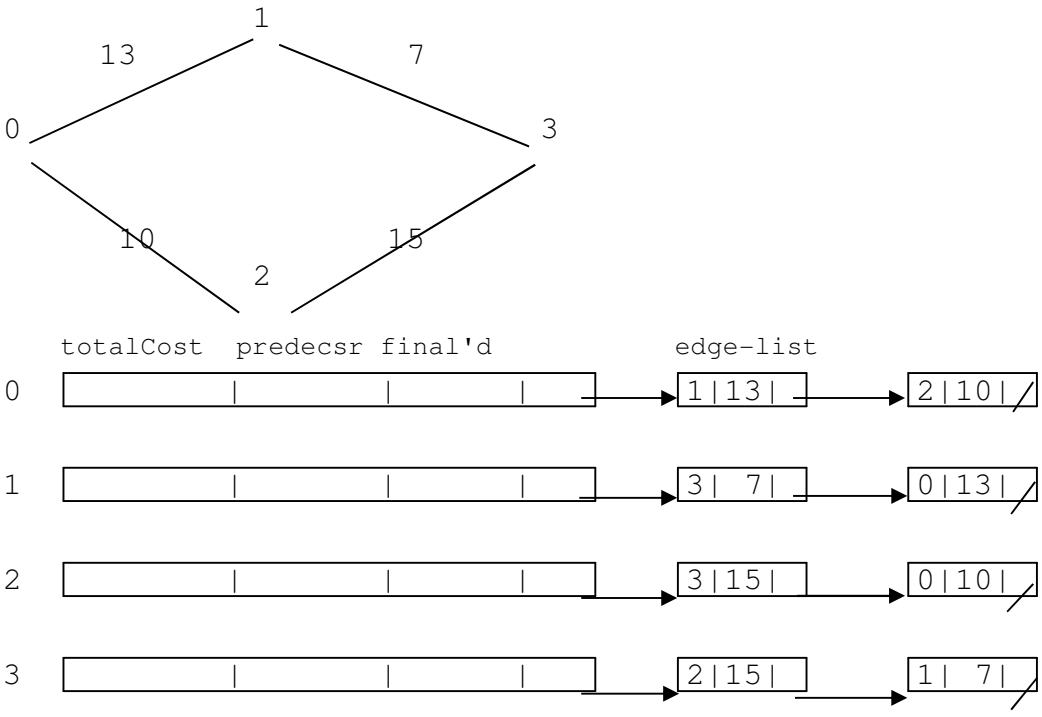
[1] (DIKE-struh) In case you haven't heard of him before, he became an object of a heated controversy regarding GOTO statements. He was advocating that they were harmful and argued for structured programs instead.

The Dijkstra algorithm proceeds by marking a vertex as "finalized" when a least cost path to it is known, and then "updating" the distance members of the non-finalized neighbors of the newly finalized vertex. All the updated members are put into the priority queue. Here is pseudo-code for the algorithm. As before, we assume the vertex array in the adjacency structure is named "vertex".

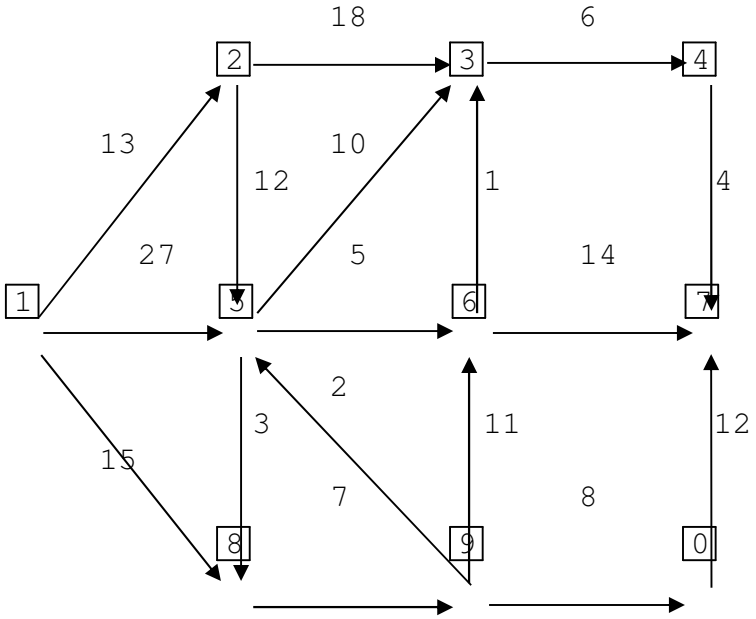
```
Graph::dijkstraAlgorithm (graph G, int origin, int destination)
{
    // initialize the information for each vertex
    for (int i=0; i<numVertex; i++)
    {
        vertex[i].totalCost = +infinity;
        vertex[i].finalized = false;
    }

    create an empty min-queue PQ;     //the objects the min-queue will
           // hold are vertices and their associated totalCost fields
    vertices[origin].totalCost = 0;
    vertices[origin].predecessor = -1; // a sentinel value
    insert the origin and its total cost into PQ;

    // main loop of the algorithm
    while (PQ is not empty)
    {
        remove from PQ into x the vertex with least totalCost;
        if (vertices[x].finalized)
            continue;           // We're seeking non-finalized vertices
        vertices[x].finalized = true; // finalize x
        if (x == destination)
            return;             // EXIT; the algorithm has done its job.

        // now update all the neighbors of x
        for each non-finalized neighbor y of x
            // do we have a better path to y than the one we know?
            if (vertices[x].totalCost + cost(x,y) < vertices[y].totalCost)
            {
                vertices[y].totalCost = vertices[x].totalCost + cost(x,y);
                vertices[y].predecessor = x;
                insert y and its new totalCost into PQ;
            }
    }  // When the function ends, the calling program can check if
       // destination has been finalized; if so, it can print path.
} // Graph::dijkstraAlgorithm()
```

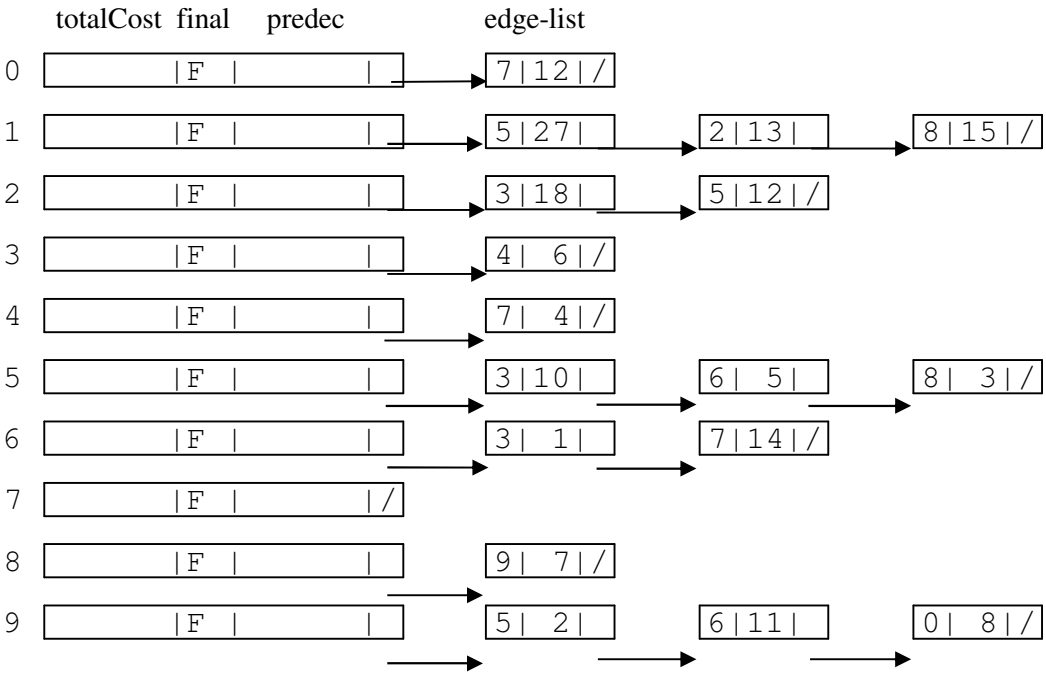
Dijkstra's algorithm often inserts a vertex and its new **`totalCost`** into the priority queue even though that same vertex is already in the priority queue with a larger **`totalCost`**. Don't worry about that. Careful mathematical analysis (not given here) has shown that this duplication is a small price to pay for simplicity in the code.

Let's practice the Dijkstra algorithm on the undirected, weighted graph shown below. Take 0 to be the origin and 3 to be the destination.



Dijkstra's algorithm works equally well on directed and undirected graphs. Let's run the Dijkstra algorithm on the following digraph. Take the origin to be vertex 1 and the destination to be vertex 7 .





We have practiced Dijkstra's algorithm, but we have not discussed why it works. Let's ask this question: is it possible that in some very complicated graph, Dijkstra's algorithm could prematurely mark a vertex as "finalized" and then later fail to spot a less costly path to that vertex? The answer is no, and it is possible to give a perfectly rigorous proof of this assertion by using mathematical induction. Instead of carrying out the details of that induction argument, however, we're going to look at the first few steps of Dijkstra's algorithm and convince ourselves that the finalizations are being done correctly and will be done correctly right to the end of the algorithm.

Since the least costly path from the origin to itself is the path of length 0, it certainly makes sense to initialize that vertex to "finalized" and to give it the "total cost" value 0. Let x_0 denote the origin in the discussion that follows.

After the initialization of x_0, we adjust the "total cost" members of all the neighbors of x_0, we make x_0 the predecessor of each of those neighbors, and we place all those neighbors and their total cost values in the priority queue. The predecessor member of each neighbor vertex w tells us we have a path from x_0 to w for which the sum of costs along that path is the number in the "totalCost" member.

Now we remove from the priority queue some non-finalized vertex, call it x_1, that has the smallest "total cost" member in the queue. We are going to finalize it because -- we claim -- there is no path from x_0 to x_1 that has a smaller sum of costs than the path we have recorded in the predecessor members. How can we prove this?

Suppose we look at any *other* path, call it P, from x_0 to x_1. That path P must start at x_0 and go to some other neighbor vertex $w \neq x_1$, and then go on somehow to x_1. But the edge from x_0 to w already has total cost at least as great as the total cost of the edge from x_0 to x_1, for otherwise we would have dequeued w, not x_1, from the priority queue. Thus the very first edge of path P has a cost at least equal to the cost of the finalized path from x_0 to x_1, and then the remaining edges along P would give a *greater* total cost. Thus we have definitely recorded the least costly path from x_0 to x_1 in the predecessor member of x_1.

After finalizing x_1, we adjust the "total cost" members of all its non-finalized neighbors, where appropriate, and we make x_1 the predecessor of each neighbor whose "total cost" was adjusted. We also put each such neighbor of x_1 into the priority queue, along with its adjusted total cost value.

Then we dequeue some non-finalized vertex, call it x_2, with smallest total cost value. We are going to finalize it because -- we claim -- there is no other path from x_0 to x_2 that has smaller sum of costs than the one we have recorded in the predecessor members of x_2 and its predecessors. How can we prove this? Let's look at the two possibilities for the predecessor of x_2.

Case 1: Suppose the predecessor of x_2 is x_0.
Now examine any path P from x_0 to x_2 other than the one consisting of the edge from x_0 to x_2.

If this path P consists of the vertices x_0, x_1, x_2 in that order, then P must have total cost at least as great as the edge from x_0 to x_2, for otherwise we would have adjusted the predecessor and total cost members of x_2 at the time that x_1 was finalized.

The only other possibility for P under Case 1 is that it has the form x_0, y, . . . , x_2 for some vertex y other than x_1 or x_2. In this case the cost of the edge from x_0 to y must be at least as large as the cost of the edge from x_0 to x_2, for otherwise the dequeuing operation would have given us y instead of x_2. When the cost of the remaining edges on P are added to the cost of this edge, the total cost of P is greater than the total cost of the path from x_0 to x_2 that we have recorded in the predecessor member of x_2.

Case 2: Suppose the predecessor of x_2 is x_1.

Now examine any path P from x_0 to x_2 other than the one consisting of the vertices x_0, x_1, x_2.

If this path P consists of the edge from x_0 to x_2, then that edge must have total cost at least as great as the path x_0, x_1, x_2, for otherwise we would *not* have adjusted the predecessor and total cost members of x_2 at the time that x_1 was finalized.

The only other possibility for P under Case 2 is that it has the form x_0, y, . . . , x_2 for some vertex y other than x_1 or x_2. In this case the cost of the edge from x_0 to y must be at least as large as the cost of the path x_0, x_1, x_2, for otherwise the dequeuing operation would have given us y instead of x_2. When the cost of the remaining edges on P are added to the cost of this edge, the total cost of P is greater than the total cost of the path x_0, x_1, x_2, that's recorded in the predecessor members of x_2 and x_1.

From this point on, the argument is similar, but more complicated. At each step, when we are about to finalize a vertex v, we know that the path we have found and recorded from x_0 to v has the least possible cost because any other path P from x_0 to v must either pass through finalized vertices only or else must pass though some non-finalized vertex, and each of these possibilities leads to the conclusion that P has total cost at least as great as the path we have already found. If you are not yet convinced, look at the case where x_0, x_1, and x_2 have been finalized and a new non-finalized vertex x_3 has been dequeued from the priority queue.

Let O and D be the origin and destination vertices respectively. Here is an informal but complete proof of the fact that Dijkstra's algorithm finds the shortest path from O to D.

The following statement is an invariant for the *while* loop:

> The value of `totalCost` for the vertices `x` marked as finalized represents the least costly path from O to `x`.

If we prove this invariant to be true, then we can remark that if the algorithm ends with the vertex x being equal to D, then D will be finalized, and thus the path marked by the predecessors from O to D will be the least costly path in the graph, which is what we want to prove.

Proof of the invariant:

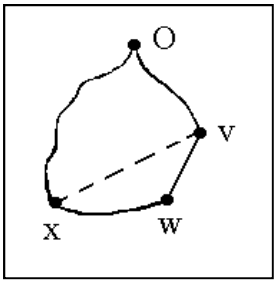
- In the beginning, origin is marked as finalized with a total cost of 0, which is the weight of the least costly path from O to O.

- After one execution of the body of the while loop, no vertex is marked as finalized except for O, and the queue will contain all the neighbors of O (y) in order of the weight attached to the edge from O to y.
- The next time, x will be the neighbor of O with the edge of minimal weight from O to x.
- This represents the least costly path from O to x, because any other path should contain an edge of higher weight.

- The proof of the induction step uses the following additional loop invariants:
- For every finalized vertex, its neighbors are in PQ.
- For every vertex in PQ, the predecessor is already finalized.

- And the following property:
- If O x_1 x_2 ... x_n is the least costly path from O to x_n, then O x_1 ... x_i is the least costly path from O to x_i, $\forall i, 1 \leq i \leq n-1$.

- We will use the proof by absurd.
- Let's suppose the invariant is true when we test that PQ is empty. Suppose that we execute the body of the while loop again. Let x be the next vertex that we dequeue.
- Suppose by absurd that there is another path from O to x that costs less that the totalCost for x. Let v be the finalized vertex in the second path that is the closest to x.



- If there is an edge from v to x, then totalCost[x] should be equal to the cost on the second path.

- If there is no edge from v to x, let w be the next vertex in the path from v to x.
- w has to be in PQ with the totalCost being equal to the cost of the path from O to v plus the weight of the edge from v to w.
- Since the cost on this path from O to x is lower than the totalCost[x], we must deduce that totalCost[w] < totalCost[x]
  which is absurd, because x should have the lowest totalCost in PQ.

Q.E.D.

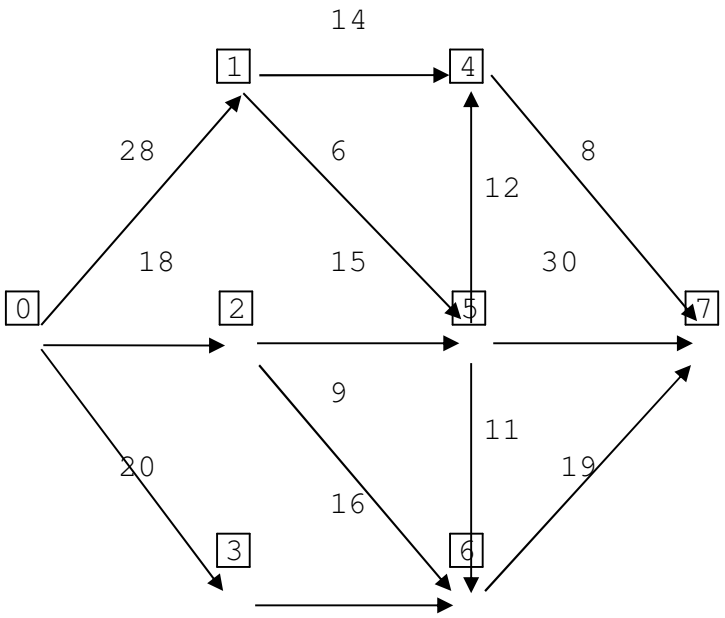
A final comment on terminology. In the foregoing discussion, the problem that Dijkstra's algorithm was invented to solve has been referred to as the "Least Costly Path Problem". In many places, however, it is referred to by the name "Shortest Path Problem", which, in these notes, has been used only in reference to the problem of finding a paths with the fewest possible edge. It is important, however, to be aware that in the wider world the phrase "Shortest Path Problem" is often used in place of "Least Costly Path Problem".

Exercises

12-35 Using the graph shown in Exercise 27G, page 19, run the simple algorithm for finding a shortest path (path with fewest edges) from vertex 8 to vertex 5. Be sure to set up the adjacency structure that a computer program would work with, and work from that structure, not from the picture of the graph. When the algorithm is complete, list the vertices "backward" along the path from the destination to the origin.

12-37 What happens when we run the shortest path algorithm on the graph in Exercise 32G, page 22, using 0 as origin and 1 as destination?

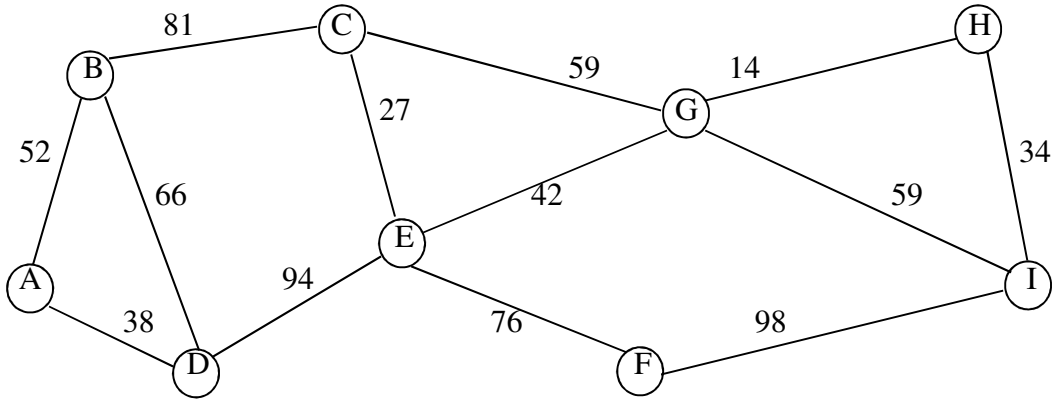
12-39 Run Dijkstra's least costly path algorithm (shortest path algorithm) on the weighted digraph shown below using 0 as origin and 7 as destination.



12-41 Dijkstra's algorithm stops when it finalizes the vertex that it has been told is the destination. What minor modification could we make in the pseudo-code for Dijkstra's algorithm to make it find, for *every* vertex v in G, a least costly path to v from the origin. In other words, when the modified algorithm stops, every vertex reachable from the origin should have been labeled with its distance along a least costly path from the origin, and it should be possible to read off (backward) the shortest path from the origin to any specified vertex.

Minimum Spanning Trees

Suppose a cable TV company is contemplating providing service to a rural area consisting of some widely separated residences. Each residence must be connected to the "grid" in some way. Connections between some pairs residences are possible, while such connections are impossible for other pairs of residences. An installation cost for each connection that's possible can be calculated. If we were given the data about which connections are possible and how much they cost, how could we calculate the least expensive way of providing the desired service? For example, suppose the residences and the *possible* connections, together with their costs, look like this:

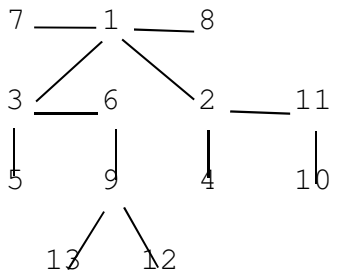


What we would be seeking in the graph above is a *subgraph* with two properties: the subgraph must be connected, and the sum of the weights of the edges in the subgraph must be as small as possible. A little thought suggests that the minimum cost subgraph will contain no cycles, because any connected subgraph that contains a cycle can be reduced to a connected subgraph with smaller total cost by removing one edge (but no vertices) from the cycle. (This last fact requires proof, which will be provided.)

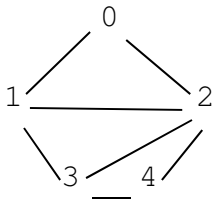
The problem we have just considered arises in many other contexts as well, and we can state it simply as a problem about graphs with weighted edges. Before doing so, it will be convenient to have available some additional vocabulary in which to talk about the problem.

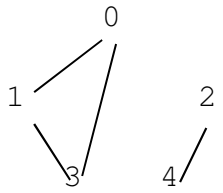
Definition: A ***free tree*** (or ***non-rooted tree***) is an undirected, connected, acyclic graph. A ***spanning tree*** for an undirected graph G is a subgraph T of G with these two properties: (1) T has the same vertex set as G , and (2) T is a free tree.

The term "free tree" comes from the fact that if we take any undirected, connected acyclic graph and choose any vertex arbitrarily ("freely"), we can "hang" the graph from that vertex and make it look like a (not necessarily binary) tree. Here is an example of a free tree. What does it look like if we hang the graph from vertex 3 ? If we hang it from vertex 2?



Below are two undirected graphs. Does either of them have any spanning trees?





We are ready now to state the graph theoretic problem suggested by the example involving the TV cable company.

The Minimum Spanning Tree Problem: Let G be an undirected graph with weighted edges. Find a spanning tree for G having the least possible sum of edge weights. Such a tree is called a ***minimum*** **(**or ***minimal***) ***spanning tree*** for G .

Before attacking this problem, we're going to look at some theorems about connectivity and free trees and spanning trees. These will help us think more clearly about what it is we are looking for when seeking a minimum spanning tree.

Theorem 1: If a graph (directed or undirected) contains a path from a vertex x to a different vertex y , then the graph contains a simple path from x to y .

Proof. Suppose a path from x to y is not simple and (thus) has a vertex list of the form

x, . . . , v, . . . , v, . . . y .

Then we can remove the vertices following the first occurrence of v up to and including the last occurrence of v and still have a path from x to y . If this path is not simple, we can remove more vertices from its vertex list. Continuing in this way we will eventually obtain a path from x to y with no repetition of vertices in the vertex list, because the original list is of finite length.

Theorem 2: If a graph G (directed or undirected) is acyclic, then all subgraphs of G are acyclic.

Proof. The contrapositive of this "if ... then ..." statement says the following:

If there exists a subgraph of G that contains a cycle, then G contains a cycle.

This is obviously true, so the statement of the theorem must also be true.

Theorem 3: If an undirected graph is connected and contains a cycle, then the subgraph obtained by removing any edge (but no vertex) from the cycle is connected.

Proof: Let G be a graph that's connected and contains a cycle. Let $\{a, b\}$ be any edge in that cycle, and consider the subgraph H obtained by removing this edge from the edge set of G (but keeping the same vertex set). It is easy to see (and tedious to prove by induction) that there is still a path between a and b in the subgraph H. To prove that H is connected, take any vertices x and y of H. We must prove that there is a path in H from x to y. Since G is connected, there is a path in G from x to y. If this path does not contain the edge $\{a, b\}$, then it is a path in H. If, however, this path does contain the edge $\{a, b\}$, then we can suppose (without loss of generality) that it appears in the path vertex list this way: $x, \ldots, a, b, \ldots, y$. If we insert between a and b the vertices along a remaining path in H from a to b, then we'll have a path from x to y in H.

Theorem 4: If an undirected graph G is connected and contains two vertices a and b but does not contain the edge between them, then the graph H obtained by adding the edge $\{a, b\}$ to the edgeset of G is connected and contains a cycle, one of whose edges is $\{a, b\}$.

Proof. The connectedness of H is obvious: since any two vertices of G can already be connected by edges in G, the two vertices can be connected by the same edges in H. To see that there is a cycle in G, observe that a could already be connected to b by a path in G. Adding the undirected edge $\{a, b\} = \{b, a\}$ to the end of that path gives a cycle that starts and finishes at a.

Theorem 5: Every undirected connected graph has at least one spanning tree.

Proof. Take any undirected connected graph G. If G is acyclic, then it contains a spanning tree, namely itself. If G contains a cycle, then by Theorem 3 we can remove an edge from the cycle to obtain a *connected* subgraph with the same vertices. If this subgraph contains a cycle, then we can remove another edge to obtain a connected subgraph with the same vertices. We can repeat this process only finitely many times because G has finitely many edges. Thus we eventually obtain a connected subgraph with the same vertices but no cycles. This subgraph is a spanning tree for G.

Theorem 6: Every free tree with exactly n vertices has exactly $n - 1$ edges.

Proof: We'll prove this by induction. The statement in the theorem is clearly true when $n = 1$. Now take any k such that the theorem statement is true for all integers n less than or equal to k. Is the theorem statement true when $n = k+1$? To prove that it is, take any free tree T with $k+1$ vertices. We must prove that T has k edges. Choose any vertex v of T. Let $w_1, \ldots, w_r$ denote the vertices to which v is connected in T (it is connected to at least one other vertex because T is connected and has at least 2 vertices). Remove from T the vertex v and all the edges out of v to produce a subgraph T^*. Let $T_1, \ldots, T_r$ denote the connected components of T^* containing the vertices $w_1, \ldots, w_r$, respectively. There are three things to observe about these components.

(1) Every vertex in T^* belongs to one of these components. To see this, take any vertex w in T^*. Then w is connected to v by some *simple* path in T (see Theorem 1, page 38), and the next-to-last vertex on that path is one of the w_i's , so w is in the connected component of that w_i .

(2) The components $T_1 , \ldots , T_r$ are disjoint from each other. To see this, suppose w_i and w_j were different but belonged to the same connected component of T^*; then the free tree T would contain a cycle consisting of the path from w_i to w_j in T_i and the edges $\{w_j, v\}$ and $\{v, w_i\}$.

(3) Each of the components T_i is a free tree, because it is connected (by its very definition) and is acyclic because T is acyclic (see Theorem 2).

Let n_i denote the number of vertices in the free tree T_i . Then by the inductive hypothesis, T_i contains exactly $n_i - 1$ edges. Also notice that by observation (2) above, the number of vertices in T^* is $n_1 + \ldots + n_r$. Since T^* contains one less vertex than T , it follows that $n_1 + \ldots + n_r = k$. Now we can write the following equations, which prove what we said we'd prove:

number of edges in T = (number of edges in all the T_i's) + (number of edges from v to the w_i's)

$= [(n_1 - 1) \ldots + (n_r - 1)] + r$

$= (n_1 + \ldots + n_r) - r + r = k .$

Corollary 7: Every undirected connected graph with n vertices has at least $n - 1$ edges.

Proof. You will be asked to prove this in an exercise.

Theorem 8: If a connected undirected graph with exactly n vertices has exactly $n - 1$ edges, then it must be acyclic (and thus it must be a free tree, since it's connected and acyclic).

Proof: You will be asked to prove this in one of the exercises.

Theorem 9: Let G be an acyclic undirected graph with exactly n vertices. Let k denote the number of connected components of G . Then G has exactly _____ edges.

Proof: Let $G_1 , \ldots , G_k$ denote the connected components of G . Since G is acyclic, we know by Theorem 3 that each G_i is acyclic. Since each G_i is connected (by definition), it follows that each G_i is a free tree. Let n_i denote the number of elements of G_i . Then $n_1 + \ldots + n_k = n$. By Theorem 6, the number of edges in G_i is $n_i - 1$. Since the edges in the connected components make up all the edges of G , the number of edges in G is

$$(n_1 - 1) + \ldots + (n_k - 1) = (n_1 + \ldots + n_k) - k = n - k$$

Theorem10: If an acyclic undirected graph with exactly n vertices has exactly $n - 1$ edges, then it must be connected (and thus it must be a free tree, since it's connected and acyclic).

Proof. Let G be any acyclic undirected graph with exactly n vertices and exactly $n - 1$ edges. By Theorem 9 the number of edges in G is $n - k$, where k is the number of connected components of G . Since G has $n - 1$ edges, it follows that $k = 1$. Thus there is only one connected component in G , which means that G is connected.

We can summarize the preceding three theorems (6, 8, and 10) in one corollary.

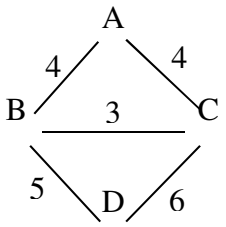
Corollary 11: Let G be an undirected graph with exactly n vertices.

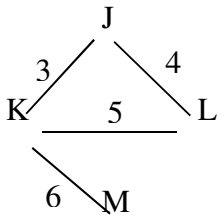
(a) If G is connected and acyclic, then G has exactly $n - 1$ edges.

(b) If G is connected and has exactly $n - 1$ edges, then G is acyclic.

(c) If G is acyclic and has exactly $n - 1$ edges, then G is connected.

Now let's begin looking at connected, undirected, *weighted* graphs and see if we can find *minimum* spanning trees in such graphs. Here are some examples.





Theorem 12: Every connected, undirected, weighted graph contains at least one minimum spanning tree.

Proof: Take any connected, undirected, weighted graph G. By Theorem 5, G contains at least one spanning tree. Since there are only finitely many edges in G, there are only finitely many possible spanning trees. By looking at the sum of the weights of the edges in each spanning tree, we can select a tree of least total edge weight. This is a minimum spanning tree.

Theorem 12 guarantees the existence of a minimum spanning tree in a connected, undirected, weighted graph, but it doesn't give us an efficient method for finding one. In general, the task of finding all the possible spanning trees in a connected undirected graph G is Herculean. We need something less exhausting. Our next theorem will give us a clue for a very efficient algorithm for finding minimum spanning trees.

Theorem 13: Let G be a connected, undirected, weighted graph, and let e be an edge of minimal weight among all the edges of G. Then there is a minimum spanning tree for G that contains e.

Proof: Take any minimum spanning tree T^* for G (we know that one exists by Theorem 12). If T^* already contains edge e then there is nothing to prove. Suppose, however, that T^* does not contain e. Then we'll construct a different minimum spanning tree that *does* contain e. Begin by adding the edge e to T^* to produce a graph H that's a subgraph of G having the same vertex set as G. By Theorem 4, H is connected and contains a cycle. Choose any edge e^* other than e in the cycle. By Theorem 3, we can remove e^* and still have a connected graph, call it T. Then T has the same number of vertices and edges as T^* (so the same vertices as G). By Theorem 8, T must be a free tree, and thus a spanning tree for G. Now let's look at the sum of the weights of the edges in T. Since e has minimal weight among all edges in G, the cost of e is less than or equal

to the cost of e^*. Since T^* and T are identical except that edge e^* in T^* is replaced by e in T, the sum of the edge weights in T must be less than or equal to the sum of the edge weights in T^*. But we know that T^* is a *minimum* spanning tree for G, so T cannot have a smaller sum of edge weights than T^*, and thus T has the same sum, and must be a minimum spanning tree like T^*. (Incidentally, it follows that e^* has the same weight as e, although we do not need that fact here.)

Theorem 13 suggests that if we want to build a minimum spanning tree in a connected, undirected, weighted graph G, then we can begin by picking out an edge of G having minimal weight, because at least one minimum spanning tree contains the edge we have picked. Then we might consider picking a second edge of minimal weight among the remaining edges and asking whether there is a minimum spanning tree that contains both of the edges we have picked. Indeed, there will be, and if we continue in this way, making sure never to choose an edge that would create a cycle, then we'll eventually have enough edges to make a spanning tree. This idea was first published by J. B. Kruskal, Jr., in 1956.

Kruskal's Minimum Spanning Tree Algorithm (1956): Suppose we are given an undirected, weighted graph $G = (V, E)$ with n vertices. Then the following algorithm will construct a minimum spanning tree $T = (V, E_T)$ for G if G is connected, and if G is not connected, it will detect this fact.

(1) Initialize the graph T so that its vertex set is V but its edgeset E_T is empty. That is, T starts out as the "totally disconnected" graph with vertex set V.

(2) Let C denote the collection of connected components of T (so that initially C consists of n singleton sets).

(3) Repeat the following step until $|E_T| = n - 1$ or until the edgeset E is exhausted:

> Remove a lowest cost edge e from E; if the endpoints of e belong to different connected components of T (i.e., different subsets in C), then add e to the set E_T and form the union of the two components containing the endpoints of e, creating a single component of the collection C.

Claim: If G is connected, then $|E_T|$ will be equal to $n - 1$ when the loop (3) above ends, and (V, E_T) will be a minimum spanning tree; but if G is not connected, then $|E_T|$ will be less than $n - 1$ when loop (3) ends.

The following two theorems prove the two halves of the claim made just above.

Theorem 14: If G is connected, then Kruskal's Algorithm does construct a minimum spanning tree.
Proof. Assume that G is connected and has n vertices. We know already that if we choose an edge e_1 of smallest weight from E there will be a minimal spanning tree for G that contains e_1. Since the endpoints of e_1 will be in different connected components of G initially, the algorithm will

allow us to add e_1 to the set E_T, and to put the two endpoints of e_1 into a single connected component. Now $T = (V, \{e_1\})$, and the number of connected components in the graph T has decreased by 1 to $n - 1$. If $n = 2$, then the algorithm will stop with the correct number of edges to make T a minimum spanning tree.

Suppose $n > 2$. Since G is connected, it has at least 2 edges (see Corollary 7), so we can pick a minimal weight edge e_2 from those that remain in E. At least one of the endpoints of e_2 will be different from the endpoints of e_1, so the endpoints of e_2 will belong to different connected components of T. Obeying Kruskal's algorithm we add e_2 to the set E_T and form the union of the two components of T containing the endpoints of e_2. This decreases the number of connected components of T by 1 to $n - 2$. Now let's prove that G has a minimum spanning tree that contains both e_1 and e_2. We already know that G has a minimum spanning tree, call it T_1 that contains e_1. If e_2 is in T_1, then there is nothing more to prove. If e_2 is not in T_1, then add it to T_1 to create a connected subgraph of G having a cycle containing e_2. Then an argument similar to the one in Theorem 12 will show that we can remove another edge from that cycle to obtain a minimum spanning tree. We need not remove e_1 if that is part of the cycle, because e_1 and e_2 cannot by themselves form a cycle in an undirected graph.

Now $T = (V, \{e_1, e_2\})$. If $n = 3$, then Kruskal's algorithm will stop with the correct number of edges to make T a minimum spanning tree.

If $n > 3$, then by Theorem 9 the current number of connected components of T is $n - 2$, which is greater than 1. Since G is connected there must be at least one edge remaining in E whose endpoints are in different connected components of T. Remove minimal cost edges from E until arriving at such an edge (its endpoints are in different components of T), and call this edge e_3. Add this edge to the set E_T and form the union of the two connected components of T containing the endpoints. This reduces the number of connected components of T by 1 to $n - 3$. Now let's prove that G has a minimum spanning tree that contains e_1, e_2, and e_3. We've proved already that G has a minimum spanning tree, call it T_2 that contains e_1 and e_2. If e_3 is in T_2, then there is nothing more to prove. If e_3 is not in T_2, then add it to T_2 to create a connected subgraph of G having a cycle containing e_3. Then we can remove another edge from that cycle to obtain a minimum spanning tree containing e_3. We need not remove e_1 or e_2 if they are part of the cycle, for if the cycle consisted of just e_1, e_2, and e_3, then the endpoints of e_1 and e_2 would have belonged to the same connected component of T before we added e_3, and e_3 would have had both its endpoints in this connected component (which we were careful to avoid).

Now $T = (V, \{e_1, e_2, e_3\})$. If $n = 4$, then Kruskal's algorithm stops with the correct number of edges to make T a minimum spanning tree. If $n > 4$, we can continue on as in the preceding paragraph. The graph T keeps growing in this way until it becomes a minimum spanning tree.

Theorem 15: Let G be an undirected graph with n vertices. If G is not connected, then Kruskal's algorithm stops with $|E_T| < n - 1$.

Proof. We'll prove the contrapositive: If Kruskal's algorithm stops with $|E_T| \geq n - 1$, then G is connected. Suppose Kruskal's algorithm stops with $|E_T| \geq n - 1$. By the specification of the algorithm, it must then stop with $|E_T| = n - 1$. Since the algorithm never permits T to add an edge that would create a cycle, T is always acyclic, and so when the algorithm stops, T is an acyclic graph with exactly $n - 1$ edges. By Theorem 10, T must be connected, and since it is a subgraph of G , it follows that G is connected.

Kruskal's Algorithm poses two interesting computer implementation problems for us.

(1) How will we make a program keep track of which edges remain in the set E and efficiently locate the one with smallest edge weight when that is required?

(2) How will we make a program keep track of the connected components of the growing graph T in such a way that given any edge of G we can quickly determine whether its endpoints are in the same component?

The answer to the first question is not hard. We put all the edges of G into a priority queue initially; this should be a min-queue because we want each delete operation on the queue to give the edge with *smallest* edge weight among those still in the queue. It is possible to construct the priority queue in such a way that we do not have to copy all the information in the edge nodes of the graph. We can simply put pointers to the edge nodes into the priority queue, and then follow the pointers whenever we want to examine an edge.

There is one extra thing we'll have to store in each edge node of our adjacency structure for the graph G: both vertices that make up the endpoints of the edge must be present in order for a program to be able to "see" the endpoints when we remove an edge (actually a pointer to an edge) from the priority queue.

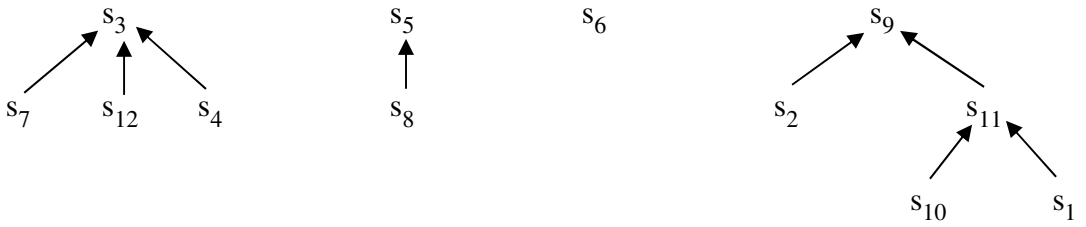
The second question about making a program keep track of connected components of T requires a digression on the problem of forming the unions of disjoint sets and determining which set a given element belongs to. The general problem can be stated this way: suppose a program contains a collection of n data objects of some kind, call them $s_1, s_2, ..., s_n$. Suppose initially we want the program to treat each object as the sole element of a singleton set, so that initially the program is working with n sets $\{s_1\}\ \{s_2\}\ \ldots\ \{s_n\}$. Suppose the program is required to perform a sequence of ***union*** operations on pairs of sets in this collection, so that after a number of such union operations the collection consists of disjoint sets such as

(*) $\{s_7, s_3, s_{12}, s_4\}\quad \{s_5, s_8\}\quad \{s_6\}\quad \{s_2, s_1, s_{11}, s_9, s_{10}\}\ \ldots$

How can the program keep track of these sets, and, for a given object s in the original collection,

how do we answer the question "Which set does s belong to?" ? This is known as the Union/Find Problem in algorithm theory.

The best known solution to this problem is covered in the course C243 Data Structures. First consider the Find Problem: the program is given an object s and asked what set the object belongs to. Recall that we handle this problem by having the program select arbitrarily an object from each set to serve as the "representative object" of that set. Then when the program is asked to "find" the set to which s belongs, it returns the representative object of the set containing s. This can be done by making the other objects in the set point either to the representative object of the set or else to some other object in a chain leading to the representative object. Thus, for example, the sets shown in line (*) above might be represented in the following way (among many possibilities):

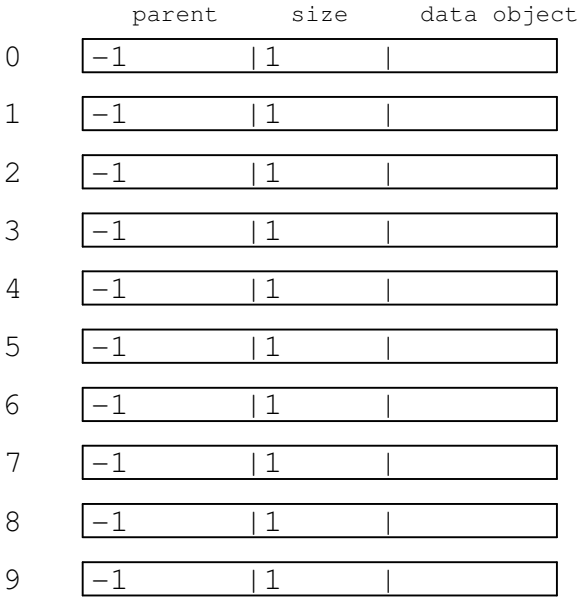


Then when the program is asked to find the set containing some particular s, the program follows a chain of pointers to the representative object of the set, and returns that object.

Now suppose the program is required to form the union of two of these sets (the Union Problem). This is easy to take care of: the program makes the representative object of one of the sets point to the representative object of the other set. To avoid producing long chains during the formation of unions, we practice **balancing**, which requires us to keep a "size" field in the representative object of each set. The size field tells the number of objects in that set. When a union operation is performed on two sets, we make the representative object of the smaller set point to the representative object of the larger set, and the update the "size" field in the representative object of the combined set. In case of a tie, either can be made to point to the other. If we consistently follow this balancing rule for deciding which root points to the other during union operations, then it is a fact that the height of each tree T in the collection is bounded by $\log_2 (n_T)$, where n_T denotes the number of objects in the tree T. You will be asked to prove this fact in Exercise 65G.

You also learned about another technique for shortening the lengths of chains in the disjoint sets. It was called **path compression**, and it is used during Find operations. We'll neglect path compression in what follows, but be aware that path compression can improve the performance of the Kruskal algorithm.

On the next page is an example of a collection of data objects in an array. To implement the union and find operations on disjoints sets of these data objects, we require two extra fields in each cell of the array: a "parent" field for pointing to a parent in the tree of data objects making up a set, and a "size" field for keeping track of the size of the set represented by a data object. We initialize all the "parent" fields to the sentinel value -1, and we initialize all the "size" fields to 1 because initially each data object is considered to be in a singleton set by itself.



Let's see how we could perform the following sequence of operations, in which the data objects are referred to by the indexes of the cells in which they are located.

`Union(3,8)` (That is, form the union of the set containing the data object in cell 3 with the set containing the data object in cell 8.)

`Union(5,2)` `Union(3,4)` `Union(9,6)` `Union(4,9)` `Union(5,0)`
`Find(9)` `Find(5)` `Find(2)`

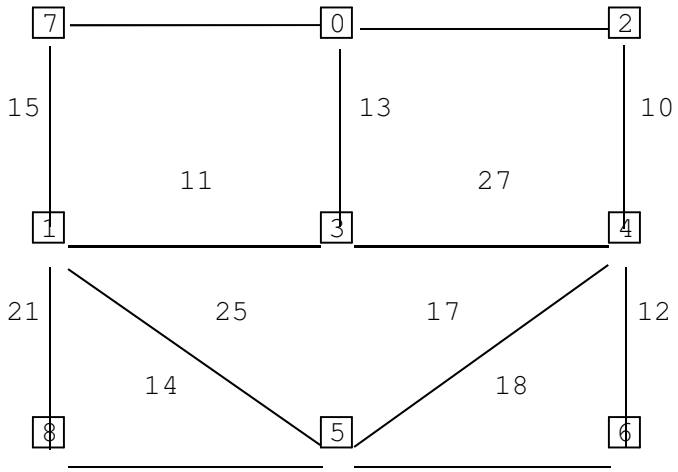
The easiest but not very efficient way to represent the connected components is to use an array, let's call it `component`, indexed by the sequence number of the vertices whose value will represent the connected component that the vertex belongs to. In the beginning, each vertex belongs to a different connected component, so we can initialize this array the following way:

```
for (i=0; i<vertexNr; i++)
    component[i] = i;
```

Suppose that we are adding a new edge {v, w} and that we have to make the union of the connected components of v and w. In this case, we can use the following code:

```
oldComponent = component[w];
for (i=0; i<vertexNr; i++)
    if (component[i] == oldComponent)
        component[i] = component[v];
```

Now let's practice Kruskal's Minimum Spanning Tree Algorithm on the following undirected graph.



Suppose that when we read the graph data from a file we end up with the following adjacency structure.

```
     parent    size     edge-list
0  |-1   |1    |-|--->|7|0|16|-|-->|2|0|19|-|-->|3|0|13|/
1  |-1   |1    |-|--->|5|1|25|-|-->|3|1|11|-|-->|8|1|21|-|-->|7|1|15|/
2  |-1   |1    |-|--->|0|2|19|-|-->|4|2|10|/
3  |-1   |1    |-|--->|1|3|11|-|-->|0|3|13|-|-->|4|3|27|/
4  |-1   |1    |-|--->|2|4|10|-|-->|6|4|12|-|-->|5|4|17|-|-->|3|4|27|/
5  |-1   |1    |-|--->|1|5|25|-|-->|8|5|14|-|-->|6|5|18|-|-->|4|5|17|/
6  |-1   |1    |-|--->|5|6|18|-|-->|4|6|12|/
7  |-1   |1    |-|--->|0|7|16|-|-->|1|7|15|/
8  |-1   |1    |-|--->|5|8|14|-|-->|1|8|21|/
```

To make the algorithm easier to carry out, here is a list of the edges in increasing order by weight:

{2,4} {1,3} {4,6} {0,3} {5,8} {1,7} {0,7} {4,5} {5,6} {0,2} {1,8} {1,5} {3,4} .

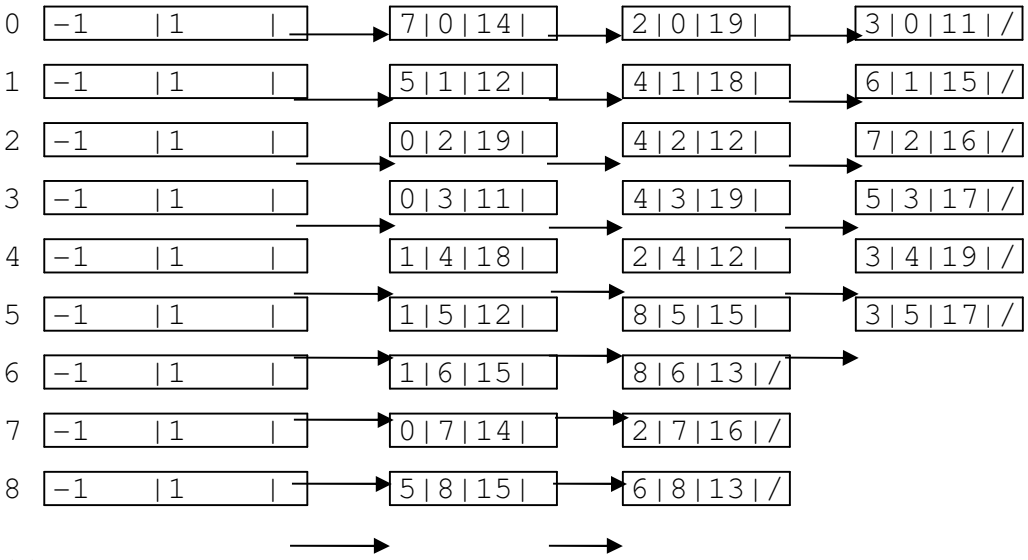
(We would insert these into a priority queue initially if we wanted to do this in the most efficient possible way.)

Exercises

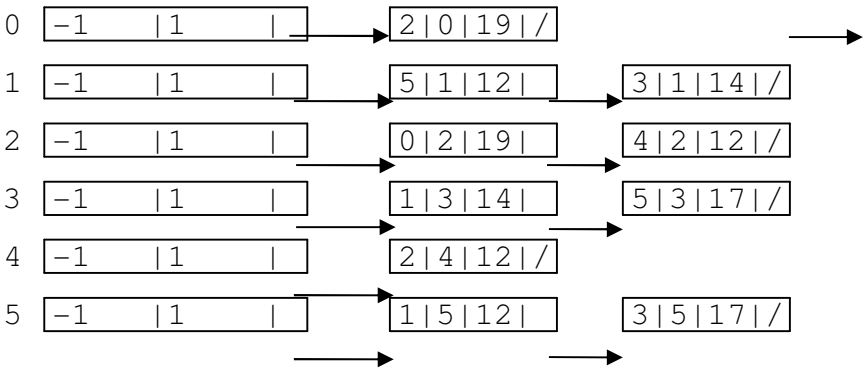
12-51 Suppose a computer program has 15 objects in an array, in locations 0 through 14. Let the objects be referred to by the index of the cell in which they are located. Using the technique described on pages 40 and 41, show how the program could perform the following sequence of operations: Union(2, 8) , Union(14, 5) , Union(3, 11) , Union(10, 2) , Union(8, 3) , Union(6, 7) , Union(12, 14) , Union(9, 14) , Union(3, 5) , Find(11) , Find(9) , Find(4) .

12-53 Using Kruskal's Algorithm, find (if possible) a minimum spanning tree for graphs having the following adjacency structures.

(a)



(b)



12-55 Prove Corollary 7, using theorems that preceded it.

12-57 Prove Theorem 8 by the method of contradiction. Your proof should begin with words like these: "Take any connected undirected graph G having exactly n vertices and exactly $n - 1$ edges. Suppose (for purposes of contradiction) that G contains a cycle." The proof will need to use some of the theorems that precede Theorem 8.

12-59 Prove that every acyclic undirected graph having exactly n vertices has at most $n - 1$ edges. Use the theorems proved in this section.

12-61 Let G be a connected, undirected graph with exactly n vertices. Prove that if G has more than $n - 1$ edges, then G has more than one spanning tree.

12-63 Let G be a connected, undirected, weighted graph. Suppose no two edges of G have the same weight. Prove that there is exactly one minimum spanning tree for G.

12-65 Suppose we begin with a collection of singleton sets, each implemented as a tree with just one node, and we perform a sequence of union operations on these sets, being careful at each stage to follow the rule described at the bottom of page 44: make the root of the tree with the smaller number of nodes point to the root of the tree with the larger number of nodes. Prove that for every tree T produced in this way, the height of T, call it h_T, and the number of nodes in T, call it n_T, satisfy the inequality $2^{h_T} \leq n_T$. (From this it follows that $h_T \leq \log_2 (n_T)$, as stated on page 45.)

HINT: Use weak induction to prove that the following predicate is true for all positive integers m.

S(m): at the m-th stage (after $m - 1$ union operations have been performed),

every tree T in the collection satisfies the inequality $2^{h_T} \leq n_T$.