Data Structures and Algorithms

David Guichard
Whitman College

Contents

1

Introduction

We will be studying ways to **structure** or organize data. Of course the way we organize data depends on what we plan to do with the data. Thus we also will study algorithms for accomplishing certain tasks and the appropriate ways to structure the data involved. Niklaus Wirth (who designed the languages Pascal and Modula) really summed up the entire course in the title of a book, **Algorithms + Data Structures = Programs**.

At the outset we need to make an important distinction between an abstract data structure (also called an abstract data type) and an **implementation** of the data structure. An abstract data structure is a set of objects together with a collection of operations on them. For example, $\mathcal{I} = (\mathbf{Z}, +, -, \times)$ is an abstract data structure: the collection of objects is $\mathbf{Z}$, which mathematicians use to denote the set of all integers; the operations are the familiar arithmetic functions. Notice that the specification of the operations is very important. The abstract data type $(\mathbf{Z}, +, -)$ is much different (and simpler, since addition is in some sense easier than multiplication).

The structure $\mathcal{I}$ is abstract, that is, there is no indication of how the objects may be represented or the operations carried out. The C language provides an implementation of a data type called int; this is an implementation of some abstract type but it is not an implementation of $\mathcal{I}$. C provides an implementation of a 'finite subrange' of the ordinary integers: the set of objects is some set of integers $\{minint, \ldots, maxint\}$, and the operations are essentially the normal integer operations, except that the limits of the underlying set must be taken into account. For example, '$maxint + maxint$' cannot have its usual interpretation because it is too big. An implementation of this abstract type involves

choosing how to represent the integers between *minint* and *maxint* (usually as binary patterns of some fixed length) and how to perform the operations (usually as a sequence of machine language instructions).

A somewhat more complex example of an abstract data structure is the array. In fact arrays must be 'of something' (like 'array of integers' or 'array of real numbers') so formally we are talking about many different data structures, yet abstractly the contents of an array have no bearing on the concept 'array.' (The contents may have an impact on the implementation of arrays, however.) Let's simply refer to the contents of the array as 'cells,' under the assumption that a cell is a member of some other data type.

A specification of an (abstract) array structure, $\mathcal{A}$, might be something like this: the objects of the data structure $\mathcal{A}$ are ordered sets of some fixed number of cells, say n. In other (more mathematical) words, the objects of $\mathcal{A}$ are 'n-tuples' of cells, which are usually denoted $(c_1, c_2, \ldots, c_n)$. Typical operations would be

retrieve(A, i): A function which returns the value of the i^{th} cell in the object A. Thus, retrieve$((c_1, c_2, \ldots, c_n), i)$ returns c_i.

insert(A, i, C): A procedure which puts the value from cell C into the i^{th} cell in A, i.e., sets c_i equal to C.

Note that the operations are specified as if they were actual procedures or functions in some high level language; this is merely a convenient notation. In a given implementation the exact syntax usually varies from the abstract specification; for example retrieve(A, i) in C is `A[i]`, and insert(A, i, C) is `A[i]=C`. The actual syntax of a particular implementation will depend on the tools you have to work with and the specifications, if any, imposed by the designer of the data structure. If you write a compiler for a high level language, for example, you could conceivably use `A(i)` for the retrieve operation. If you are using a high level language to implement a data structure then you will be limited by the syntax of the language, and you will usually be forced to write the operations as actual procedures. In C++, you can actually use existing symbols, like '+', to represent new operations.

As for the abstract type integer, this specification of $\mathcal{A}$ does not say anything about how to implement arrays. Most languages do implement arrays, but we will not be too concerned with exactly how this is done: for us, arrays are as basic as the types int, char and float—we can use them without understanding the implementation details. As we implement more complicated data structures, we will strive for this 'encapsulation' feature as well: if we specify clearly what the objects are and what can be done with them, then it should be possible to use the data structure without understanding the details. C++ encourages this kind of programming by providing **classes**. In other languages, like Pascal, you would conceptually group a number of procedures and functions together, thinking of

them as the operations of the data structure, but the language provides no way to make sure these operations are used.

Besides the obvious convenience of using a tool without needing to understand everything about it, there are other good reasons to use this style of programming, in which implementation details are hidden. It makes writing and debugging programs easier, because the location of an error is easier to discover: all the code that deals with a particular structure is in one place. If necessary, the implementation details of a data structure can be changed, without requiring any change in the programs that use the data structure. Even if you are implementing a data structure for your own use, these features of encapsulated programming will make your job easier.

The solution of a particular problem typically involves the design of an algorithm and appropriate abstract data structures, and then the implementation of the two. These four tasks are mutually dependent on one another. A particularly nice abstract data structure may make implementation easier, and conversely the tools you have to work with in doing the implementation may guide you in the design of the abstract structure. The choice of algorithm and data structures may also be governed by concerns other than simply solving a given problem, and these concerns can be in conflict, for example:

a. By appropriately organizing data you may be able to provide a better solution to a given problem; here 'better' might mean that your program runs faster or uses less space than it might otherwise.

b. Use of the right data structure may make a problem easier to solve by providing a framework which makes the algorithm particularly apparent; notice that the use of a structure for this purpose might well provide poor performance compared to an algorithm and data structures chosen to satisfy (*a*). You should usually choose a structure which makes the problem easy to solve. Once you understand the problem and a method of solution thoroughly it should be easier to rewrite the program in a more efficient way if necessary.

c. The choice of a particular algorithm and collection of data structures may make the solution to the problem particularly easy to understand; this is in a sense a more extreme version of (*b*): if a particular program is to have a long life and be subject to tinkering by many programmers in the future, it may be desirable to tolerate even worse performance so that those programmers do not have to spend large amounts of time learning how the program operates.

2

Lists

We use lists so often and in so many different contexts that it shouldn't be surprising that they are very useful as a data structure. First, a word about terminology: There are two senses in which I will use the word 'list' (and the name of almost every other data structure we talk about). At times I might use the word to refer to a specific data structure, including a specific collection of operations. For the most part, however, I will use 'list' to mean 'list-like', as a generic term for any data structure that behaves like a list.

You certainly know what a list is and hence have some idea of what a list data structure should be; let's make it more precise. Like arrays lists must be 'of something,' but we need not specify what. A list is a list of items which we simply refer to as cells.

Also like arrays, lists are linearly ordered and there is a natural notion of **position** on a list. In fact, what it 'means' to be a list is in some sense completely determined by the positions of the cells on the list. Just based on ordinary (that is, non-computer) experience it is easy to guess some operations on lists that are likely to be useful. Some more-or-less standard list operations are:

first(L)	returns the position of the first cell on the list L
last(L)	returns the position of the last cell on L
next(L,P)	returns the position of the cell following the cell at position P
prev(L,P)	returns the position of the cell preceding the cell at position P
retrieve(L,P)	returns the cell at position P on L
insert(L,C,P)	inserts the value of cell C at position P on list L
remove(L,P)	removes the cell at position P from the list L

If you are thinking of 'position' as an integer, which is a natural thing to do, it may seem that some of these specifications of operations are long winded and obscure ways to say something trivial (for example, 'next(L,P)' would simply be 'P + 1'). In part this is correct, but it is because you *are* thinking of position in a particular way, that is, you are thinking in terms of a specific implementation of lists, rather than the abstract notion of list. In fact, the difference between arrays and lists may not yet be clear, and you may be visualizing a list as an array. Our goal here is to identify the crucial features of lists without locking ourselves in to any particular way of implementing them. 'Position' is really a much more general notion than how far that position is from the front of the list. For example, a dictionary is a list of words (and associated information). You use a dictionary, which certainly involves something like the operations listed above, without ever knowing or caring whether a particular word is the 47,018th word in the dictionary.

What does distinguish a list from an array? A list is usually thought of as having no fixed length: it can grow and shrink as additions or deletions are made anywhere on the list. When an insertion is done the new value does not replace an existing value (which is what an array insertion does). When a new value is inserted at position P, the previous values all remain unchanged on the list but those which appeared at position P or later will now have new positions on the list. Similarly, a deletion removes a value from the list but does not leave a gap: the remaining values assume new positions. (The phrase 'new position' does not necessarily imply that values are actually moved from one location to another; that will depend on the implementation chosen.)

There are some other very useful, even necessary, operations which might not occur to you at first. One is the operation 'create(L),' which defines L as a list with no entries. This is much like taking a blank piece of paper and writing at the top, 'THINGS TO DO,' without actually making an entry on the list; we often want to make the act of creating a list separate from actually putting something on it. In Pascal an array is 'created' when it is declared—something you must do explicitly. We will need some explicit initialization function to create a list; in C++ this will be done by a constructor function. We may also want an operation to erase all entries from a list while leaving the list intact: makenull(L) or makeempty(L); often we include a test function which returns true or false depending on whether the list is empty: isnull(L) or isempty(L).

When solving a particular problem you are free to define 'list' in any way that is suitable for a solution. On the other hand, your entire task may be to set up a list structure for future use by many users. For example, in writing a compiler for a Pascal-like language, you may choose to include lists as a predefined structure. In this case you would probably want to provide many operations to make the structure as flexible as possible. Even when you are defining a structure for use in a single program, it may be wise to define a more

general structure than you need. Future modifications to the program may demand more operations, and it is probably easier to put them in at the beginning.

How you choose to implement lists depends on exactly what operations you are including and on what you demand of the implementation. It may be that you want a 'quick and dirty' implementation of lists, because you are really interested in writing a program that will use the lists. Once your program is up and running, you can then go back and improve your implementation (if necessary) without affecting the program. We will look at three common implementations of lists in detail.

2.1 Naive Array Implementation

An array has a number of list-like properties, so it should not be surprising that a simple way to implement lists is to use arrays with as little modification as possible. Each list will live in an initial segment of its own array: if the list has five elements then it will reside in the first five positions in the array. In this implementation the order information for the list is given by the index type of the array, and the position of a cell is simply the index of the cell in the array. The constant `LNULL` is used as a special flag value for a position in a number of circumstances, when a real position value would be wrong. We also include a field called `lastentry` in the class to indicate the index of the last item on the list. The declarations for this implementation might look like figure 2.1.

```
#define LNULL -1
typedef  int list_position;
class list {
public:
  list();
  void insert (int n, list_position target);
  void remove (list_position target);
  list_position next(list_position p);
  list_position prev(list_position p);
  int retrieve(list_position p);
  list_position first();
  list_position last();
private:
  list_position lastentry;
  int thelist[MAXLNTH];
};
```

Figure 2.1 Array implementation

```
list_position list::next(list_position p) {
  if (++p > lastentry) {
    return LNULL;
  } else {
    return p;
  }
}
```

Figure 2.2 Next function

Now operations like first, last, next, prev, retrieve, isnull and makenull are all quite easy to implement. For example, `next` might look like figure 2.2. We return `LNULL` if the position would be 'off the end' of the list.

Insert and remove present a bit more of a problem. Since we must maintain the list in an initial segment of the array (why?), insertion and deletion (except at the end of the list) will require movement of elements within the array. This is bound to be inefficient on long lists, but it may be acceptable for a number of reasons: It may be that most or all insertions and deletions *will* be at the end of the list; perhaps insertions and deletions will not occur very frequently; perhaps the lists will rarely be very large and the inefficiency will be negligible; perhaps the ease of writing this implementation makes it acceptable for testing purposes, and a more efficient one will be written before the program is put into use. The insert routine might look like figure 2.3.

```
void list::insert (int n, list_position target) {
  for (int i=lastentry; i>target; i--) {
    thelist[i+1] = thelist[i];
  }
  thelist[target+1] = n;
  lastentry++;
}
```

Figure 2.3 Insert function

Notice that the insertion is performed after the indicated position, not before. Either would be acceptable, and inserting before might seem a bit more natural, but inserting after the specified position is somewhat simpler. In particular, note that to insert an item at the beginning of the list, we can specify `target=LNULL`, and to insert at the end target must simply be the last position on the list. You should think about how you would insert at the end of the list if you want to insert before the target position. Similar comments

apply to the remove function—we can remove the item at position `target`, which is more natural, or following it, which is consistent with insertion.

Note also that when moving items out of the way before the insertion, the highest numbered item is moved first: the for loop covers the desired range in decreasing order. It is easy to see that this is necessary to avoid overwriting and hence losing information. When moving information 'up' in the array we must start at the top, and when moving items down we would start with the lowest item to be moved. This simple but crucial algorithm for moving data will come up again; the rule is sometimes summarized as 'Move the leading edge first.'

There are disadvantages to this implementation besides the possible inefficiency. Although an abstract list can grow without bound, we are limited by the declared size of the array used to hold a list. Of course the computer itself is finite, so we cannot hope to implement really unbounded lists. The real problem here is that we must decide in advance how big any one list can be, and we must allot that much space for each list; in practice much of this space will be wasted on small lists. If we don't know how many lists will be required by the program, we will have to overestimate that as well, leading to even more waste since some lists will remain empty. Nevertheless, these considerations may in some applications turn out to be unimportant or outweighed by the advantages of the implementation. Some languages, including C, make it relatively simple to increase the length of an array or replace an array by a longer one; other languages make this more difficult. Even when possible, this is is not as good a solution as the next implementation.

2.2 Cursor Implementation

There is another way to design an array-based implementation for lists of cells which is much more flexible than the naive approach we have seen. In the simplest version, there is a single large array, sometimes called `node_space`, in which all lists reside. The linear order of a particular list will *not* be given by the index of the array; instead *each cell* will include information about what follows it, so that the next cell on a list need not be in the next position in the array.

Instead of a single array, we will use an array called `cell_space` to store the actual data items, and a second array to record the index at which we can find the *next* item on the list; these two arrays together will be our `node_space`, and the combined information in a cell and its associated link is called a node. Here is a picture of a very small `node_space` containing four lists starting at positions 0, 1, 4 and 8 in the array.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
cell	L	Z	I	R	X	T	A	C	Y	D	M	N	H	G	B
next	10	7	13	2	3	−1	14	9	6	−1	11	−1	5	12	−1

The *next* value associated with a cell contains the location of the cell that follows it on its list; a value of `LNULL`, which is not a legal array index, is used to signal the end of a list. The value in the *next* field is known generically as a **link** or **pointer** (in the case at hand, i.e. when the pointer value is used as an array index, it is commonly called a **cursor**, and any implementation of lists which uses some sort of pointer is called a linked list). The pointer value `LNULL` is usually called the **nil** pointer. The cells may be of any type; here the cells are simply characters. The list starting at position 4 in the array is thus: X, R, I, G, H, T. The C++ declarations to set up a cursor implementation to store lists of integers might look like figure 2.4.

This definition of `list` is superficially quite similar to the previous one, but really quite different. In the naive implementation, each list 'contains' all of its own elements, in an array. In contrast, the `static` designation on the private variable `space` in the cursor implementation means that there is just one array shared by the entire class. Every object of class `list` refers to the same array; only the variables `start` and `lastentry` are 'in' the list object. We also need a destructor function in this class; when a list is destroyed, we want to recycle the nodes it used.

To insert a new value on a list, we now need to request space in which to store it; this is the job of the `get_node` function. Likewise, when we remove an item from a list, we can recycle the space using `return_node` so that it can be reused if needed. To manage the available space, we keep all unused nodes linked together in a *free list*; `free` points to the first available node on this free list. The constructor for `node_space` must initially link all the nodes into one big free list. The `get_node` function would look something like figure 2.5.

The primary advantages of this implementation are flexibility and the efficiency of the insert and remove routines. Because all nodes live in a community node space, both the length of any one list and the number of different lists possible are bounded only by the size of node space. The same space can be used for many small lists, a few large ones or some mixture of the two; small lists use only as much space as they need, and a new list may be started at any unused node. In many languages, including C and C++, we also may choose to replace the arrays in `node_space` if they fill up, though this could be quite

```
typedef  int list_position;
typedef  int list_item;

class node_space {
public:
  node_space();
  list_position get_node ();
  void return_node(list_position);
  list_position get_next(list_position);
  list_item get_val (list_position);
  void set_val (list_item, list_position);
  void set_next (list_position, list_position);
private:
  int free;
  list_item cell_space[MAXLNTH];
  list_position next_space[MAXLNTH];
};

class list {
public:
  list();
  ~list();
  list_position first();
  list_position last();
  list_position next(list_position p);
  list_position prev(list_position p);
  void insert (list_item n, list_position target);
  void remove (list_position target);
  list_item retrieve(list_position p);
private:
  list_position start;
  list_position lastentry;
  static node_space space;
};
```

Figure 2.4 Cursor implementation

```
list_position node_space::get_node () {
  list_position p = free;
  free = next_space[free];
  next_space[p] = LNULL;
  return p;
}
```

Figure 2.5 Cursor `get_node`

costly for large arrays, since the entire contents of the arrays must be copied to new, larger arrays.

Of the standard operations only `prev` will be terribly inefficient in this implementation, since the list must be traversed from the beginning to find the node preceding a given one on the list; if an efficient `prev` is important, a doubly linked list can be used; see the description at the end of the chapter. The insert and remove operations will be dramatically more efficient than they were in the naive implementation, since we do not need to actually move any cells to accomplish an insertion or deletion—we need only reset a few pointers; figure 2.6 shows a typical remove procedure. Naturally we have to pay for some of these improved features with some new disadvantages. If the value field is small the space used by the next field may be significant, since it could double the space requirements for a single node and effectively cut in half the total number of nodes which may be in use. (Since we expect to make more efficient use of the nodes we have, this may not be a problem.) Also, it turns out that some of the simple operations will be slightly more expensive than in the naive array implementation; this will rarely be significant. Any linked implementation is in some ways more 'dangerous' than the naive implementation. It is possible, for example, for two different nodes to point to the same next node; the use of pointers introduces a whole new spectrum of possible programming errors. As with any data structure, your choice of one implementation over another should be carefully considered in the context of your problem, but except in the simplest of circumstances linked lists are almost always the list structure of choice.

```
void list::remove (list_position target) {
  list_position p;
  if (target == LNULL) {
    p = start;
    start = space.get_next(p);
    if (start == LNULL) lastentry = LNULL;
  } else {
    p = space.get_next(target);
    space.set_next(space.get_next(p),target);
    if (space.get_next(p) == LNULL) lastentry = target;
  }
  space.return_node(p);
}
```

Figure 2.6 Cursor remove

The `remove` function removes the item after position target, though the item at the target position would be more natural. It would be much more difficult to remove the

item at the target position, because the link field of the previous node must be altered, and finding the previous node is quite inefficient (but again, doubly linked lists will solve the problem).

Here is a peculiar feature of this implementation: if we execute `L.remove(q)`, and `q` is not in fact pointing to an item on list `L`, the item following `q` will be removed anyway, from whatever list it is on! The obvious way to correct this is very inefficient: we would need to start at the beginning of `L` and call `next` until we find `q`; there is a better way, which is quite efficient but requires more memory; see the description of 'Head Pointers' at the end of the chapter.

If you anticipate that the number of items ever removed will be small relative to the size of node space, you could instead simply initialize `free` to zero, and increment it every time you grab a new node to insert on a list. When a node is removed, you can just ignore it—nothing points to it, so it won't 'get in the way,' but you won't be able to reuse it, either. A third alternative is to ignore removed nodes unless `free` actually reaches the end of the array. Then you can go through the entire node space to discover unused nodes, link them into a list, and proceed from there—again, when a node is removed from a list it can simply be ignored, not added to the free list. To identify unused nodes you will need some way to mark a node as unused, perhaps by setting the link field to some otherwise illegal value, like -2. Any operation that searches for unused space like this is called *garbage collection*; some programming languages, including LISP and Java, employ a scheme like this to manage the memory allocated to programs by the operating system.

Often we want to search a list until we find a particular node and remove it. Unfortunately, when we finally discover that node `q` is the right one, we are past the previous node which we need to use as a parameter to the remove procedure. Of course, if the list implementation supplies a `prev` operation, that will do the trick, but it may be unacceptably inefficient if this happens frequently. The standard approach to this task is to keep two running pointers or 'fingers' (instead of one) to the list, one right behind the other. When `finger` is pointing to the node to be removed, `prevfinger` is the correct parameter for remove. The code would look something like figure 2.7.

This code is not enough to do the job, of course. The most serious problem is that we might fall off the end of the list, since `searched_for_cell` might not be on the list. The trouble will first appear when finger equals `LNULL` and we call `L.retrieve(finger)`. This type of error is called **referencing through a nil pointer** or **dereferencing a nil pointer**—attempting to follow a link that leads nowhere.

```
finger = L.first();
prevfinger = LNULL;
while (L.retrieve(finger) != searched_for_cell) {
  prevfinger = finger;
  finger = L.next(finger);
}
L.remove(prevfinger);
```

Figure 2.7 Search and destroy

2.3 Pointer implementation

Although the cursor implementation of linked lists is much better than the naive array approach, it still suffers from the need to specify in advance the size of `node_space`. We expect to make better use of space, but we will still have to over-estimate our needs and usually waste space, or take the time to create and populate new arrays. This will often not be a big problem, and in any case there may not be much you can do about it. However, many languages, including Pascal and C, provide tools to help in the construction of linked lists and do not require *a priori* bounds on the number of nodes. Instead a program can request increased storage space as the need arises at run time.

Both Pascal and C provide pointers as part of the language. (Although cursors can also be called pointers, it is common to use the word pointers to mean the sort of built in pointers provided by a programming language.) Using pointers is, not surprisingly, very much like using cursors, but with a different syntax. In C, a variable may contain a pointer, which acts in most important ways just like a variable containing a cursor. There is a pre-declared constant called `NULL` which is the equivalent of the `LNULL` in the cursor based implementation. The similarity is apparent in the list implementation in figure 2.8.

The most obvious difference is that the arrays in `node_space` have disappeared, replaced by the built-in ability of C to manage space; indeed, `node_space` now contains no data at all, merely member functions. These functions could of course be merged into the `list` class itself, but by separating the functions, different implementations of `node_space` can be used without altering the `list` member functions. The links are now wrapped individually with a cell, in a new class called `node`, since there is no longer a way to refer to a link space 'parallel' to the data space.

The operations in the new `node_space` class are somewhat simpler than before, because C++ handles memory for us. In particular, `get_node` is really just a wrapper for `new`, and `return_node` is just `delete`.

By using true pointer-based linked lists, we avoid the need to estimate memory requirements in advance. The program using the lists will be given memory as needed, up to

```
#define LNULL 0

typedef int list_item;

class node {
  friend class node_space;
 private:
  node* next;
  list_item v;
 public:
  node();
};

typedef node* list_position;

class node_space {
public:
  node_space();
  list_position get_node ();
  void return_node(list_position);
  list_position get_next(list_position);
  list_item get_val (list_position);
  void set_val (list_item, list_position);
  void set_next (list_position, list_position);
};

class list {
public:
  list();
  ~list();
  list_position first();
  list_position last();
  list_position next(list_position p);
  list_position prev(list_position p);
  void insert (list_item n, list_position target);
  void remove (list_position target);
  list_item retrieve(list_position p);
private:
  list_position start;
  list_position lastentry;
  static node_space space;
};
```

Figure 2.8 Pointer implementation

the maximum permitted by the operating system. As an added bonus, it is simpler to program with pointers, because the memory management tasks are left to the programming environment.

2.4 Variations

Although linked lists are in most respects superior to the naive array implementation, we have seen that some simple tasks, like finding the preceding item on a list, are quite difficult. A number of common variations on linked lists address some of these problems, typically with a small cost in memory and time.

Doubly Linked Lists: When `prev` is a frequently used operation, a second set of links going the opposite way through the list may be desirable. The costs involved may not be trivial, as the amount of work involved in updating the pointers and the space required to store the pointer are double the requirements for singly linked lists. The code involved will be more complicated, but of course once the core routines are debugged, using the lists will be no more complicated. The existence of these extra links also makes it feasible to define `remove` in a more natural way without losing efficiency.

Header Nodes: In this scheme, every list consists of at least one node called the header or head node, even empty lists containing no cells. There are two main reasons for using a header node: its presence often makes the code needed to handle lists much cleaner, and you will be able to use the header node to store information which is necessary or convenient to have when the program is running. For example, you will not need special cases to deal with the beginning of the list when you use a header node; compare this remove routine with the previous version, assuming that `start` now points to the header node:

```
void list::remove (list_position target) {
  list_position p = new node;
  if (target == LNULL) {
    target = start;
  }
  p = space.get_next(target);
  space.set_next(space.get_next(p), target);
  return_node(p);
}
```

Removing the first item on a list is now the same as removing any other item, since there is a node that precedes it, namely, the header node.

Since there is no cell associated with the header node, the space normally occupied by the cell may be used to hold other information. The `lastentry` field could be moved to the head node, for example.

Head pointers: Recall that it is difficult to determine if a pointer to a node is in fact a pointer to a node on the current list. If this is needed, we can add yet another link field to the nodes, pointing to the header node. Suppose each node has a field `head` that points to the header node. Then if `p` really points to a node on list `L`, it will be true that `L.start==p->head`, and this is easy to check.

In an object-oriented language like C++ the class `list` serves to some extent as a header. It is still true that, for example, inserting at the beginning of the list requires special handling, but it is possible to have what amount to head pointers without a separate header node. The upshot is that while a header node might still be useful in a C++ implementation, it is more often not used. In languages without classes and especially languages without the powerful pointers that C and C++ have, header nodes are more common.

Circular Lists: The next field of the last node on the list is normally `LNULL`, but it can be set to point back to the front of the list, again to either the first node on the list or to the header node. This makes some programming a bit cleaner than it otherwise would be, and also makes it more difficult to accidentally dereference the nil pointer. In this scheme, every node in `node_space` that is being used is pointed to by some other node. This information can be used by a garbage collection routine looking for free nodes.

Free List with Pointers: Using `new` and `delete` is a relatively expensive operation. If a list is very active, a program may call these frequently, giving back memory just before asking for more. Instead of giving back the memory, you could keep a free list, just as in the cursor implementation. When you need a node for an `insert` operation, you first check the free list, and only if it is empty do you resort to `new`. You will want just one free list for the entire class, and you will need to ensure that all the memory gets returned (via `delete`) when the last list disappears.

2.5 A MORE FLEXIBLE IMPLEMENTATION

A more or less standard approach to linked lists has been developed for C++. Not surprisingly, part of the flexibility comes from carefully separating tasks and assigning them to different classes. But there is another substantive change: The implementation we have already seen constructs nodes that contain both a link and the actual data; this means that the data is truly part of the list, and that if we want a single data item to be on two different lists we will have to copy it. If the data might change over time, we will have to

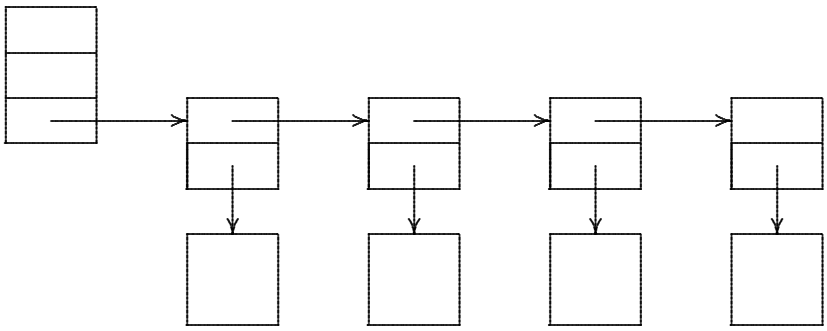


Figure 2.9 A linked list

find all copies of it when we do an update. An alternative is to maintain a single copy of the data, and have the list nodes point to it.

A schematic view of the new implementation is shown in figure 2.9. The links are shown as square boxes divided into two sections; the data items are the undivided square boxes; the three-section box is a header for the list; it is how the list is found and referred to elsewhere in the program. In fact, from the point of view of a user of this structure, the header *is* the list.

We will store information in the header that describes the entire list, so it is important that this information is updated whenever the list is altered. To enforce this, we will allow changes to be made to the list only by the header class. This means that it is convenient for the header to maintain a "current" pointer, indicating where inserts and removals are done. The header will of course need to be able to adjust the position of current. With all of this in mind, the class definitions in figure 2.10 should be mostly self-explanatory. The data items in this example are just ints, but in practice they could be any type of data—railroad cars or personnel records, for example.

Note the use of the friend mechanism: "`friend class list`". We want to make sure that only the header alters the list. By putting everything in the link class in the private section, we guarantee that the outside world can't mess with the link, but then the header also can't access or modify the link data. By declaring class list to be a "friend" of class listlink we allow header to have access to all private data and functions in the link class.

Let's look at an implementation of a few of the functions, beginning with `add_after` in figure 2.11. Here we see another significant difference from our previous implementation: insertion (and removal) always occur at the position `current` rather than a position

```
class listlink {
  friend class list;
 private:
  listlink *next, *prev;
  int * data;
  list * head;
  listlink();
  listlink(int *n);
};

class list {
 private:
  listlink *start, *last, *current;
  int the_size;
 public:
  list();
  ~list();
  int size();
  bool is_empty();
  void reset();             // reset current to NULL
  void add_after(int* n);   // add n after current
  void add_before(int* n); // add n before current
  void add_first(int* n);   // add n at the beginning
  void add_last(int* n);    // add n at the end
  int* remove();            // remove the current item
  bool advance();           // advance current to next item
  bool backup();            // move current to previous item
  bool go_first();          // set current to first item
  bool go_last();           // set current to last item
  int * access();           // return a pointer to the data item
};
```

Figure 2.10 Classes for linked lists

provided as a parameter. This is not crucial, but it is typically how C++ lists are implemented and so we include it as an alternative to our earlier approach. We begin by creating a new link and pointing it at the data item. Then we link the new item in after the current link, unless `current` is the null pointer, in which case we put the new item at the beginning of the list.

Removing a data item is a bit trickier than adding one, as illustrated in figure 2.12. It is not safe to delete the data item itself, as it may still be in use. On the other hand, if the link contains the last pointer to the data item, simply deleting the link results in a memory leak. We want to delete the link and return a pointer to the data item, so that the caller can decide what to do about the data item. We also need to decide what to do with `current` after we remove the current item. The code shown moves current to the link

```
void list::add_after(int* n) {
  listlink * l = new listlink(n);
  if (current) {
    if (current->next) {
      current->next->prev = l;
    }
    l->next = current->next;
    l->prev = current;
    current->next = l;
    l->head = this;
    if (last==current) last=l;
  } else {
    l->next = start;
    l->prev = NULL;
    l->head = this;
    if (start) start->prev = l;
    else last = l;
    start = l;
  }
  the_size++;
}
```

Figure 2.11 The `add_after` function

following the removed link, if there is one, and moves it to the link preceding the removed link otherwise, unless of course the current link is the only link, in which case `current` becomes the null pointer.

The `advance` function is simple but does have a few potential pitfalls. If `current` is the null pointer, we advance it to the first item on the list, except of course if the list is empty. Otherwise `current` really points to a link, so then we check its `next` field to see if we can advance. We return `true` if we are able to advance `current` and `false` otherwise.

```
bool list::advance() {
  if ( !current )
    if ( !start ) return false;
    else {
      current = start;
      return true;
    }
  if ( !current->next ) return false;
  current = current->next;
  return true;
}
```

Figure 2.13 The `advance` function

```
int* list::remove() {
  if (!current) return NULL;
  listlink* new_current;
  int* d = current->data;
  if (current == start) {
    start = current->next;
    if (start) start->prev = NULL;
    else last = NULL;
    delete current;
    current = start;
  } else {
    current->prev->next = current->next;
    if (current->next) {
      current->next->prev = current->prev;
      new_current = current->next;
    } else {
      last = current->prev;
      new_current = current->prev;
    }
    delete current;
    current = new_current;
  }
  the_size--;
  return d;
}
```

Figure 2.12 The `remove` function

It is frequently desirable to examine the items on a list without disturbing the `current` pointer. It might even be necessary to keep track of more than one additional location on the list. On the other hand, we probably don't want to allow any modifications to the list except through the header and its `current` pointer. The standard approach here is to define a third "iterator" class that contains its own version of `current` and is allowed to examine data items, but is not allowed to add or remove data. The class definition is shown in figure 2.14. Our iterator class needs to access the data in the header and the links, so we need to add "`friend class listiterator`" to both the `listlink` and `list` classes. The variables and functions of the iterator class will be almost a subset of the list class, and the function definitions will be almost identical. We add one constructor and a list variable so that an iterator can be associated with a particular list; indeed, in our implementation, the default constructor will produce a useless iterator since it will not be associated with any list. A writer function could be added to allow an existing iterator to be associated with a new header.

```
class listiterator {
 private:
  listlink *current;
  list * head;
 public:
  listiterator();
  listiterator(list* head);
  void reset();
  bool advance();
  bool backup();
  int * access();
};
```

Figure 2.14 The iterator class

2.6 A LIST TEMPLATE

So far we have assumed that the underlying data type is built in to the list definition, which at some level it must be. But it would be much more convenient if the classes didn't need to be literally rewritten for each new data type. C++ provides a mechanism to accomplish this, so that the essential parts of the list classes can be written just once, as "templates", and then lists containing different types of data can be created easily with syntax like this:

```
list<int> L1;
list<double> L2;
```

These declarations create two lists, one to hold ints and one to hold doubles. The basic idea for templates is simple: in the list code, we use a placeholder name, say `list_item`, instead of an actual type like `int`. Then when the compiler sees a declaration like `list<int>`, it runs through the template replacing `list_item` by `int`, and compiles the resulting code to produce classes capable of maintaining lists of integers. The syntax of templates takes some getting used to, but it is not intrinsically difficult. Figure 2.15 shows "templatized" list declarations. The first two declarations are required so that C++ knows about `list` and `listiterator` before they are referred to in `listlink`.

Note that before *each* new item (in this case, all the items are classes) the code `template <class list_item>` is required. This code would be contained in a `.h` file as usual, say `lists.h`. The templatized code would go in `lists.cc`, portions of which are shown in figure 2.16. This is obtained from the original list code by replacing the original fixed cell type `int` with `list_item`, and replacing the class names `list`, `listlink`, and `listiterator` with `list<list_item>`, `listlink<list_item>`, and `listiterator<list_item>`, *except* (somewhat confusingly) in the names of the constructor and destructor functions.

Since `lists.cc` contains merely templates for real functions, it does not make sense to compile `lists.cc`. Instead, `lists.cc` is treated much like a `.h` file. First we include it in the main file, using `#include lists.cc`, then when the compiler sees `list<double>` it can replace `list_item` by `double` everywhere and compile the resulting code.

In reality, it's not quite so simple because most programs are written in multiple files, compiled separately, and then linked together. If two different files use lists of integers, then we don't want two identical copies of the code to be generated and linked into the final executable program. There are two ways to deal with this: modify the linker to identify multiple copies of the code and put only one copy in the final program, or somehow guarantee that only one copy exists. There are two disadvantages to the first method: time is wasted compiling the code multiple times, and the linker has to be rewritten. The second method also has a disadvantage, namely, we need to guarantee that there is only one copy to begin with. Here again there are two options: make the compiler smart enough to create only one copy, or require the programmer to do it. The g++ compiler is capable of all three variations. I will describe here the simplest, though it requires some extra work by the programmer, namely, it is up to the programmer to provide a single copy of every required class.

Conceptually, however, it is quite simple. For each list type that you need, you create one small `.cc` file to instantiate that type, for example, you might use a file called `double_list.cc` to create a list class for doubles. In other `.cc` files, as usual, you include only `lists.h`, but in `double_list.cc` you include both `lists.h` and `lists.cc` (or include only `lists.cc` if it in turn includes `lists.h`). The rest of `double_list.cc` is then exceptionally simple—it consists of a single line:

```
template class list<double>;
```

Then `double_list.cc` is compiled to produce `double_list.o` which is linked in as usual. If a single program uses, say, both lists of doubles and lists of integers, you would do the same sort of thing in `int_list.cc` and link it in as well.

```
template <class list_item>
class list;
template <class list_item>
class listiterator;
template <class list_item>
class listlink {
  friend class list<list_item>;
  friend class listiterator<list_item>;
 private:
  listlink *next, *prev;
  list_item * data;
  list<list_item> * head;
  listlink();
  listlink(list_item *n);
};
template <class list_item>
class list {
  friend class listiterator<list_item>;
 private:
  listlink<list_item> *start, *last, *current;
  int the_size;
 public:
  list();
  ~list();
  int size();
  bool is_empty();
  void reset();
  void add_after(list_item* n);
  void add_before(list_item* n);
  void add_first(list_item* n);
  void add_last(list_item* n);
  list_item* remove();
  bool advance();
  list_item * access();
};
template <class list_item>
class listiterator {
 private:
  listlink<list_item> *current;
  list<list_item> * head;
 public:
  listiterator();
  listiterator(list<list_item>* head);
  void reset();
  bool advance();
  list_item * access();
};
```

Figure 2.15 List class templates.

```
template <class list_item>
list<list_item>::list() {
  the_size = 0;
  start = last = current = NULL;
}

template <class list_item>
list<list_item>::~list() {
  current = start;
  while ( current ) { remove(); }
}

template <class list_item>
int list<list_item>::size() {
  return the_size;
}

template <class list_item>
bool list<list_item>::is_empty() {
  return (the_size==0);
}

template <class list_item>
void list<list_item>::add_after(list_item* n) {
  listlink<list_item> * l = new listlink<list_item>(n);
  if (current) {
    if (current->next) {
      current->next->prev = l;
    }
    l->next = current->next;
    l->prev = current;
    current->next = l;
    l->head = this;
    if (last==current) last=l;
  } else {
    l->next = start;
    l->prev = NULL;
    l->head = this;
    if (start) start->prev = l;
    start = l;
    if (!last) last = l;
  }
  the_size++;
}
```

Figure 2.16 List class functions.

3

Stacks and Queues

Queues and stacks are two special list types of particular importance. They can be implemented using any list implementation, but arrays are a more practical solution for these structures than for general purpose lists.

3.1 Stacks

A stack is a list with the operations insert and remove restricted to one end of the list, called the top; insert and remove are usually called push and pop respectively. The natural analogy is to a stack of books: it is easy to add or retrieve a book from the top of a pile, but more difficult in the middle of the pile. Another good model is the spring loaded stack of dishes found in many dining halls and cafeterias; 'push' and 'pop' seem particularly appropriate in this context. The properties of a stack are often referred to as LIFO (Last In First Out), which is sometimes also used as a noun, that is, a stack is a lifo.

Stacks are often useful in systems programming and organization. A simple example of a use for a stack is as a text buffer in a line editor. As a line of text is typed in, it is stored in a stack. When a mistake is noticed and the delete key is pressed the editor merely pops the stack, thus eliminating the most recently typed character.

Stacks are usually implemented as arrays. In any given application there will typically be only a few stacks in use, so the problem of wasted space is not important. If stacks are expected to be highly variable in size, it may be appropriate to use a pointer based approach.

For the array implementation we can borrow directly the naive array based implementation of lists. In the record declaration the `lastentry` field would usually be called `top`, and the `push` and `pop` routines would of course not allow for a parameter indicating position in the stack. Typically the `pop` function will not only remove the top item from the stack but return the value of the item. Sometimes this is the only way to examine a value, and there is no separate `retrieve` function. On the other hand, in some modified implementations, it may be possible to examine any item on the stack, but only to insert and remove at the top. In this case, the `get_top` function might be declared as `get_top(int&,int)`, and the call `get_top(m,-2)` would retrieve the value stored in the second item below the top of the stack.

```
class int_stack {
 public:
  int_stack();    // initialize a stack with default capacity
  int_stack(int); // initialize a stack with capacity at least n

  // The next 4 functions return true on success, false on failure

  bool push(int n);     // push n on the stack
  bool pop(int& n);     // put the top value in n and pop the stack
  bool get_top(int& n); // put the top value in n
  bool clear();         // reinitialize the stack so it is empty

  bool empty();         // return true if the stack is empty, false otherwise
  bool full();          // return true if the stack is full, false otherwise
  int size();           // return number of items currently on the stack

  void dump();          // dump the stack to cerr

 private:
  int top,capacity;
  int* stk;
}
```

Figure 3.1 Array based stack

3.1.1 Arithmetic evaluation

A good illustration of the use of stacks is in the evaluation of arithmetic expressions. We will be concerned with two standard ways to write arithmetic expressions called postfix

(or reverse Polish) and infix. An example of the same expression written in both forms:

$$4 + 5 * 3 - 8$$
$$4\ 5\ 3 * +\ 8\ -$$

To evaluate an expression in postfix you read left to right and perform an operation as soon as it is encountered, using the two operands immediately to the left; these operands are then deleted and replaced by the result. There is never a need for parentheses in postfix; these are equivalent expressions:

$$(4 + 5) * 3 - 8$$
$$4\ \ 5\ +\ 3 * 8\ -$$

You should be able to see quite easily that a stack is just the ticket for keeping track of the evaluation of a postfix expression. The symbols are read left to right; each value is pushed onto a stack; when an operator is read, the action it denotes is performed using the operands at the top of the stack, those operands are popped and the new result is pushed. Let's follow the stack through the evaluation of a simple expression, $945+3*8--$, shown below.

		5		3		8		
	4	4	9	9	27	27	19	
9	9	9	9	9	9	9	9	−10

There are some very definite advantages to postfix notation, but infix will probably continue to be much more widely known and used. An obvious approach to evaluating infix notation automatically is to convert to postfix and then use the algorithm outlined above to evaluate. Figure 3.2) shows a step by step conversion of an expression from infix to postfix. When we encounter a number, we can immediately write it in the output postfix expression; we must hold on to operators for a while until it's appropriate to write them. Notice that in postfix both operands must come before the operator, so each operator will have to be held at least one step before it can be written. When an operator is seen in the input, we have to decide whether it indicates that some waiting operators can now be written to output; for example, when the first '−' is seen, we can tell that the waiting '∗' and '+' can be written, because multiplication is done before subtraction in infix expressions, and since the '+' has the same precedence as '−', but occured first, it too can be written.

Except possibly for the parentheses, this should look fairly straightforward, and you might be able to throw together an ad hoc program to handle these operators. Notice

Remaining Input	Output	Ops Waiting
$1+2*3-4+5/6/(7+8)$		
$*3-4+5/6/(7+8)$	1 2	+
$-4+5/6/(7+8)$	1 2 3	+ *
$-4+5/6/(7+8)$	1 2 3 *	+
$-4+5/6/(7+8)$	1 2 3 * +	
$+5/6/(7+8)$	1 2 3 * + 4	−
$+5/6/(7+8)$	1 2 3 * + 4 −	
$/6/(7+8)$	1 2 3 * + 4 − 5	+
$/(7+8)$	1 2 3 * + 4 − 5 6	+ /
$(7+8)$	1 2 3 * + 4 − 5 6 /	+ /
)	1 2 3 * + 4 − 5 6 / 7 8	+ / +
	1 2 3 * + 4 − 5 6 / 7 8 +	+ /
	1 2 3 * + 4 − 5 6 / 7 8 + /	+
	1 2 3 * + 4 − 5 6 / 7 8 + / +	

Figure 3.2 Infix to postfix

especially that the operators waiting for action are behaving as if they are on a stack. The key to programming this whole process in a uniform manner is the idea of operator precedence. Normally '$*$' takes precedence over '+'; it is this which governs whether an operator should be written out or saved. A simple precedence scheme might assign to each operator an integer; then the ordering of the integers would reflect the ordering of the operators in terms of precedence. Now when we read an operator with high precedence, say '$*$', and compare it to an operator with low precedence, say '+', waiting on the top of the stack, we know by comparing the precedence numbers that the '$*$' should be saved without first writing out the '+'. Parentheses temporarily disturb the precedences, allowing a '+' to be placed on the stack above a '/'.

The problem with this for more sophisticated use is that some symbols need to have different precedences at different times. It turns out that a slick way to handle parentheses is to treat '(' as an operator, i.e., push it on the 'waiting' stack and use it to control, via its precedence, the order in which other operators are written. The problem is that its precedence must be different when it has just been encountered than when it is on the waiting stack. When we first see '(' it must be put onto the stack immediately, before any waiting symbols are written out; thus it must behave as if it had high precedence. Once on the stack, however, it should remain until a ')' comes along; thus it must behave as if it had low precedence. We can take all this into account by keeping precedence information in a two-dimensional array of boolean values that indicate whether one operator has precedence over another. Each pair of operators can be associated with two spots in this array depending on which is listed first. This allows us to have separate entries for operators

		onstack				
		+	−	∗	/	(
	+	F	F	F	F	T
	−	F	F	F	F	T
incoming	∗	T	T	F	F	T
	/	T	T	F	F	T
	(	T	T	T	T	T
	)	F	F	F	F	T

Figure 3.3 Operator precedence

when first encountered and when already on the stack. A portion of the precedence array might look like figure 3.3.

Here an entry of F means that the 'incoming' operator should cause the operator on top of the stack to be written as part of the postfix expression, because it is false that the incoming operator takes precedence over the top operator. Notice that I have not bothered to list ')' among the 'onstack' operators, since it will in no case be pushed. The 'driver' routine for the translation using this table would look something like figure 3.4

```
while ( !prec[token][top_item(op_stk)] ) {
  if (!op_stk.pop(op)) {
    cerr << "op_stk underflow\n";
  } else {
    cout << op;
  }
}
if ( token != RPAREN ) {
  op_stk.push(token);
} else {
  op_stk.pop(op);
}
```

Figure 3.4 Infix to postfix code

You should work through the above example using this routine and the precedence table. The '(' will now appear on the stack at the appropriate point and will insure that the '+' (adding 7 and 8) gets written before the '/'. You will also notice that the algorithm is not correct as it stands, because there is no provision for stopping the while loop when there is no top operator. One solution is to introduce an 'imaginary' symbol, often denoted '$', which is pushed on the stack at the beginning of processing. This symbol has very low precedence so that nothing will force it to be popped: it will stop the while loop whenever it becomes the top operator. Notice that this '$' need only appear on the 'onstack' side

of the table; since it is not a 'real' symbol we will never actually read one as an incoming symbol. Also there is no automatic way to signal the end of an expression so that all waiting operators can be written out. For this we can introduce an end-of-expression marker ';' with low precedence which will force everything (except '$') to be popped. This marker will only appear on the incoming side of the table.

Now notice that there is really no need to evaluate infix expressions in two stages (translation to postfix and then evaluation). Whenever it is time to write out an operator to the postfix expression, it is time to *do* the operation. That is, if we keep the operands on a separate stack, then we can do the postfix evaluation simultaneously with the translation. This is not only quicker, it eliminates the storage necessary to hold the postfix expression until evaluation. In this case the body of the while loop above will become more complex; it will consist of a switch statement with a label corresponding to each operator that might be on the stack.

Once the framework is in place and working it becomes quite easy to add more operators, pretty much just by adding a new case label with statements that know how to perform the new operation. The unary minus, assignment and exponential operators are all quite easy to add. The exponential operator ($**$) is usually assumed to associate to the right instead of the left. This can easily be handled by setting the table entry `prec[**,**]` equal to T, unlike other diagonal entries which are F.

3.1.2 Recursion and Run-time Stacks

Another common and very important use for stacks is the run-time stack. A compiler for a language that allows recursive functions creates this stack when it translates a program into machine language. The stack is in operation when the translated program (*not* the compiler) is running. The translated program uses this stack to keep track of the order in which functions are called and the variables to which each function has access.

When a function is called, whatever local variables it owns become active. If there were no recursive procedures the memory locations for these local variables could be set aside for the duration of the program; the compiler, as it generates code for the procedure body, would know where these values could be found. In a recursive call of a procedure, however, we need to keep track of more than one copy of every local variable: one copy for each time the procedure has been called. Moreover, we can not in general predict how many copies will be needed.

The run-time stack is a stack of **activation records** which contain enough space to store all the local variables (including parameters) of the function. When the compiler needs to generate code involving the use of a local variable, it uses the appropriate loca-

tion in the activation record; at run-time the program knows how to find the top activation record, so it automatically gets the right copy of the local variable. In addition the activation records contain a field indicating where control should be passed when the procedure finishes, i.e., where the next program step can be found. Recursive procedures are not always the best way to solve a problem, but often a recursive process will be very easy to program and understand. The recursive structure of the program can do some very complex bookkeeping almost as if by magic; underlying this is really the run-time stack.

The knapsack problem is an example of one that is particularly easy to solve recursively; a non-recursive solution would be much more difficult to organize. A simple version of the problem goes like this:

> *We have a number, n, of objects with weights $w_1, w_2, \ldots, w_n$ (not necessarily distinct). The problem is to determine whether we can choose some subset of the objects with a total weight equal to some specified target weight W.*

A solution is simple to state: try all combinations until the target weight is made or until we run out of combinations. On a small set of weights this is easy to do by inspection, but the programming of such a task for a large number of weights is not trivial. (Unfortunately, this is an example of a problem for which the best known solution is of the 'try all possibilities' variety.)

Let's state the proposed solution with a bit more precision and see how this restatement leads almost directly to the recursive procedure to solve the problem. Given a target weight W, we first guess that we can use the first weight w_1. In that case we have a new version of the knapsack problem to deal with: the target weight is now $(W - w_1)$ and the list of available weights is $w_2, w_3, \ldots, w_n$. We solve this smaller problem; if the answer is *yes* we're in business, because then the answer to the original problem is also *yes*. If the answer to the smaller problem is *no* then we decide that the weight w_1 is of no use to us, and instead ask whether we can make the target W out of $w_2, w_3, \ldots, w_n$. We proceed in this manner until we have tried all possible combinations. The only thing left is to examine a couple of special cases in which we can give an immediate answer. If at any point we are trying to solve any version of the knapsack problem with target weight 0 then we can answer *yes* immediately, since we can always make a target of 0 by choosing no weights at all. If we have a target weight which is less than 0, the answer is clearly *no*, since a negative weight makes no sense. Finally, if we have a non-zero target weight but no more weights to consider, we return the answer *no*. This has a nearly direct translation into a recursive procedure, shown in figure 3.5 along with a simple main program to test it. When this program runs, it produces the following output:

```
Target : 10
```

```
Weight number: 4   Value: 1
Weight number: 3   Value: 4
Weight number: 1   Value: 5
Success
Target : 23
Failure
```

Let's follow how the run-time stack can keep track of the whole mess. The activation records look like this:

target
candidate
return to

The target and candidate fields are initialized to the value of the parameters when the function is called. The 'return to' field will have the value '$*$' or '$**$' corresponding to the lines marked in the procedure, except for the first call which is from the main program. Below is a trace of the run-time stack as knapsack works on the problem with target 10 and weights 7, 5, 8, 4, 1. The stack is shown after each push and pop, i.e., as each call to knapsack is about to start and when control is passed back to the calling location. Remember that knapsack returns a boolean value to the point of call where it either controls an if-statement (at $*$) or simply passes the value on ($**$).

10
0

10	3
0	1
	$*$

10	3	-2
0	1	2
	$*$	$*$

10	3
0	1
	$*$

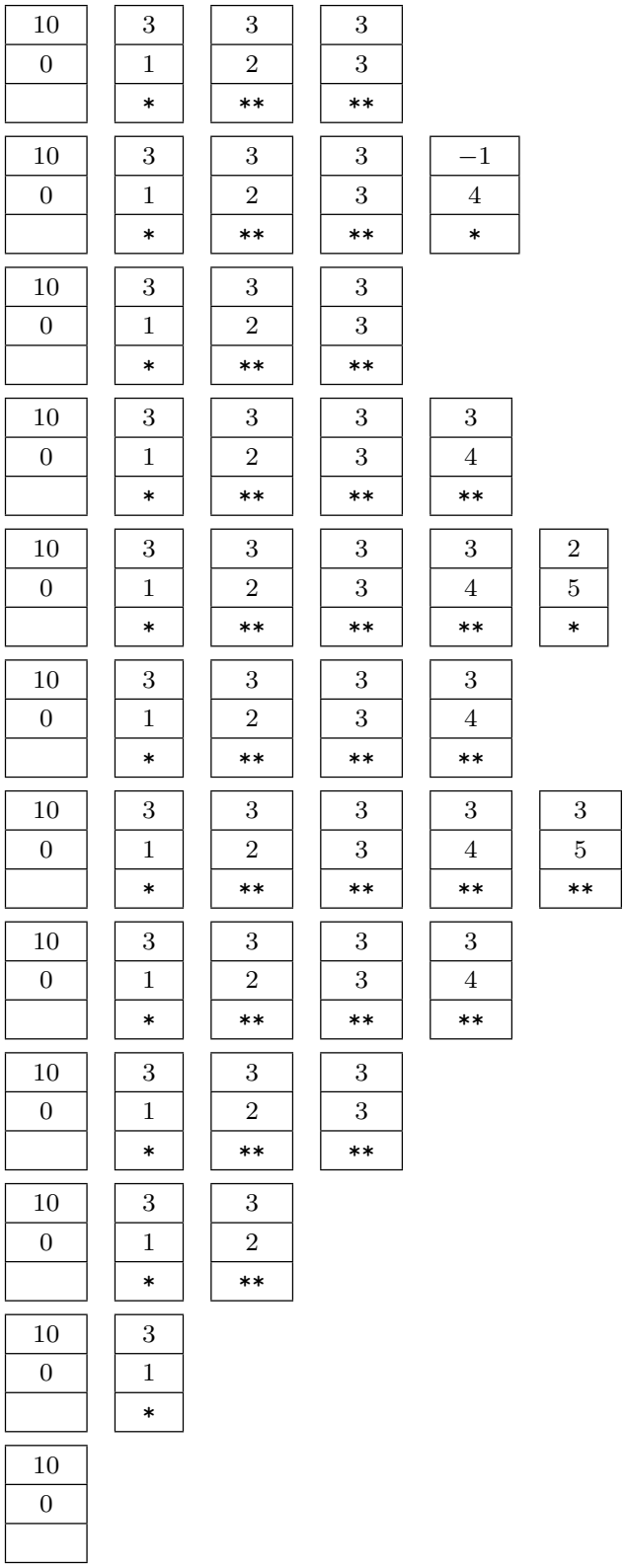
10	3	3
0	1	2
	$*$	$**$

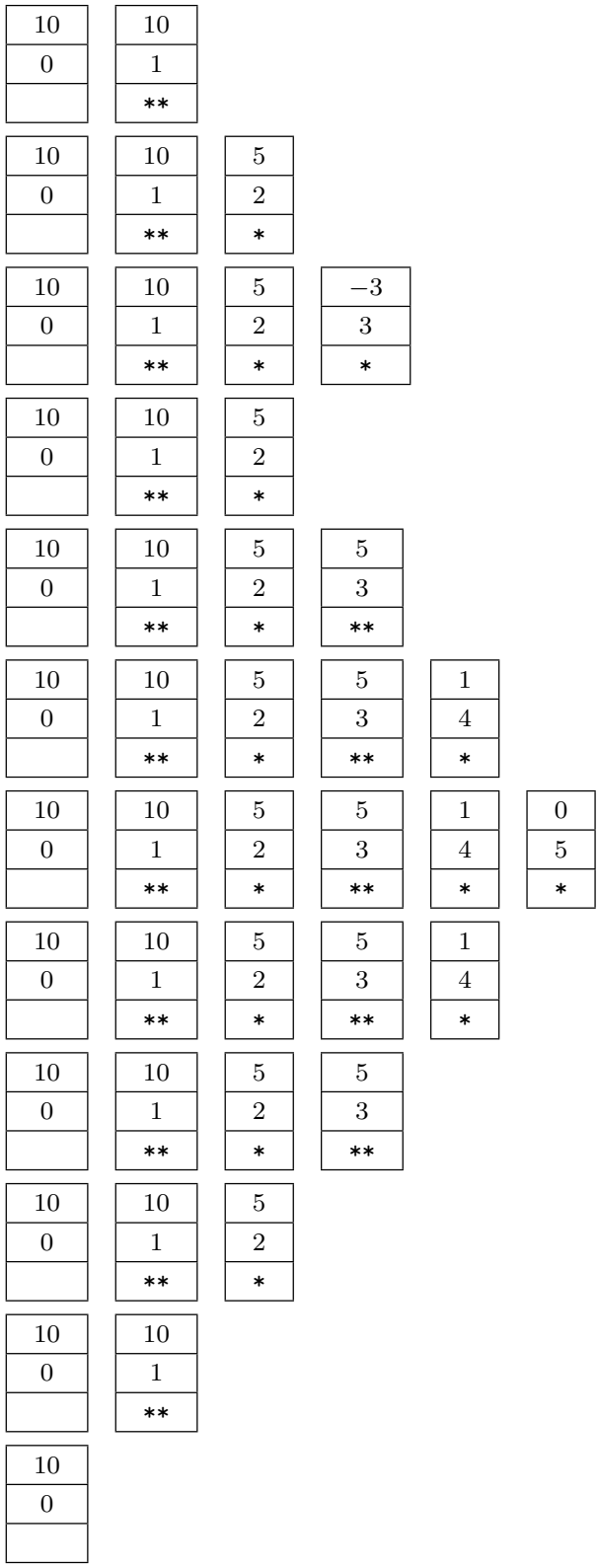
10	3	3	-5
0	1	2	3
	$*$	$**$	$*$

```
#include <iostream.h>

#define MAX 5

int knapsack (int target, int candidate, int wt[MAX]) {

  if (target == 0)
    return (true);
  else if (target < 0 || candidate >= MAX)
    return (false);
  else if ( knapsack(target-wt[candidate],candidate+1,wt) ) {            // *

    // At this point we have a 'yes' answer to the sub-problem using
    // 'target-weight[candidate]' and 'candidate + 1'. Hence we know
    // that we can use the current weight 'candidate' in a solution to
    // the problem.

    cout << "Weight number: " << candidate;
    cout << "   Value: " << wt[candidate] << endl;
    return (true);
  } else {

    // At this point we know that weight[candidate] won't help,
    // so we try the problem with the same target but starting with
    // the next weight, 'candidate + 1'. }

    return(knapsack(target, candidate + 1,wt));                          // **
  }
}

main() {
int weight[MAX] = {7,5,8,4,1};
int t = 23;

 cout << "Target : 10" << endl;
 cout <<  (knapsack (10,0,weight) ? "\nSuccess" : "\nFailure") << endl;

 cout << "\nTarget : " << t << endl;
 cout <<  (knapsack (t,0,weight) ? "\nSuccess" : "\nFailure") << endl;

}
```

Figure 3.5 Knapsack function

10	3	3
0	1	2
	*	**


10 3 3 3
0 1 2 3
* ** **
10 3 3 3 −1
0 1 2 3 4
* ** ** *
10 3 3 3
0 1 2 3
* ** **
10 3 3 3 3
0 1 2 3 4
* ** ** **
10 3 3 3 3 2
0 1 2 3 4 5
* ** ** ** *
10 3 3 3 3
0 1 2 3 4
* ** ** **
10 3 3 3 3 3
0 1 2 3 4 5
* ** ** ** **
10 3 3 3 3
0 1 2 3 4
* ** ** **
10 3 3 3
0 1 2 3
* ** **
10 3 3
0 1 2
* **
10 3
0 1
*
10
0


10
0
10
1
**
10
0
10
1
**
5
2
*
10
0
10
1
**
5
2
*
−3
3
*
10
0
10
1
**
5
2
*
10
0
10
1
**
5
2
*
10
0
10
1
**
5
2
3
**
10
0
10
1
**
5
2
*
5
3
**
1
4
*
10
0
10
1
**
5
2
*
5
3
**
1
4
*
0
5
*
10
0
10
1
**
5
2
*
5
3
**
1
4
*
10
0
10
1
**
5
2
*
5
3
**
10
0
10
1
**
5
2
*
10
0
10
1
**
10
0

3.2 Queues

A queue is a list in which insertions can be made only at one end (the rear) and deletions at the other (the front). The properties of a queue are often summarized as FIFO (First In, First Out); the queue is useful whenever some sort of waiting is involved. Queues are widely used in writing system software because of the competition for scarce resources. When a process wants to use the CPU, the printer, the disk or any other system device it might well be placed on a queue to wait its turn. (Frequently this will be some sort of modified queue since queues are a bit inflexible. For example, you can remove an item from the printer queue before the system 'deletes' it (i.e., prints it), and there is a scheme of priorities so that even if you 'get in line' first, some job with higher priority might get printed first.)

Sometimes what is called a buffer is a queue. For example, when you 'type ahead' at a terminal without seeing the characters you are typing, they are being stored in some sort of text buffer. When the computer gets around to you again, it takes the waiting characters from the front of the queue, while you may continue to type and add them at the rear.

You should always try the simplest possible implementation of a new structure, both because it's often better to have something that works than nothing at all and because the simplest way may be the best if your application is not particularly demanding. The simplest possible way to organize a queue is in an array with the rear 'growing' toward the higher indexed slots in the array and the front moving behind it. This is easy to implement and quite efficient. The problem is that when the rear hits the top of the array no more items can be added, even if there are empty spots at the bottom of the array. The obvious solution (if one is needed) is to move everything down whenever a deletion is made; this probably conforms closely to your idea of a 'queue' in the normal sense of 'waiting line.' We could implement this by using the naive array implementation of lists and modifying insert and delete to conform to queue specifications (this just amounts to removing one parameter, making it impossible to specify where to do the insert or delete). Alternately, we might move the whole queue down only when we actually run out of room at the top. Then we get the benefit of many incremental moves for the cost of only one (notice that it is not more expensive to move a block of items all the way down the array than it is to move everything one position down; this is because arrays are implemented as random access structures). For lists we needed to move items after every delete because the items on the list had to occupy adjacent spots in the array (otherwise the bookkeeping would get out of hand). The items on the queue will always be adjacent so the move is not crucial

to the function of the queue but rather a way to make more efficient use of the allotted space.

Either of the modified versions above will work and will make the best possible use of space, that is, will not crash for lack of space until the array is really full. Nevertheless, queues are often quite active beasts and the cost of repeated moves of long blocks of items may add up. Suppose we start with the simplest version again (just let the rear go until it hits the top) and modify it in a different way. Instead of moving the whole queue we can simply make our addition in an existing empty position. We have to do this in some reasonable way so that it is easy to tell who is next in line. Fortunately there is a natural answer: we simply let the queue 'wrap around' to the beginning of the array again, and grow toward the retreating front. This implementation is referred to as a 'circular array' implementation and is the one most frequently encountered. It makes very good use of space, it is efficient in operation; its main drawback is the possibility that the array will become full, a property shared by all simple array implementations.

It should be obvious that a queue can be implemented using any linked list implementation, but this is not often necessary. As a rule only some small number of queues will be needed in any particular environment, so the lack of flexibility in the use of space is rarely a problem. The necessity of declaring the space for the queue in advance is also not usually a problem. Queues are used for waiting; if a queue grows until it fills the array it is in, then of course you should try increasing the size of the array. If the problem persists, it probably means that the resource being waited for is badly overloaded and increasing the size of the queue without bound will not solve the problem. You should usually consider using linked lists to implement queues only when the size of the queue in normal operation varies considerably or the space an array would occupy when the queue is nearly empty is needed elsewhere. In some situations it may be deemed unacceptable ever to get a full queue, in which case a linked implementation would be appropriate.

The details of the circular queue implementation deserve some discussion. We will need two pieces of information for each queue: where in the array we can find the front and the rear of the queue. The action of the insert and remove routines should be fairly clear: to insert we find the rear of the queue, insert in the appropriate spot and update the value of the rear variable. Typical functions are shown in figure 3.6. I have really combined the `remove` and `retrieve` operations since in many uses of a queue retrieve without a remove is not done; of course they could be implemented separately.

```
class cell_queue {
private:
  int front,rear,capacity,size;
  cell* the_queue;
public:
  cell_queue();
  cell_queue(int n);
  ~cell_queue();
  bool enqueue (cell n);
  bool dequeue(cell& n);
};

bool cell_queue::enqueue (cell c) {
  if (size < capacity) {
    rear = (rear + 1) % (capacity);
    the_queue[rear] = c;
    size++;
    return true;
  } else {
    return false;
  }
}

bool cell_queue::dequeue(cell& c) {
  if (size > 0) {
    c = the_queue[front];
    front = (front + 1) % (capacity);
    size--;
    return true;
  } else {
    return false;
  }
}
```

Figure 3.6 Circular queue

4

Hash Tables

Many useful data structures can be viewed as *sets*, distinguished from each other by the collection of operations each provides. Names commonly used for different set types are *dictionary*, *symbol table*, *union-find*, *delete-min*, and *list*. A typical dictionary, for example, would provide operations to insert, remove and find; the items stored would be words together with associated information, and the find operation would generally operate on only the word itself. Of course, a list can be used to implement a dictionary, but the linear order imposed by the list structure may not be required. This may seem a peculiar thing to say about a dictionary, but to support the operations insert, remove, and find, it may not be necessary to store the words in order. In fact, we know that the linked list is not a very efficient lookup structure.

Often the best implementation for a dictionary (and less often for some of the other set types) is the structure known as the hash table, which has two common forms. In both we attempt to associate a given value, such as an English word, with an index for an array, and store the value and associated information, such as the definition of the word, at that location in the array. This association takes place via a *hash function*, say h, which takes as input the identifier or word and returns the index value associated with it. If the array were called `Webster`, for example, we would like to look up a word with a statement like

```
if (Webster[h(the_word)].word == the_word) { ...
```

Of course the existence of such a function is not in doubt—we could, for example, list the words in alphabetical order and say that h of the first word is 1, h of the second word is

2, and so on. The problem is that this function is difficult to compute; we seek a function that can be computed rapidly.

In one way a dictionary of the English language is a poor example here, because the contents are relatively static. Another typical example, usually called a symbol table, is the dictionary of variable and function names that a compiler must maintain as it compiles a program. We can't afford to allocate an array large enough for all possible variable names, so it is in fact impossible to design a hash function h that will send different names to different array indices. What we can hope for is that in practice very few names in a program will map to the same index. This means that we want h to have some special properties:

1. h is uniform: it should distribute all possible names evenly over the range of index values; for example, one half of all names should be sent to the first half of the array. Moreover, a random selection of names should be spread uniformly over the index range.
2. h is randomizing: similar names should not cluster around the same index value, but should be randomly distributed.

In the case of names or words, that is, just sequences of printable characters, a reasonably good choice for a hash function is: add the ordinal values of all characters in the name and then mod by the table size (i.e., divide by the table size and take the remainder). This is easy to do and seems to perform well in practice (assuming that the table size is fairly small compared to a typical ord value). Note that it does not perform particularly well with respect to property 2, since variable names like `counter_1` and `counter_2` hash to consecutive values. A similar function that may work better, and accommodates larger table sizes, is

$$h(s) = \sum n_i \cdot \text{ord}(s[i]) \pmod{M},$$

that is, multiply the value of each character by a number n_i before adding mod M. The n_i should probably be small prime numbers, and for simplicity will repeat, that is, the sequence n_i might be $(3, 5, 7, 11, 13, 3, 5, 7, \ldots)$. Hash functions for character strings and other types of data have been extensively studied, and depending on the application a more sophisticated function may be necessary.

It is inevitable that in practice two names will appear that want to occupy the same place in the table; when this happens we have a **collision**. A number of techniques for handling collisions have been studied; they fall into two general categories:

Internal addressing. This is sometimes called *closed addressing*, but unfortunately is also sometimes called *open addressing*, so it's better to avoid those terms. If we attempt to insert a name into a position that is already occupied, we search the array in some

prearranged way until we find an unused space. The performance of this method depends on the sophistication of the search strategy. What about looking up a name? The following algorithm would seem to work: we first go to the position indicated by the hash function. If the spot is empty we conclude that the name is not in the table. If it is occupied, but by some other name, we search (in the order dictated by the chosen strategy) until we either find the name or find an unused position.

This procedure has one sneaky flaw: If we allow deletions to be made we must search the entire table before concluding that the name is not present. This is because we may have inserted the name and then later deleted one of the intermediate names, i.e., one of the names that caused us to continue searching during the insertion process. When we are doing a lookup, we will come to an unused position that was not unused at the time of insertion, so it is not safe to assume that the name we seek is not in the table. (If we never remove items, of course, this is not a problem; often this is true of symbol tables, because once a name is declared it remains active as long as the symbol table is active.) We can correct this by using a special value to mark deleted positions so that we can distinguish 'never used' from 'empty.' One remaining problem is that we cannot store more names than there are positions in the array. This means, as usual, that we will have to overestimate the number of names that might come up in practice, or be prepared to create a new, bigger table if the table fills up.

The simplest strategy when searching for a new position is simply to consider the array positions in order. The problem is that the data starts to clump, and as clumps grow, it becomes ever more likely that a new value will be added to an existing clump. When these clumps of items get large, searching for items in the table becomes slower. Nevertheless, this strategy can be valuable as long as the table stays sparsely populated. A more sophisticated approach is to search through the table by increments bigger than 1, and to vary this increment based on a secondary hash function. For example, suppose w is a word and $h(w) = i$. We examine `table[i]` and discover it is occupied. We compute $h_2(w) = m$, and then start to examine positions `table[i+m]`, `table[i+2*m]`, and so on. If m is relatively prime to the the length of the table, this will eventually look at every position if necessary, but different words will generate different paths through the array.

Chained bucket hashing. This is sometimes called *open addressing*, but unfortunately is also sometimes called *closed addressing*, so it's better to avoid those terms. This method uses each array position as the header for a linked list. Thus we can store many names at one array position: whenever we hash to a position in the array, we simply add new nodes to the list that begins there. This method is faster and simpler when collisions occur and it is next to impossible for the hash table to become full, so we don't have to specify in advance how many entries will be allowed. If we have chosen a good hash

function the length of a typical list will be only N/B, where N is the number of names in the table and B is the length of the array. If B is large compared to N then a typical list will contain only a few elements—ideally, the lists will have an average length less than 1. The time required to look up an entry will thus be the time spent to compute the hash value plus the time needed to search a (very) small list.

If the number of items in a hash table reaches the length of the hash table, then of course collisions become more frequent and the lists begin to grow. In some applications it may be worth it to create a new, longer hash table at this point. The items in the original table must be removed and inserted into the new table one at a time, since the hash values will almost certainly be different in the new table. This is relatively expensive, but the time lost will be recovered if the table continues to grow and lookups are frequent.

5

Trees

5.1 Introduction and definitions

Once you understand the key features of linked lists, in which a node itself contains information about "where to go next," it should be apparent that a node can contain many such directional pieces of information and can be pointed to by the link fields of many other nodes. With a little imagination you can see what a nightmare unrestricted linking might be; complicated linked structures can be very useful, but you should always strive to keep the structure as simple as possible.

Fortunately there are very useful kinds of multi-linked structures with quite restricted linking structure. After arrays and lists, **trees** are probably the most widely used data structures. Even more than is the case for lists, "tree" is a broad term that covers a wide variety of structures with basic similarities but many differences in detail.

Like lists, trees are composed of nodes containing information of some other type, say "cell," together with link information. Recall that some nodes in a linked list might not contain cell information, namely, header nodes which contain link information only (or link information plus some other data useful to the implementation). Such nodes occur more often in some applications of trees.

The key relationship between nodes in a list might be described as the *next* relation, i.e., one node is or is not next after another. The fundamental relationship between nodes in a tree is usually called the *parent-child* relation: one node may be a **child** of another, which is said to be its **parent**. As you might guess, a good example of a tree structure is the family tree, although the terminology of the subject in computer science turns out to be

a peculiar mixture of genealogy and botany. To be a bit more precise about terminology: In a tree every node is the child of at most one other node called its **parent**; exactly one node has no parent and is called the **root** of the tree. A typical picture of a tree appears in figure 5.1.

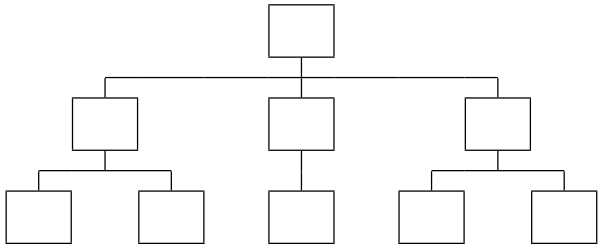


Figure 5.1 A typical tree

Trees are extremely useful as ways to structure data and have received an enormous amount of theoretical and practical attention. For theoretical work, the brief description given above is not sufficient to capture precisely what it means to be a tree. For example, it does not rule out loops in the linking structure. Rather than continue to list requirements until it seems that we have finally nailed down the idea of tree, it is common to see something like the following more formal definition:

1. The empty set is a tree, called the empty tree.
2. A single node is a tree.
3. If X is a node and if there are k trees $T_1, T_2, \ldots, T_k$ with roots $X_1, \ldots, X_k$ (k is some integer), then we may form a new tree by declaring that X is the root and $X_1, \ldots, X_k$ are its children.

This is an example of a **recursive definition** since we are told how to make a tree assuming that other trees have already been produced. As usual, there is an escape clause which lets us stop the recursive process: a single node is a tree by itself.

Besides the theoretical importance of a precise definition, this particular definition is interesting in a practical sense, for many algorithms that deal with trees do in fact construct big trees out of small ones in precisely the manner described in the definition. We will see an excellent example below in the Huffman algorithm.

Compared to lists, there is quite a bit of terminology used to talk about trees; fortunately most of it is quite suggestive. We have already mentioned the **root**, the only node with no parent. A **leaf** in a tree is a node with no children; there will usually be a large number of leaves in any large tree. All nodes that are not leaves are called **interior** or

internal nodes, including the root. A **path** from one node to another is a sequence of nodes in the tree such that each is the parent of the next, i.e., a path is a list embedded in the tree. If there is a path from node X to node Y, we say that X is an **ancestor** of Y and that Y is a **descendant** of X. The **length** of a path is the number of nodes in the path minus one (this is the number of links in the path). The phrase "path through the tree" often is used to denote a path that starts at the root and ends at a leaf. The **height** of a tree is the length of the longest path through the tree.

A **subtree** of a tree is formed by selecting a node and *all* of its descendants; the selected node is then the root of the subtree. It is convenient to refer to the "subtrees of a node," namely the subtrees whose roots are the children of that node.

The children of a single node are called **siblings** of each other. We will assume that these siblings are ordered from left to right; if we wish to ignore any such order, we refer explicitly to an **unordered tree**. It is usually more useful to maintain the order. The left to right order can be extended to other nodes in the tree: if X and Y are siblings with X to the left of Y, then we say that all descendants of X are to the left of all descendants of Y. For any two nodes in a tree, either one is to the left of the other or one is the ancestor of the other.

Some examples of the uses of trees:

1. The organization of a book into chapters, sections, subsections, etc. can be represented in a natural way as a tree. Here the relationship "X is a child of Y" means "X is contained in Y". Also notice that the interior nodes provide only information about this containment; all stored information (the actual text) is in the leaves.
2. An arithmetic expression can be represented as a tree; in fact many compilers do store and process expressions in tree form. See figure 5.2 for an example. Notice that the tree contains in its structure information about the order in which operations are to be performed; this expression is $(1 + 2) * (3 - 4)$.

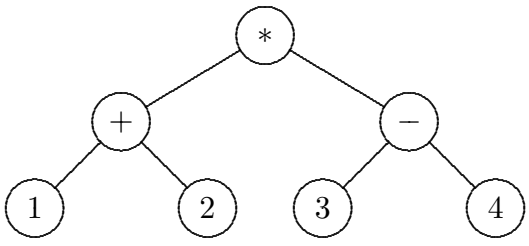


Figure 5.2 An arithmetic expression tree

Typical operations on trees are what you might expect after studying lists, except that there are more ways to move around, since the structure is more complicated. As usual, the exact operations may vary depending on the application, but here are the most common, in generic form.

1. Parent(T,X): returns a pointer to the parent of node X in T.
2. Child(T,X,k): returns a pointer to the kth child of X.
3. LChild(T,X): returns a pointer to the leftmost child of X.
4. RSib(T,X): returns a pointer to the next sibling of X in left to right order.
5. Retrieve(T,X): returns the contents (i.e., the information) in node X.
6. Create(T,T_1, ..., T_k): Creates a new tree T with the roots of T_1 through T_k as the children of its root.
7. Root(T): returns a pointer to the root of tree T.
8. Clear(T): turns T into the empty tree.

It is frequently desirable to list a tree in a linear order, or simply to **traverse** the tree (visit all its nodes) in some particular order, performing some sort of processing at each node. There are four standard orders in which the nodes may be listed, known as: **in-order**, **post-order**, **pre-order**, and **level-order** (this last term is less common than the first three). For a tree with one node, all four orders are of course the same: simply list the single node. Otherwise:

1. In-order: List the nodes of the leftmost subtree of the root in in-order, then list the root, then list the nodes of the remaining subtrees of the root (from left to right) in in-order.
2. Post-order: List the nodes of the subtrees of the root (left to right) in post-order, then list the root.
3. Pre-order: List the root, then list the nodes of the subtrees of the root in pre-order.
4. Level-order: List the nodes by level—first the root, then all children of the root, then all children of those nodes in left to right order, and so on.

These definitions are recursive and have very natural translations into actual recursive procedures. See figure 5.3 for an example. Because the recursive calls must include a node parameter, we use a simple "driver" function that merely gets the recursive function started on the root.

For example, these four orders for the expression tree above are: $1+2*3-4$, $12+34-*$, $*+12-34$, and $*+-1234$. The first two are infix and postfix notation expressions, but notice that the infix expression does not correspond to the expression represented in the

```
void inorder(tree& T) {
  inorder_rec(T,T.root());
}

void inorder_rec(tree& T, treenode X) {
  treenode p;
  int done;
  p=T.lchild(X);
  if (p)
    inorder_rec (T,p);
  process (T,X);
  while (p && (p=T.rsib(p))) {
      inorder_rec (T,p);
  }
}
```

Figure 5.3 In-order traversal

tree. The necessary parentheses must be inserted by appropriate additions to the basic in-order algorithm.

5.2 Implementations of Trees

It is possible to store each tree in its own array, or to use a linked representation, either cursor or pointer based. In practice a linked implementation is the most useful, just as for lists, though there are exceptions.

The simplest representation for trees employs a single link field in each node that points to the parent of the node. This captures the primary structure of the tree, but does not indicate the left to right order of the nodes in any way; thus this is really an implementation of unordered trees. Moreover, most of the common operations will be quite inefficient under this implementation. If trees are really needed in a particular application, it is almost always worth using more space to provide more link fields. The best general purpose choice is a total of three links: parent, leftmost child and right-sibling. With these three links, it is relatively easy to traverse the tree in any of the standard orders. Of course, if in a particular setting one or more of these links is never used, it may be left out.

If the maximum number of children of any node is known in advance, there is another option. Instead of the sibling links, a node can contain a link field for each of its possible children. This will generally involve more link fields of course, but it may make some algorithms easier to implement, e.g., if finding the kth child of a node is a frequent task. In particular this implementation is common if the maximum number of children is only two or three.

Notice that this implementation introduces a feature that was not specified for trees: it is possible for a particular subtree of a node to be empty. For example, a node may be able to have a maximum of five children, and may in fact have only three; these three may be the 'first,' 'third' and 'fifth' children, in the sense that the first, third and fifth link fields are in use. In the sibling link version of trees, this idea of a "gap" in the sequence of siblings doesn't arise. Depending on the use to which the trees are being put, this possibility of empty subtrees may be a nuisance or it may be a real help.

The most common form of tree in which the number of children is limited is the binary tree. Each node has at most two children, usually referred to as the left child and right child; the possibility that the left child does not exist while the right child does is built into the abstract specification of binary tree. The usefulness of this notion can be seen in the simple case of an arithmetic expression. Consider the infix expression $(a+b)*-c$; this can be diagrammed in tree form as in figure 5.4.

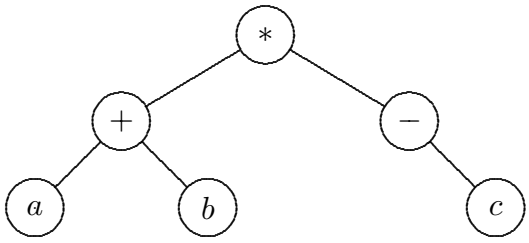


Figure 5.4 Expression with unary minus

If the left subtree of '−' were not recognized as existing (though empty), the in-order listing of this tree would be: $a+b*c-$, which is not correct infix notation. With the empty subtree in place however, the empty tree will be listed before the '−' (of course it will not result in any actual output), then the '−', then the c as desired.

5.3 Huffman Codes

A particularly nice application of binary trees is the Huffman encoding algorithm. In a typical text file on a computer, all characters are stored in one byte of memory, i.e., as an eight bit code. For a typical file this is bound to be inefficient since all characters take up the same amount of space, no matter how frequently they occur. It is better to use a shorter code for frequently occurring characters and thus reduce the overall size of the file.

Suppose a file contains 5 characters: *a, b, c, d, e*, with frequency of occurrence as listed in the table below. A 'standard' bit code might use three bits to represent each

letter. But a clever choice of codes can reduce the storage required for the file.

char	freq	stnd. code	clever code
a	12	000	000
b	40	001	11
c	15	010	01
d	8	011	001
e	25	100	10

It is easy to compute that the first code results in a storage requirement of 300 bits, the second only 220 bits.

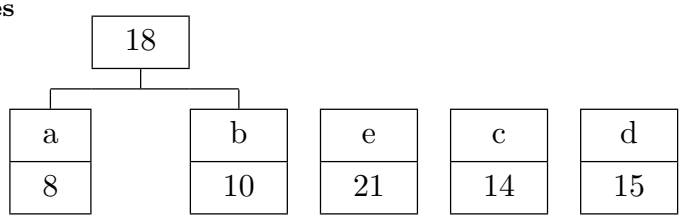
We must be careful when generating such clever codes, as it is essential that we be able to decode the file. Decoding a standard code is easy since we always know how many bits to expect for a character. In a variable length code we must guard against the following situation: We are reading bits and discover that we have the code for a character. How do we know that we shouldn't read further, discovering a longer code for some other character? We *can't* know what should be done unless we know in advance that no legal code can be extended to another legal code; such a good code is said to have the *prefix property.*

Of course we would like an algorithm that will figure out these codes for us, and ideally will discover a code with the smallest possible total file length. In fact the clever code above is not the best possible; a=1111, b=0, c=110, d=1110, e=10 is better, with a total length of 215. This is the best possible, and was generated by the Huffman algorithm.

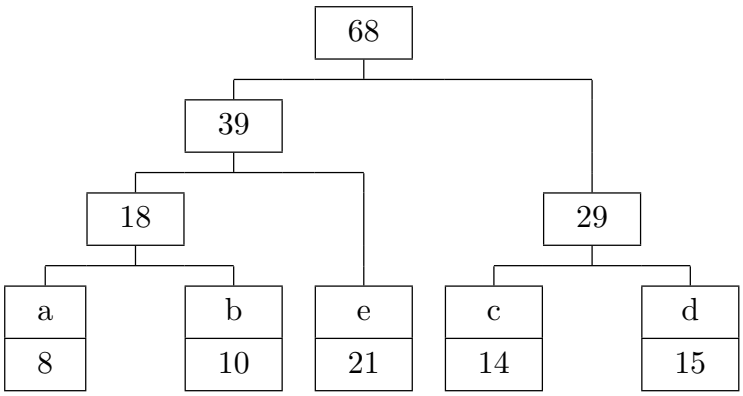
Here's how the algorithm works; we'll use a different example than the one above. You should try out the algorithm using the data from that example; if you don't get the same code, check to see that it also requires 215 bits. Start with a "forest" of trees, each tree a single node containing the character and the frequency with which it occurs in the file:

a	b	c	d	e
8	10	14	15	21

Now at each step, look through the forest and pick the two trees with the smallest weights in the root nodes. Combine these two trees into a new one by creating a new node to serve as the root, and set its weight field equal to the sum of the two weights from the trees being combined. The other field is not used for interior nodes. After one step the forest will look like:



Eventually the forest will consist of a single tree:



Notice that the order of the leaf nodes has been changed; this is purely for the sake of drawing the tree on paper and has no bearing on the algorithm.

Now the code for any letter can be read off the tree. We arbitrarily decide that left will mean 0 and right will mean 1. Start at the root and trace a path to, say, the node containing b. Translating left and right into 0 and 1, we get the sequence 001; this is the code for b. Notice that the code represented by this tree must have the prefix property, since there is no way to continue a path past a leaf node. We will see below that in fact this procedure gives the best possible code.

In reality, when we want to encode a letter, say b, we will have to start at the leaf node. Then we work our way up the tree keeping track of right and left; this will give us the sequence 100, the code for b written backwards. The code must therefore be generated and stored in an array, then written to the file in reverse order.

We also use the tree in the decoding process. Starting at the root we travel a path through the tree as indicated by a sequence of bits in the file. When we reach a leaf node, we know we have discovered the code for a character and we can simply write out the character that we find in the leaf node.

It is important to realize that the shape of the tree and the resulting codes depend on a particular file; this is not something that is done once and for all. Hence the encoded file must contain enough information about the tree to allow us to do this decoding. This tree information will not take up very much room, fortunately, and so except on very small files will not have a serious impact on the size of the file. The encoded version, even with

the tree information in it will typically be quite a bit smaller (by 20 to 40 percent) than the standard file.

In a certain restricted sense, the Huffman algorithm gives the best possible code, that is, results in the shortest encoded version of the information. The fine print is that we only compare the Huffman code to a code that replaces each character by a fixed, variable length binary pattern, and that these patterns have the prefix property, and we do not count the storage space required to "remember" the actual assignment of codes to characters.

Some kinds of data have features that allow other coding schemes to do better than Huffman. For example, some data may have long strings of identical components; such strings can be compressed by storing a count of the number of components, together with the value of a component. For example, a black and white image is typically stored in uncompressed form as a long binary string in which 0 stands for black and 1 stands for white. Instead of storing, say, 100 0-bits we could store "100B" in some reasonable form using much fewer than 100 bits. For example, the count could be stored in a 32-bit integer, and this could be followed by a single 0. Even better, since strings of 0s and 1s alternate, we could simply indicate the color of the first pixel, and then simply list the counts, knowing that each successive count represents the color different from the previous one. Further savings are possible by allowing the counts to be represented by a varying number of bits, typically a multiple of 8.

So within this restricted type of code, let's see why Huffman is the best. First note that any code of this type can be represented by a binary tree, so we can simply talk about binary trees. The quantity we are interested in is the length of the encoded information. We can express this in the context of the binary tree as follows. Suppose that we have a binary tree, with "weights" attached to the nodes as follows: Leaf nodes have some "arbitrary" weights, and then the weight of an interior node is the sum of the weights of its children. In the case of a binary tree representing a code, the weights of the leaves are the frequency counts of the corresponding characters. The length of the encoded information is then

$$\mathrm{wt}(T) = \sum_v \mathrm{d}(v)\,\mathrm{wt}(v).$$

Here $\mathrm{wt}(v)$ is the weight of node v, $\mathrm{d}(v)$ is the depth of v in the tree (the number of edges between the root and v), and the sum is over all leaf nodes v. Now our goal is to show that among all binary trees with the same weights on the leaf nodes, the Huffman tree has the smallest possible $\mathrm{wt}(T)$.

The key to the proof will be to know what happens to $\mathrm{wt}(T)$ when two nodes are swapped in T. Suppose that node v is swapped with node w, where neither of these is a descendant of the other. What will change in the sum defining $\mathrm{wt}(T)$? Only the depths

of those leaf nodes below v and w; thus, we need only pay attention to the contributions of these leaf nodes before and after the swap.

Let $\mathrm{d}(v) = d_1$ and let $\mathrm{d}_v(x)$ denote the depth of node x below node v, so that $\mathrm{d}(x) = d_1 + \mathrm{d}_v(x)$. The sums below are over all leaf nodes x below v. The contribution of the nodes below v is

$$\begin{aligned}\sum_x \mathrm{d}(x)\,\mathrm{wt}(x) &= \sum_x (d_1 + \mathrm{d}_v(x))\,\mathrm{wt}(x)\\ &= \sum_x d_1\,\mathrm{wt}(x) + \sum_x \mathrm{d}_v(x)\,\mathrm{wt}(x)\\ &= d_1 \sum_x \mathrm{wt}(x) + \sum_x \mathrm{d}_v(x)\,\mathrm{wt}(x)\\ &= d_1\,\mathrm{wt}(v) + \sum_x \mathrm{d}_v(x)\,\mathrm{wt}(x).\end{aligned}$$

Similarly, the contribution of the nodes below w is

$$\begin{aligned}\sum_x \mathrm{d}(x)\,\mathrm{wt}(x) &= \sum_x (d_2 + \mathrm{d}_w(x))\,\mathrm{wt}(x)\\ &= \sum_x d_2\,\mathrm{wt}(x) + \sum_x \mathrm{d}_w(x)\,\mathrm{wt}(x)\\ &= d_2 \sum_x \mathrm{wt}(x) + \sum_x \mathrm{d}_w(x)\,\mathrm{wt}(x)\\ &= d_2\,\mathrm{wt}(w) + \sum_x \mathrm{d}_w(x)\,\mathrm{wt}(x),\end{aligned}$$

where $d_2 = \mathrm{d}(w)$. Note that it is possible that v or w is a leaf node, in which case these expressions simplify considerably, since the "sums" are not really sums, or are "trivial" sums of a single term.

Now suppose we interchange nodes v and w, which of course means interchanging the subtrees below v and w as well. In the new tree, $d_1 = \mathrm{d}(w)$ and $d_2 = \mathrm{d}(v)$. The contributions of the subtrees are now:

$$\begin{aligned}\sum_x \mathrm{d}(x)\,\mathrm{wt}(x) &= \sum_x (d_2 + \mathrm{d}_v(x))\,\mathrm{wt}(x)\\ &= \sum_x d_2\,\mathrm{wt}(x) + \sum_x \mathrm{d}_v(x)\,\mathrm{wt}(x)\\ &= d_2 \sum_x \mathrm{wt}(x) + \sum_x \mathrm{d}_v(x)\,\mathrm{wt}(x)\\ &= d_2\,\mathrm{wt}(v) + \sum_x \mathrm{d}_v(x)\,\mathrm{wt}(x)\end{aligned}$$

and

$$\begin{aligned}\sum_x \mathrm{d}(x)\,\mathrm{wt}(x) &= \sum_x (d_1 + \mathrm{d}_w(x))\,\mathrm{wt}(x)\\ &= \sum_x d_1\,\mathrm{wt}(x) + \sum_x \mathrm{d}_w(x)\,\mathrm{wt}(x)\\ &= d_1 \sum_x \mathrm{wt}(x) + \sum_x \mathrm{d}_w(x)\,\mathrm{wt}(x)\\ &= d_1\,\mathrm{wt}(w) + \sum_x \mathrm{d}_w(x)\,\mathrm{wt}(x).\end{aligned}$$

If we subtract the first two contributions from the second two, we get the net change in the total $\mathrm{wt}(T)$:

$$d_2\,\mathrm{wt}(v) - d_1\,\mathrm{wt}(v) + d_1\,\mathrm{wt}(w) - d_2\,\mathrm{wt}(w) = (d_1 - d_2)(\mathrm{wt}(w) - \mathrm{wt}(v)).$$

If either $d_1 = d_2$ or $\mathrm{wt}(w) = \mathrm{wt}(v)$ then the net change is zero; otherwise we can tell whether the net change is positive or negative by examining the sign of the factors. Figure 5.5 illustrates what we've done.

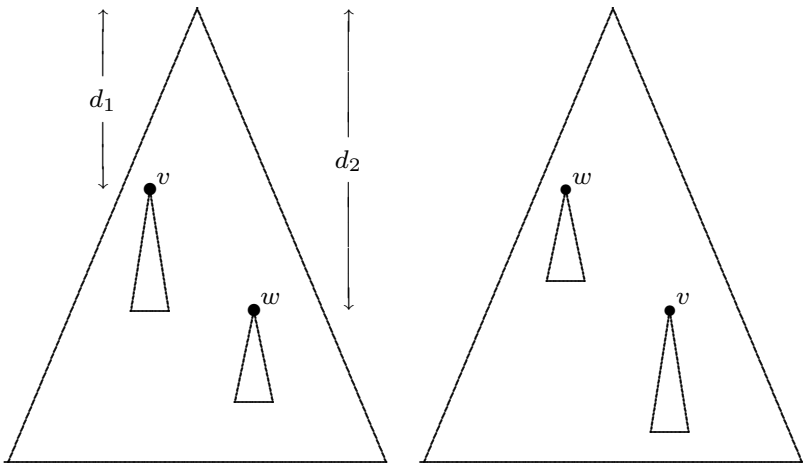


Figure 5.5 Swapping two nodes.

Now for the main proof. The proof is by induction on the number of leaf nodes in the Huffman tree; call this number n. Recall how proof by induction works: We check that the desired statement is true for the smallest "reasonable" value of n. Then we assume we know the statement is true up through some value $n - 1$ and prove that it is true for n. The smallest reasonable value for n is 2, that is, 2 leaf nodes and one interior node. There is only one way to put two leaf nodes into a binary tree, so the Huffman tree is the best possible.

Now suppose that we know that among all binary trees with up through $n-1$ leaf nodes, the Huffman tree is best possible. Next suppose that T_h is a Huffman tree with n leaf nodes and T_m is a tree with the same leaf weights and the minimum possible value for $\text{wt}(T_m)$. We want to show that $\text{wt}(T_h) = \text{wt}(T_m)$. We know that the Huffman tree has in it two leaf nodes of smallest possible weight as siblings; call them v and w. Where are v and w in T_m? If they happen to be siblings, that's great. If not, we want to produce a new tree T'_m with $\text{wt}(T_m) = \text{wt}(T'_m)$ and in which v and w are siblings. So suppose that in T_m, s is the sibling of v and t is the sibling of w. Note that s and t might be leaf nodes or they might be interior nodes.

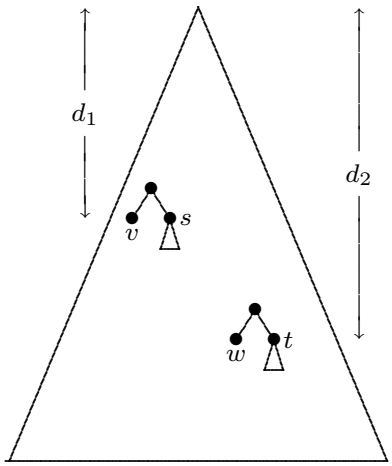


Figure 5.6 Make v and w siblings.

We may assume that $d_1 \leq d_2$, as shown in figure 5.6. If not, we can just rename the nodes so that w is at least as deep in the tree as v.

Suppose we try to swap t with v. The net change in total tree weight is $(d_1 - d_2)(\text{wt}(t) - \text{wt}(v))$, and this must be greater than or equal to zero, since T_m has minimum possible total tree weight. If $d_1 = d_2$ then the net change is zero, so we go ahead and do the swap. What if $d_1 < d_2$? Note that $\text{wt}(v) \leq \text{wt}(t)$, because if $\text{wt}(v) > \text{wt}(t)$ the Huffman algorithm would not have picked v and w to put together first. Then $wt(t) - \text{wt}(v) \geq 0$, so $(d_1 - d_2)(\text{wt}(t) - \text{wt}(v)) \leq 0$. But this net change cannot be less than zero, so it must be equal to zero, so we can go ahead and do the swap.

In any case, we have a tree T'_m (which might actually be T_m if v and w were siblings to start with) in which v and w are siblings and $\text{wt}(T_m) = \text{wt}(T'_m)$.

Now comes the induction part of the proof. Suppose we remove the nodes v and w in each tree, so that the common parent of the two becomes a leaf node in a new tree. The new

tree T'_h is itself a Huffman tree, because the algorithm proceeds in the same way whether or not the parent of v and w is a leaf node, and T'_h has $n-1$ leaf nodes. Also, the new tree T''_m is a tree that has the same leaf node weights as T'_h. By the induction hypothesis, it must be that $\text{wt}(T'_h) \leq \text{wt}(T''_m)$ (we can't say that $\text{wt}(T'_h) = \text{wt}(T''_m)$ because we don't know that T''_m is best possible). Also, it is easy to see that $\text{wt}(T'_h) = \text{wt}(T_h) - \text{wt}(v) - \text{wt}(w)$ and likewise $\text{wt}(T''_m) = \text{wt}(T'_m) - \text{wt}(v) - \text{wt}(w)$. So now

$$\begin{gathered}
\text{wt}(T_h) - \text{wt}(v) - \text{wt}(w) = \text{wt}(T'_h) \leq \text{wt}(T''_m) = \text{wt}(T'_m) - \text{wt}(v) - \text{wt}(w) \\
\text{wt}(T_h) - \text{wt}(v) - \text{wt}(w) \leq \text{wt}(T'_m) - \text{wt}(v) - \text{wt}(w) \\
\text{wt}(T_h) \leq \text{wt}(T'_m)
\end{gathered}$$

so, since $\text{wt}(T'_m)$ is as small as possible, $\text{wt}(T_h) = \text{wt}(T'_m)$, and $\text{wt}(T_h)$ is also as small as possible.

6

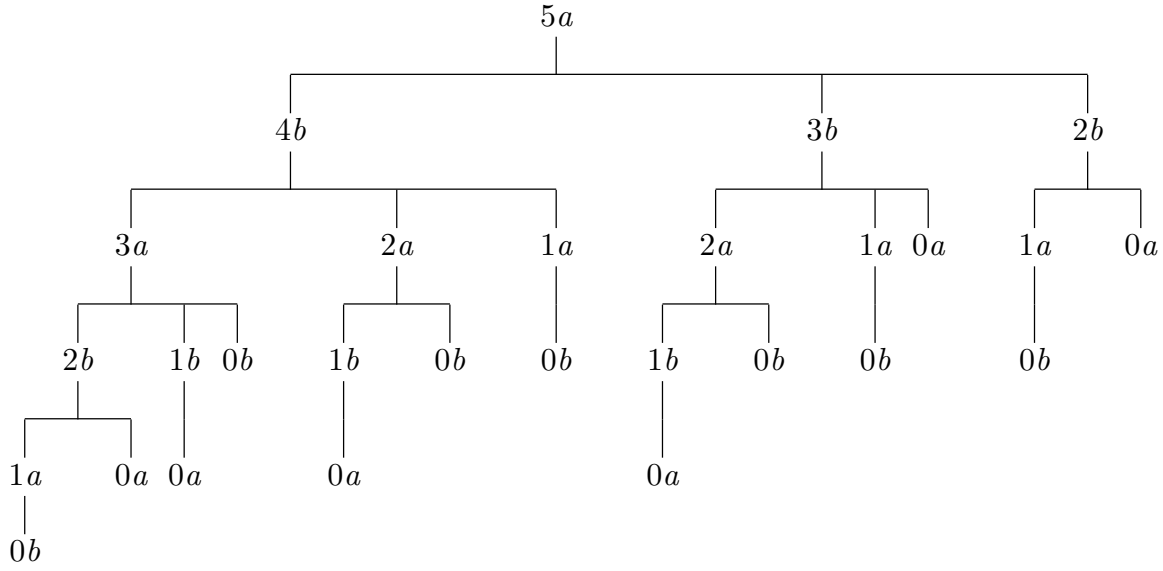
Game Trees

A substantial amount of effort has been devoted to developing programs that play games, because this is intrinsically interesting and because games seem to require intelligence when played by humans, and yet games present a restricted environment by comparison to most human activity. Game playing has thus seemed an ideal place to start teaching computers to exhibit intelligent behavior.

One of the most fruitful approaches to the representation of games uses a tree, called a **game tree**. The nodes in such a tree represent the state of the game after some number of moves. The play of the game corresponds to traveling a path through the tree from the root to a leaf. Such a tree contains *every* possible instance of the game and hence is usually very large.

The nodes of a game tree must contain enough information to fully describe the state of the game. This information is called the **configuration** of the game and will usually have some obvious form given a particular game. For example, the states of chess, checkers and nim can be easily represented by appropriate arrays (although there may be clever ways to represent the configurations that make the program more efficient). The configurations in leaf nodes correspond to the end of play and are called **terminal configurations**. The nodes are arranged into a tree by letting the children of a node contain all configurations that can be reached in one legal move from the configuration in that node. Notice that there might be many nodes with the same configuration, one for each possible line of play that results in the configuration.

Consider a very simple game: There is a pile of 5 stones. Each player in turn removes 1, 2 or 3 stones; the player who picks up the last stone loses. We may represent the configurations by a single integer value, and the tree would then look something like the following (the nodes are shown as an integer for the number of stones, and an a or b depending on whose turn it is).



It is easy to determine by examination that the first player cannot win this game if the second player chooses the best available moves. (For this particular game, with any number of stones, there is a better way to discover and use a winning strategy, but for most interesting games that is not the case.) We want to formulate an algorithm that can examine such a tree and discover who can win and how. To do this, we will assign to each node some measure of what the configuration is worth to the first player, player "a."

Suppose we have figured out the values of all the children of some a node. Since it is a's turn to move, he is free to pick the move that results in a configuration of highest possible worth. The value to a of the current configuration is therefore the same as the maximum of the values of all configurations in children of the current node. If we let $V(X)$ denote the value of node X then we have, if X is an a node:

$$V(X) = \max\{V(Y) | Y \text{ is a child of } X\},$$

assuming that we know the values of the children of X, which are b nodes.

Now suppose we want to assign a value to some b node. Since b is free to pick the best move, the value to a of the node is the value of the worst (for a) configuration that b can choose. Thus:

$$V(X) = \min\{V(Y) : Y \text{ is a child of } X\}$$

if X is a b node.

To use this scheme, we must start by assigning values to leaf nodes in some other way; since leaf nodes represent the end of the game, this is easy. In the example pictured above, we start by assigning the value 1 to all leaf nodes containing "0a," and the value -1 to other leaf nodes. Now move values up the tree until the root gets a value; 1 means a can win, -1 means b can win. The values of the nodes also tell us how each player should move if possible: a should opt for nodes with value 1, b for nodes with value -1.

In general there may be more than two values that might be assigned to nodes; for example, the number 5 in a leaf node could mean that upon reaching that terminal configuration, one of the players wins \$5. For discussions about games in general we simply assume the existence of some evaluation function $E(X)$ that assigns values to leaf nodes X. We then define $V(X)$ on all nodes in the tree as:

$$V(X) = \begin{cases} E(X), & \text{if } X \text{ is a leaf node;} \\ \max\{V(Y) | Y \text{ a child of } X\}, & \text{if } X \text{ is an } a \text{ node;} \\ \min\{V(Y) | Y \text{ a child of } X\}, & \text{if } X \text{ is a } b \text{ node.} \end{cases}$$

This is called the **minimax** procedure for determining the value of a node X. When it comes time to program this, there is a neat trick to avoid keeping track of when to 'min' and when to 'max.' Replace $E(X)$ by an $E'(X)$:

$$E'(X) = \begin{cases} E(X), & \text{if } X \text{ is an } a \text{ node;} \\ -E(X), & \text{otherwise.} \end{cases}$$

Now replace $V(X)$ by the following $W(X)$:

$$W(X) = \begin{cases} E'(X), & \text{if } X \text{ is a leaf node;} \\ \max\{-W(Y) | Y \text{ is a child of } X\}, & \text{otherwise.} \end{cases}$$

You should convince yourself that W is the same as V on all a nodes, and is just $-V$ on b nodes. Thus, W computes essentially the same information via a more compact algorithm. It is now an easy matter to add this algorithm to a procedure that traverses the tree in post-order (see figure 6.1).

It should be clear from the simple example above that the tree for any reasonably complicated game will be quite large, too large for even a computer to deal with. For example the tree for chess is estimated to have in excess of 10^{100} nodes. Even at a rate of one trillion nodes per second (an exceptionally generous assumption) it would take more than 10^{80} **years** to traverse such a tree.

Perhaps surprisingly, this does not mean the game tree is worthless if one wants to automate game playing. It does mean that only a small portion of the tree can be examined

```
int W(treenode X) {
  int temp;
  treenode q;
  if ( lchild(X) == NULL ) {
    return(E'(X));
  } else {
    q = lchild(X);
    temp = -W(q);
    while ( rsib(q) != NULL ) {
      q = rsib(q);
      temp = max(temp,-W(q))
    }
    return (temp);
  }
}
```

Figure 6.1 Tree evaluation using maxima only

at any time. Near the beginning of the game the "truncated" subtree we are able to "see" will probably contain no terminal nodes; certainly most of the leaf nodes in the subtree will in reality be interior nodes in the full tree. The propagation algorithms V and W will still work, but to get started we must be able to assign values to the leaf nodes (of the subtree). We thus need a way to estimate the value of a node given the configuration. Once the leaf nodes of the subtree are assigned values in this way, we may then proceed as before. A great deal of time and effort have been spent on developing good estimates of the value of a configuration in particular games.

Even with a reasonably good evaluation function, the deeper into the tree we can look the better we would expect to play. A human player does of course look ahead, but not in the exhaustive way that the algorithm does. A skilled chess player is able to examine only those lines of play that have some promise. Considerable effort has been devoted to discovering ways in which the subtree may be "pruned," so that we do not in fact need to examine the entire subtree. In this way we can increase the depth to which we look along those paths that are not pruned off. We will look at one of the simplest ways in which pruning may be done. It does not depend in any way on the game being represented, so one might expect that the results, while good, are not amazing.

To each a node in the tree we will attach an **alpha-value**, and to each b node a **beta-value**. These values are simply the 'current' values of the nodes as the minimax procedure runs. That is, the function V computes minimums or maximums in stages, keeping the smallest or largest value it has seen so far in its traversal through the children of a node. This running maximum or minimum value is the α-value or β-value respectively. We will

prune the tree according to the relative sizes of some α-values and β-values. There are two pruning rules, corresponding to the two types of node.

α-cutoff: If the β-value of a b node is less than or equal to the α-value of its parent, then we may ignore remaining children of the b node.

β-cutoff: If the α-value of an a node is greater than or equal to the β-value of its parent, then we may ignore remaining children of the a node.

Let's examine α-cutoff more closely; β-cutoff is nearly the same. The α-value of the parent node is the current maximum of that node's children. The b node is the child currently under scrutiny. By looking at children of the b node we update a running minimum, the β-value. If ever the β-value is less than the parent's α-value it is pointless to continue. The β-value will only get smaller as we look at more of the b node's children, but already we know that it will have no effect on the α-value of its parent since α-values may only increase.

This process depends not only on the values of the nodes in the tree but on the order in which they are visited; the best way to gain an understanding of how and why it works is to use the cutoff rules by hand on a relatively small tree.

There are a number of modifications that can be made to this basic idea to make the programming go smoothly. In particular we would like to fit it to our basic propagation algorithm that only takes maxima. We replace the distinct α and β values by an $\alpha\beta$-value, which is the running maximum as used by the W function. It is also convenient to avoid treating the leftmost child as a special case merely to initialize the $\alpha\beta$-value. Instead we assume that we have a constant '$-\infty$' (called `-INF` in the code) that is smaller than any legal node value. If we initialize the running maximum to this constant, then at the first step it will actually be replaced by the value of the leftmost child node. The final algorithm will look like figure 6.2.

Finally, there is one slight improvement that can be added at little cost to the α-β pruning algorithm. Instead of initializing the running maximum as `temp = -INF`, we use for an initial value the $\alpha\beta$-value of the grandparent of X. This can result in the pruning of grandchildren of X that otherwise would not be pruned. Thus this rule involves nodes at five levels of the game tree, making it somewhat more confusing than the basic cutoff scheme. If you play around with a few examples you should see what's going on.

```
int W(treenode X,int ab) {
// Here "ab" is the alphabeta value of the
// parent of node X which controls whether
// we look at more children of X.
  treenode q;
  int temp;
  bool prune = false;
  if ( lchild(X) == NULL ) {
    return(E'(X));
  } else {
    q = lchild(X);
    temp = -INF;
    while ( q != NULL && !prune ) {
      if temp >= -ab
        prune = true;
      else {
        temp = max(temp,-W(q,temp));
        q = rsib(q);
      }
    }
    return (temp);
  }
}
```

Figure 6.2 Game tree evaluation using α-β pruning

7

Sorting Algorithms

We now investigate a very common and typically time consuming task, that of sorting a number of values into their correct order. This will not involve any radically new data structures; we will concentrate on using ones we already know to implement efficient algorithms.

We generally are not simply interested in sorting values themselves but records containing a variety of information fields, one or more of which contain the values to be sorted. For example, the records may contain personnel data including names, social security number, age. We may want a list by last name, with order determined by first name within a last name group; or we may want a list by age or social security number. A field that controls the sorting procedure is called a **key** for the record. For simplicity, we assume that the records are simply integers, but in practice this will not always be the case, and the code will be somewhat more complicated.

Often the number of records to be sorted is enormous and even the best sorting algorithms will take a very long time to complete the task. We thus need to estimate in some way the running time behavior of different algorithms in order to make a reasonable choice. This **analysis of algorithms** is a difficult and complex subject; we will see only a few basic methods for estimating the running times of algorithms.

We begin by looking at three simple sorting algorithms that are reasonably good for small collections of records. They are all similar in design. To begin with we assume that the set of records is small enough that all records can be stored in an array; such sorting

algorithms are known as **internal**. When there are too many records to fit in internal storage, **external** sorting algorithms are required.

7.1 Bubble sort

Imagine the array drawn vertically with the smallest index at the bottom. The idea is to make repeated **passes** up the array, "bubbling" the light (in this case, "light" means "large") key values to the top. On each pass the next lightest value will appear in the proper place. Assuming the array is indexed `[0..n-1]`, we require $(n-1)$ passes to guarantee that the array is sorted. The bubbling process compares adjacent values and insures that the larger of the two is on top. By considering successive adjacent pairs starting at the bottom of the array, each pass bubbles the next largest element to the correct location. After doing three passes the top three values are correct, so subsequent passes stop without even looking at the top three pairs. See figure 7.1.

```
void bubblesort(int A[], int n) {
  int top,pair;
  for (top = n-1; top > 0; top--) {
    for (pair = 0; pair < top; pair++) {
      if (A[pair] > A[pair+1])
        swap(A[pair],A[pair+1]);
    }
  }
}
```

Figure 7.1 Bubble sort

Let's see what we can say about how long this will take to sort n items. The time varies depending on the initial order of course; we do a **worst case** analysis, so we get an estimate of the longest this algorithm might take. For all the simple algorithms we examine, the worst case running time is (essentially) the same as the average running time. Average case analysis is typically more difficult; we will only attempt such an analysis for one algorithm that is better than these simple ones.

The *if* statement will take at most some constant amount of time, say a units of time. Of course, depending on the value of the boolean expression the time taken can vary, but there is some small maximum amount of time it can take. The inner for-loop executes *top* times, so the total running time for the loop is

$$(top)(\text{amount of time taken in the loop}).$$

The amount of time in the loop is

$$a + (\text{amount of time spent controlling the loop}).$$

The latter is no worse than, say, b units of time. Hence the inner loop takes $(top)(a+b)$ time in the worst case. Now the outer for loop executes $n-1$ times, but the running time for each varies. The total time T is no worse than the sum:

$$\sum_{top=1}^{n-1} \big((top)(a+b)+c\big),$$

where c is the maximum amount of time needed to control the loop on each iteration. To all this we must add the amount of time necessary to start up the algorithm; this is some constant amount of time and is necessary in all cases, so we can ignore it for the purpose of comparing different algorithms.

We can rearrange this sum:

$$\begin{aligned} T &\le (a+b)\sum_{t=1}^{n-1} t + \sum_{t=1}^{n-1} c \\ &= (a+b)\big(1+2+\cdots+n-1\big) + (n-1)c \\ &= (a+b)(n)(n-1)/2 + (n-1)c \\ &= \left(\frac{a+b}{2}\right)n^2 + \left(c - \frac{a+b}{2}\right)n - c \end{aligned}$$

Thus, the running time of *bubblesort* is no worse than a quadratic function of n. The constants a, b and c are essentially impossible to estimate: they depend on the language used, the quality of the compiler and the underlying machine. The important feature of this estimated running time is the n^2. When n is large, the n^2 term will be the dominant factor in the actual running time. Because of this term, doubling n will make the running time go up by a factor of about four. In the long run this is the most significant feature of the algorithm. We sum this up by saying that *bubblesort* is a $O(n^2)$ algorithm (read "big oh of n squared") in the worst case. It is also $O(n^2)$ on the average.

7.2 Insertion sort

This has much the same flavor as bubblesort but some noticeable differences (see figure 7.2). It makes $(n-1)$ passes over the array; after k passes the first k items in the array are in the correct order, but they may not be the smallest k items in the whole array.

```
void insertionsort(int A[], int n) {
  int j,k;
  for (k=1; k < n; k++) {
    j=k;
    while (j>0 && A[j] < A[j-1]) {
      swap(A[j],A[j-1]);
      j--;
    }
  }
}
```

Figure 7.2 Insertion sort

The variables have different names, but the procedure is still basically two nested loops. In bubble sort the body of the inner loop was executed *top* times no matter what. Insertion sort is somewhat smarter: in most situations the while loop will execute fewer than k times. Unfortunately, the worst case time is still $O(n^2)$, since in the worst case the while loop will execute the maximum number of times and will hence behave just like the inner for loop of the bubble sort algorithm. It can also be shown in a precise way that the average running time for insertion sort is $O(n^2)$. In particular cases insertion sort may be faster than bubble sort, but in all fairness it must be mentioned that it is easy to jazz up bubble sort to avoid useless passes.

7.3 Selection Sort

Finally we look at selection sort. This too is $O(n^2)$ and the analysis is nearly identical to bubble sort and insertion sort. It does have one feature of interest. In the previous sorts, a record was moved to the proper place by being swapped through every position between its starting location and final location. Especially when the records are large, this may be expensive. Selection sort moves the swap step out of the inner loop, so the number of swaps is limited to the number of times that the outer loop executes, which is $O(n)$.

In practice this feature of selection sort is not particularly important. If the records are large, we can keep an array of pointers to records and simply swap pointers instead of whole records. Then if necessary we can actually move each record to its proper place when the sorting is completed.

7.4 Quicksort

In practice, $O(n^2)$ behavior is unacceptable, unless the number of records to be sorted is small. There are two obvious questions at this point: Is there an algorithm that is

```
void selection(int A[], int n) {
  int i,j,min;
  for (i=0; i<n-1; i++) {
    min=i;
    for (j=i+1; j<n; j++) {
      if (A[j]<A[min]) min=j;
    }
    if (i != min) swap(A[i],A[min]);
  }
}
```

Figure 7.3 Selection sort

faster than $O(n^2)$? Is there one which at least on the average is faster than $O(n^2)$? The answers to both are yes. There is an excellent algorithm that is $O(n^2)$ in the worst case but $O(n \log n)$ in the average case. There is also an algorithm which is $O(n \log n)$ in both the worst and average cases. Surprisingly the former is the better algorithm in practice. Due to the complexity of the bookkeeping involved, the second algorithm runs slower on the average than the first.

Quicksort is a recursive sorting algorithm whose worst case running time is $O(n^2)$ but whose average time is $O(n \log n)$. In outline, it works like this: We pick a **pivot value** from among the key values in the records; ideally we would like this pivot value to be the median of the key values, but we won't be able to guarantee this. We then divide the records into two collections: those with key value less than or equal to the pivot and those with key value greater than the pivot. (If the pivot is close to the median, these two collections will be about the same size, which is what we want.) We rearrange the array so that the former occupy the low end of the array and the latter the high end, with the pivot itself in between the two collections. Notice now that if we sort the two collections separately in their respective subarrays that the entire array will be sorted as well. Thus we simply call *quicksort* recursively on the two subarrays.

How should we estimate the median, i.e., pick the pivot? It is of course possible to find the median precisely, but it turns out to be too time consuming to do so. Picking an element at random is in some ways the best, but this may be impractical for a variety of reasons; picking the median of a small number of elements in the subarray, say 3 or 5, also works well.

Now we must rearrange the records so that the low key values all come at the low end of the subarray; suppose that the subarray includes positions i through j. It simplifies matters if we move the pivot value itself to the left end of the subarray. Then we start with cursors $l = i + 1$ and $r = j$. These cursors move toward each other until they meet

and cross. We insure that at all times the records to the left of l have key values less than or equal to the pivot and the records to the right of r have keys greater than the pivot. When the cursors cross the entire subarray will be properly arranged. The rearrangement may be broken down into the following steps:

1. Move l to the right past any key values less than or equal to the pivot (these keys are already in an acceptable position); move r to the left past any key values greater than the pivot.
2. If $l > r$ then stop.
3. Swap `A[l]` with `A[r]`.
4. goto 1.

When this process halts, $r + 1 = l$ and r points to the record with highest array index that has key value less than or equal to the pivot. If we now swap the pivot at position i with the item at position r, the pivot will be in the correct place, and we can recursively sort the subarrays in positions i through $r - 1$ and l through j. The final code would look something like figure 7.4.

How long will quicksort take in the worst case? There are two important components to examine: the nested *while* loops that do the partitioning of the subarrays, and the depth of the recursion.

A casual analysis of the partitioning phase looks bad: the outer loop can execute something like $m = j - i + 1$ times, and either of the inner loops can execute about m times as well, so an upper bound is m^2—just for rearranging at one level! A more careful analysis shows that the time is not nearly this bad. Think of the cursors `l` and `r` as being "responsible" for certain positions in the subarray; that is, if `l` points to a particular position, `l` is charged with making sure that the item there is less than or equal to the pivot value. It only takes `l` a small constant amount of time to accomplish this: either a simple check, or a check and swap. Likewise for `r`. Despite the loop structure, it takes only a constant amount of time per subarray position to do the partitioning phase of the function, so the time is $O(m)$. If n is the total number of items to sort, then the total time spent partitioning the entire array at a single recursive depth is $O(n)$. This means that the total time spent sorting the array is no worse than $O(n \cdot (\text{depth of the recursion}))$. Unfortunately, the maximum depth is about n: if the pivot is consistently chosen to be the smallest or largest element in each subarray, every subarray will be split into a very tiny subarray, and a subarray almost as large as itself, so the recursive depth will be $O(n)$ and the total time will be $O(n^2)$..

The average running time of *quicksort* is much better, $O(n \log n)$. It is not hard to see why we might hope that this is the case. If on the average the subarrays produced by

```
void quicksort(int i, int j)
{
  int l,r,m;
  if (i+1 < j) {
    // There are at least 3 items in this subarray.
    l=i+1; r=j; m=(i+j)/2;
    if (A[i]<A[m]) swap(A[i],A[m]);
    if (A[j]<A[i]) swap(A[j],A[i]);
    if (A[i]<A[m]) swap(A[i],A[m]);
    // The median of A[i], A[m], A[j] is now at position i.
    while (r>l) {
      while ( l <= j && A[l] <= A[i] )
        l++;
      while ( A[r] > A[i] )
        r--;
      if ( l < r ) swap (A[l],A[r]);
    }
    swap(A[i],A[r]);

    quicksort(i,r-1);
    quicksort(l,j);
  } else
    // If there are two items in the subarray, sort them.
    if (i<j && A[i]>A[j]) swap(A[i],A[j]);
}
```

Figure 7.4 Quicksort

partition are the same size, then the depth of the recursion is only about $\log_2 n$, so the total time is $O(n \log n)$.

What do we really mean by "average" running time? Each original ordering of the records to be sorted will result in some different sequence of steps during the execution of the algorithm. We want to average the running time over all possible initial orderings of the key values. To make the analysis proceed smoothly, we make some assumptions: First, we assume that all key values are different; this simplifies the analysis, but duplicates do not change the running time significantly. We also assume that the order of any subarray during processing is random, i.e., that any such order is as likely as any other. This depends on the exact method for choosing the pivot and doing the rearrangement; subtle non-random bias is often very difficult to detect. The method we have chosen should be very good in this respect. Likewise, we need to assume that all possible split points (that is, the final value of `r` during the partitioning phase) are equally likely.

Denote by $T(n)$ the average time taken by *quicksort* on input of size n. Then $T(n)$ is at most some constant times n (for the partitioning phase prior to the recursive calls) plus

the average time taken in the two recursive calls:

$$\begin{aligned} T(n) &\leq Kn + \frac{1}{n}\sum_{i=0}^{n-1}\big(T(i) + T(n-i-1)\big) \\ &= Kn + \frac{1}{n}\sum_{i=0}^{n-1} T(i) + \frac{1}{n}\sum_{i=0}^{n-1} T(n-i-1) \\ &= Kn + \frac{2}{n}\sum_{i=0}^{n-1} T(i) \end{aligned}$$

We're interested in an upper bound for $T(n)$, so let's assume the worst, namely

$$T(n) = Kn + \frac{2}{n}\sum_{i=0}^{n-1} T(i),$$

and see if we can figure out how $T(n)$ behaves, that is, find a 'nice' function $f(n)$ so that $T(n) = O(f(n))$.

This is an example of a recurrence relation, i.e., a relationship between the values of a function at different arguments. The recurrence relation allows us to compute as many values of $T(n)$ as we want, but it's not obvious what the function $T(n)$ is 'really' like. Finding a direct expression for $T(n)$ is called *solving* the recurrence relation. Recurrence relations have been studied extensively so there are many well known techniques for solving them. We'll need a few 'tricks' here—though they are all so useful they really deserve to be called 'methods'.

The first problem is the summation—it's too nasty, so we'll get rid of it. Multiply through the recurrence relation by n to clear the fraction, then replace n by $n-1$:

$$\begin{aligned} (n)T(n) &= Kn^2 + 2\sum_{i=0}^{n-1} T(i) \\ (n-1)T(n-1) &= K(n-1)^2 + 2\sum_{i=0}^{n-2} T(i). \end{aligned}$$

Now subtract these two equations, noticing that the sums are almost identical so that there is lots of cancellation; solving for $T(n)$ we get

$$T(n) = 2K - \frac{K}{n} + \left(\frac{n+1}{n}\right) T(n-1).$$

This is a much nicer recurrence relation, but it's still not trivial to solve it. We need a more familiar trick this time: substitution (much like the substitution technique for integration). Replace $T(n)$ by $(n+1)x_n$:

$$(n+1)x_n = 2K - \frac{K}{n} + \left(\frac{n+1}{n}\right)(n)x_{n-1}.$$

Solving for x_n:

$$x_n = \frac{2K}{n+1} - \frac{K}{n(n+1)} + x_{n-1} = K\frac{2n-1}{n(n+1)} + x_{n-1}.$$

Now you should be able to see quite easily that

$$x_n = x_1 + K\sum_{i=2}^{n} \frac{2i-1}{i(i+1)},$$

so that

$$T(n) = (n+1)x_1 + K(n+1)\sum_{i=2}^{n} \frac{2i-1}{i(i+1)}.$$

Note that $x_1 = T(1)/2$, which is just some constant—half the length of time it takes quicksort to sort a one-item array. What about that sum? It's still not obvious how $T(n)$ behaves, although at least we don't have T expressed in terms of itself any more. By drawing a suitable picture you should be able to convince yourself that

$$\sum_{i=2}^{n} \frac{2i-1}{i(i+1)} \le \sum_{i=2}^{n} \frac{2i}{i^2} = 2\sum_{i=2}^{n} \frac{1}{i} < 2\int_1^n \frac{1}{x}\,dx = 2\ln(n),$$

so finally we have

$$T(n) \le (n+1)x_1 + 2K(n+1)\ln(n) = O(n\log(n)).$$

This means that on the average, quicksort takes much less time to sort an array than do the simpler sorting methods: $n\log(n)$ is much smaller than n^2 when n is even modestly large. Of course, we don't know exactly how much better quicksort is because we don't know the values of the constants involved in the various sorting algorithms. In practice, however, quicksort fully lives up to expectations: not only is the average running time much smaller, the algorithm almost never strays very far from its theoretical average, as long as the choice of pivot is effectively random.

It is possible to improve the performance of quicksort by careful coding. The most important improvement is to switch to a simpler method of sorting when the subarray

is very small, because on very small subarrays insertion sort, for example, is faster than quicksort. The best method is to have quicksort simply ignore subarrays that are small, then to perform a single insertion sort on the entire array after quicksort finishes. Because of the sorting accomplished by quicksort, the inner loop in insertion sort will never execute many times, so the insertion sort will take time that is $O(n)$. Careful theoretical and empirical studies have shown that subarrays of length less than 10 should be left to the insertion sort stage.

Another improvement is to replace most of the array references by pointer references. As `l` moves through a subarray, it is always incremented by 1. It is more efficient to perform a pointer increment `p++` and then refer to the array element as `*p` than to increment `l` and then perform the arithmetic involved in an array reference like `A[l]`.

7.5 Heapsort

The next sorting algorithm we look at is called *heapsort* and has average and worst case running times that are $O(n \log n)$. The algorithm uses a special type of binary tree called a heap. In addition to being a binary tree, a heap satisfies the following two properties:

1. The tree is 'full," i.e., all levels (except possibly the last) in the tree contain the maximum possible number of children. The last level has all its nodes as far to the left as possible.
2. The key value of any node is less than or equal to the key values of its children.

Notice that the smallest key value is at the top of the heap (making the first step in sorting obvious) and the depth of the tree is approximately $(\log n)$.

The outline of the algorithm is easy:

while (*there are more records*) {
  read a record and add it to the heap
}
while (*the heap is not empty*) {
  remove and store the root record
  rearrange the remaining nodes into a new heap
}

Each of the while loops will execute n times; inside each loop there is a single potentially expensive operation. In the first loop, we must insert a new record into the heap and maintain the "heapness;" in the second loop we must combine two heaps (formed when the root is removed from a single heap) back into a single heap.

Consider the problem of adding a new record to a heap. We must keep the tree full, which we can easily do by adding a new node for the new record (call it `N`) in the last level

of the tree as far as possible to the left (this may actually mean opening a new level if the current last level is full). Now we must rearrange the records within the existing nodes so as to satisfy property (2). The only potential problem is that the new record has a smaller key value than its parent (call the parent `X`); if so we merely swap `N` with `X`. Notice that `N` is now the parent of `X`, which is OK, and possibly the parent of a node `Y`, which was the other child of `X` before the swap. Since `N.key` $<$ `X.key` $\leq$ `Y.key`, property (2) is satisfied for the `N`–`Y` pair. The only problem now is that `N.key` might still be less than the key of its parent; another swap will take care of that and again will not introduce any new problems, except between `N` and its new parent. As soon as `N` becomes the child of a node with smaller key value (or reaches the root) the process stops and we have a heap.

Now suppose we have removed the record in the root. Our first move is again to create a tree with the proper shape and then worry about the ordering. We take the record (say `N`) from the node farthest right in the last row and put it in the vacant root node. Suppose the children of `N` are now `X` and `Y`; then `N.key` may be larger than one or both of `X.key` and `Y.key`. In this case, we swap `N` with the record (`X` or `Y`) with the smaller key; after this swap, the only place property (2) might not be satisfied is between `N` and its new children. We continue until `N` has no children or until property (2) is satisfied for both children.

Now because we have forced the tree always to have depth approximately equal to $(\log n)$, the maximum running time for the body of each while loop is $O(\log n)$, and hence the total running time is $O(n \log n)$. Although the worst case behavior of QUICKSORT is n^2, that limit is never approached in practice; on the basis of the analysis we have done there is no reason to choose either *heapsort* or *quicksort* over the other. Actual testing indicates that *quicksort* is faster by a small factor and hence is the algorithm of choice for large data sets.

It is possible to improve *heapsort*. Notice that both while loops take the same $O(n \log n)$ time. This is unavoidable in the second loop but the first can be improved to $O(n)$, a substantial change. Instead of adding to the heap one record at a time, we can first put all the records into a full binary tree, without worrying about property (2). Then start at the bottom and consider all the subtrees with roots in the next to last level of the tree, each with one, two or three nodes. Convert each such subtree into a heap; this takes at most one swap per subtree. Then move up a level, and convert all the subtrees with roots in that level into heaps; each will take at most two swaps to rearrange (note that the algorithm for rearranging a heap after removal of the root record works for rearranging these subtrees into heaps). In this way we work our way to the top, until we finally create a heap out of the whole tree. The total number of swaps can be computed precisely and is $O(n)$, as follows.

The next to last row in a full binary tree with n nodes normally contains approximately $n/4$ nodes, but always at most $n/2$ nodes. The time to heapify the trees with roots in the next to last row is therefore less than $n/2$ times a constant. The next row up has at most $n/4$ nodes and the time is proportional to $2n/4$; the next row gives $3n/8$, and so on. If the depth of the tree is d, the total time is therefore at most a constant times

$$\frac{n}{2} + 2\frac{n}{4} + 3\frac{n}{8} + \cdots + d = \frac{n}{2}\sum_{i=1}^{d} i\left(\frac{1}{2}\right)^{i-1}.$$

Let $f(x) = \sum_{i=1}^{d} ix^{i-1}$, so the sum we are interested in is $(n/2)f(1/2)$. Then $f(x)$ is the derivative of $\sum_{i=1}^{d} x^i$; this sum is a finite geometric series minus the usual first term 1, so it is equal to $(1 - x^{d+1})/(1 - x) - 1$. The derivative of this function is easy to compute, giving

$$f(x) = \frac{(1-x)(-(d+1)x^d) + 1 - x^{d+1}}{(1-x)^2}.$$

Now we substitute $x = 1/2$ and get

$$\frac{n}{2}f(1/2) = \frac{n}{2}\left(4 - \frac{d+1}{2^{d-1}} - \frac{1}{2^{d-1}}\right) < 2n.$$

Even with this change *quicksort* is better, but there is a practical use for this improved *heapsort*. Suppose out of a large collection of n records we want to draw the smallest k in order, where k is much smaller than n. *Quicksort* is not much help since it is not designed to pick off the elements in order. (It is possible to modify *quicksort* to handle this task, but it's a little tricky.) *Heapsort* does work this way however; if we stop after k executions of the second while loop we already have the k smallest key values and the overall running time is $O(n + k \log n)$. If k is small (less than $n/\log n$) this is just $O(n)$.

A heap is really a new data structure and it is therefore worth some effort to see if there is a better implementation than one we might use for general binary trees. Because of the balanced nature of a heap, we can use an array to hold the records without adding any explicit link fields, using what is called an **implicit data structure**. To store a heap in an array, start at the root and number the nodes level by level and from left to right (that is, in level order), so the root is numbered zero. Then it is easy to see that the node numbered i has parent $(i - 1)/2$ and children $2i + 1$ and $2i + 2$. If we put the nodes into the array just as they are numbered we will be able to move easily from parent to child or vice versa without link fields, just by doing some simple arithmetic. (It is possible to pull this stunt with general binary trees, but then there may be many array slots that never get used, since general binary trees are not obliged to fill one level completely before starting a new one.)

Finally, consider what is to be done with the root record when it is removed. An obvious answer is to store it in a second array, so that when we are done we have an array containing the records in sorted order, just as for the other algorithms. The other sorting algorithms all were able to make do with one array, however, which may be important for large amounts of data. But as *heapsort* proceeds, the number of nodes in the heap decreases, which really means that the number of array slots in use decreases. Because of the nature of the heap implementation, these free slots all open up at the end of the array. Hence we may simply transfer the root record to the end of the array. When we are done, we have the records in decreasing order. To get them in increasing order, we may alter property (2) of heaps so that the key value of each node is greater than or equal to the values in its children.

7.6 Lower Bound on Sorting Time

Can we do better than $n \log n$? We will sketch the proof of a theorem that says that we cannot, except in special cases. Specifically, we assume that the sorting is done by making comparisons between two key values at a time. (We will see a way to sort that does not do this, and is $O(n)$.)

We need a way to think about sorting without limiting ourselves to one method of sorting. We can think about any sorting algorithm abstractly as a series of questions like, "Is `N.key` $<$ `M.key`?," followed by appropriate action depending on the answer. We can visualize this process as a **decision tree**: a binary tree in which each node corresponds to a question, and the answer directs us to either the left or the right child, where we find the appropriate next question. We now associate with each node a set of initial orderings of the keys, in the following way: The root contains all $n!$ possible initial orderings, corresponding to the fact that we haven't done anything yet. Now suppose we have managed to assign some set of initial orderings to a particular node and want to do the same for its children. There is a question about order that corresponds to this node; for some initial orderings the answer would be "yes" and for the rest "no" at this stage in the algorithm. We use this to decide how to divide the orderings into two sets, associating the "yes" orderings with the left child and the others with the right child. We can continue in this way dividing the sets until all the nodes are assigned a set of initial orderings.

Now imagine sorting a particular collection of input records. As the algorithm proceeds we can follow the corresponding path through the decision tree. We cannot be sure that the data is sorted until we reach a leaf node associated with just one item, for by definition if two items are associated with the same node they are initial orderings that have not been distinguished in any way by the questions asked so far. Thus at least one of the

orderings cannot be sorted at this point in the algorithm. (If both initial orderings had been sorted by the same sequence of moves, then running those moves in reverse would produce both initial orders from the sorted order, but of course running the moves in reverse can't possibly produce more than one outcome.)

Since there are $n!$ possible orderings of n records there are $n!$ leaf nodes in the tree. The height of a binary tree with $n!$ leaves is at least $\log(n!)$, so there is at least one path of length greater than or equal to $\log(n!)$ corresponding to at least one initial ordering that will take $\log(n!)$ steps to sort. It is not hard to show that $\log(n!) \geq (n \log n)/2$. Thus no matter what the method, there is at least one worst case that takes a long time to sort. It is possible to show in fact that the average length of a path through a binary tree with $n!$ leaves is also at least $\log(n!)$, which translates into an average running time no better than $O(n \log n)$.

7.7 Binsort

How might we sort without simply asking a series of questions about order? If we have no information about what the keys will be like, we can't. But suppose we know that the key values are all in some small range ($1 \ldots M$, for example) and moreover that each key value occurs exactly once in the input records. This is clearly an easy task: as soon as we examine the record we can tell where it belongs and we can just put it into the right place in the array: `A[record.key] = record`. The total running time is obviously $O(M)$, since for each of M records we do only a fixed amount of work.

In a more realistic situation we might well know that all the keys occur in some manageable range, but the records will not be evenly spread over the range, and may repeat. The method is much the same, however; we simply need a way to put more than one record into a single 'bin.' We assume that the number of records to be sorted is at least as large as the number of bins; if not, there are probably so few records that *quicksort* will work very well. Now we use the actual array slots as header nodes for linked lists, and link together all those records with the same key value. The header should contain pointers to both the first and last records on the list. We read through the records once, putting each on the list in the appropriate bin; at the end we link all the bin lists together into one big sorted list. It takes $O(n)$ time to put n records into the bins, then $O(M)$ time to link all the lists together, but since $n \geq M$, the sum is $O(n)$.

There is a neat extension of the binsort idea, called radix sort, which is really just a sequence of binsorts on different keys. For example, all ten letter identifiers is too big a set for binsorting, but a ten letter identifier can be thought of as a record with ten different keys, and each key comes from a very small range of characters. If we do a binsort on each

different key, the running time will be ten times that for a single binsort but overall is still $O(n)$.

The radix sort should do the binsort on the least significant key first, which I find counter-intuitive. As a simple example, let's sort some two digit numbers by considering each digit a different key and using a binsort with ten bins. My initial inclination would be to first sort on the tens digit (the most significant) and then sort ten smaller lists on the unit digit. It is surprisingly more uniform and efficient to sort the whole list twice, using the unit digit first.

Suppose the input keys are: 96, 4, 76, 55, 1, 24, 75, 15, 60. The first pass binsort will produce the following:

0	1	2	3	4	5	6	7	8	9
60	1			4	55	96			
				24	75	76			
					15				

Note that the records have been added at the end of the lists: for radix sort this is crucial. You should try it with additions at the beginning to see what goes wrong (you won't see the problem until after the second pass). Now we concatenate the lists into: 60, 1, 4, 24, 55, 75, 15, 96, 76, and feed it through again, paying attention to the tens digit. The second pass then gives us:

0	1	2	3	4	5	6	7	8	9
1	15	24			55	60	75		96
4							76		

Finally concatenate into: 1, 4, 15, 24, 55, 60, 75, 76, 96.

There is no reason for the collection of bins at each stage to be the same. We may want to sort personnel records by last name, first name, birth date. This could be done by using binsort on birth date, first name and last name in that order; of course sorting birth dates and names might involve a radix sort as well.

7.8 External Sorting

In all the sorting algorithms we have seen, we assumed that the records were kept in an array. In practice, we may have too many records to fit in the main memory of the computer, so that most of the records will be in files in secondary memory (usually disk memory). It is possible to use the algorithms we have seen, but we must now continually swap portions of the data in and out of main memory to examine and manipulate it. On

modern machines the bookkeeping involved in this process may be done by the operating system using a system known as "paging;" on more primitive machines the program itself would have to keep track of the information. In either case the principal drawback is speed. There are many users competing for the use of the disk and access to the disk involves mechanical motion, which of course is much slower than the electronic processes involved in actual computation. It is therefore important to limit the number of reads and writes involving the secondary memory.

Although a program will typically do reads and writes one record at a time, the actual disk access is usually done a block at a time. (A block is the minimum amount of disk storage that can be allocated to a file. This varies from system to system, but 1024 bytes is typical.) When a single record is requested, the operating system will move an entire block of the file into main memory, and subsequent reads will not involve the disk until the block is exhausted. Similarly, the operating system will buffer writes until a whole block has accumulated, then it will transfer the block to the disk all at once. To minimize the number of such block accesses, we want to devise a sorting algorithm that does not jump around in the data and that does not make too many passes over the data. The simplest sorting algorithms, like *bubblesort*, do not jump around much but do make many passes over the data. *quicksort* jumps around more but looks at each portion of the data less often; nevertheless it does too many disk access operations. Finally, because of the tree structure embedded in the linear order of the data, *heapsort* does far too much jumping around to be practical.

The external sorting algorithm we examine resembles *quicksort* in that it is also a divide and conquer algorithm, that is, it proceeds by breaking the problem into smaller subproblems that may be solved much more quickly. Naturally, it continues by breaking down these subproblems over and over until it finds a very easy instance of the problem.

The good average performance of *quicksort* is due to the fact that when we break the array of data into two, the pieces are usually about half the size of the original array. In *mergesort* we guarantee that the pieces are half the size of the original; this makes the "pasting together" of the pieces more involved, of course, but it turns out that altogether the running time is still $O(n \log n)$ in the worst case. *quicksort* remains the better algorithm for internal sorting, but it turns out that *mergesort* is better suited for adaptation to the external sorting situation.

To begin, we look at an internal version of *mergesort*; once the idea is clear we can see how to adapt it effectively to work on files instead of arrays. To *mergesort* an array of records, we begin by dividing the array in half and sorting each half (by a recursive call). We now need to merge these two sorted lists into one. We start by placing a pointer at the left hand end of each half, say `left` pointing to the left half and `right` to the right.

We compare the two key values at positions `left` and `right` and copy the smaller of the two to a new array, and then we increment `left` or `right` as appropriate. We continue in this way until we exhaust one half or the other, and then simply copy the remainder of the other half to the new array. The actual algorithm is written to work on a subarray so that it can be called recursively; see figure 7.5.

```
void mergesort(int i, int j) {
  int B[j-i+1];
  int index,right,left;

  if (i<j) {
  mergesort(i,(i+j)/2);
  mergesort( (i+j)/2+1,j);
  index = 0;
  left = i;
  right = (i+j)/2+1;

  while ( left <= (i+j)/2 && right <= j) {
    if (A[left] < A[right]) {
      B[index] = A[left++];
    } else {
      B[index] = A[right++];
    }
    index++;
  }

  while ( left <= (i+j)/2 ) {
      B[index++] = A[left++];
  }
  while ( right <= j ) {
    B[index++] = A[right++];
  }
  for (index = i; index <= j; index++ )
    A[index] = B[index-i];
  }
}
```

Figure 7.5 Mergesort

Although this presentation of *mergesort* as a recursive procedure is very natural, it is easy to program essentially the same algorithm as an iterative procedure that is more readily adapted to use on files. We begin by making one pass through the array arranging adjacent pairs in the correct order. That is, we make sure that the records at positions 1 and 2 are in the correct order, and likewise for positions 3 and 4, 5 and 6, etc. We then start over at the beginning and merge the "runs" of size two into runs of size four, then

eight, etc. Whenever we merge two runs of length k, the time taken is $O(2k)$, because as we merge we transfer one of the $2k$ items to the array `B` each time we compare two key values; after $2k$ comparisons we have accomplished the merge. Thus, a pass through the whole array takes $O(n)$ time, and we make $(\log n)$ passes through the array, since after doubling the run size $(\log n)$ times we get a run size bigger than n. The total (worst case) time for this algorithm is therefore $O(n \log n)$.

Since this non-recursive algorithm simply makes repeated passes through the array, it is quite easy to use a file instead of an array. On each pass of the algorithm, we would open an input file containing the data organized into runs of some size k, and an output file initially empty. We then read two runs into memory, merge them and write the run of size $2k$ into the output file. The only problem here is that the runs will eventually get too big to fit in main memory, yet we cannot start the merge until we know the first key value in the second run. The solution is to maintain two input and two output files. We open the two input files and start reading records from each, merging the first runs from each file, then the second and so on. In this way we never have to read an entire run into memory. As we write the runs to the output files, we alternately put runs into the first and second output files, so when the pass is complete, we can reset the output files, use them as input files, and start the next pass. The records will initially be in a single file, so we need to do something special to get this process going. The best way to start is to read as many records as possible into memory and use some fast sorting algorithm, like *quicksort*, to produce a large sorted run and write it to an output file. We continue until the single input file is exhausted, writing long runs alternately into two output files. Now we start the scheme described above, but we have a nice "head start," that is, we already have large runs to merge and they were produced more efficiently than if we had started with run size one or two. For example, if we are sorting 10,000,000 records and main memory will hold 10,000 records at a time, then we start by producing runs of size 10,000. We then need to make 10 more passes to get a run size of greater than 10,000,000. If we were to use *mergesort* from the beginning, starting with run size 1, we would need to make 24 passes before the records were sorted.

Index of Numbered Items

Index