Chapter 6

I/O of Data Structures

6.1 An Introduction

At this point in the semester, we've seen many of the fundamental kinds of data structures (and will cover more, later in the semester). We've generally explored these data structures from the perspective of how they can be represented and manipulated in *memory* using C constructs. However, it is often desirable or necessary to store data structures in some other form, such as a file, where the methods of representation available are quite different. The general nature of the techniques is a special case of the general process of *serialization* (the process of converting a data structure into a sequential or "serial" sequence of bytes so that it can be sent over a network, printed, or stored on a "serial" device such as a CD-ROM or magnetic tape). This chapter does not focus on how serialized data structures can be efficiently accessed or manipulated in more general ways– for example, how to perform a FIND operation on a binary tree stored as a disk file. The only operations we will focus on are how to store and restore data structures.

Rather than try to give solutions for every possible situation and discuss every tradeoff between different approaches, this chapter attempts to illustrate several key themes in data structure serialization, and show how they can be used, either alone or together.

6.1.1 A Word of Advice

Currently, computer processing speeds are enormously faster than I/O speeds. A typical PC can perform one million arithmetic operations in less time than it can read a single byte of information from its disk. Much the same situation exists in networking (although the range of network speeds varies widely)– a typical PC can perform a considerable amount of computation in the time it requires to send an inquiry to a remote server and receive its response. This means that for many kinds of problems, if the answer can be computed by purely computational means, it is more efficient to recompute the answer every time it is needed than it would be store the answer in a file and consult the file (or to ask a remote server to compute the answer to the question and return the answer).

Therefore, before asking *how* a data structure can be represented in a file, it usually makes sense to ask *whether* it makes any sense to store the data structure at all! For the purpose of this chapter, we will assume that the answer to this question is always yes, but in practice this is an important question to consider.

There are many situations where the data or its structure is truly important, usually because it cannot be recomputed by any known means. There many other cases where the data is relatively easy to recompute, however, and the line between easy and hard to compute is changing constantly. Early computers used tables to compute trigonometric functions, but today the idea of storing such tables on disk rather than recomputing them as necessary seems unthinkable.

6.2 Arrays

The first case study that we'll investigate is how to write and read arrays. We will begin with fixed-size one-dimensional arrays of integers, and then generalize our methods until we are able to deal with arrays of unknown dimension, extent, and type.

6.2.1 Arrays of Fixed Size

For the one-dimensional case, the solution is very easy– in C, one-dimensional arrays are stored as a contiguous block of memory. The standard I/O library makes it easy to read and write such blocks (i.e. using `fread` and `fwrite`). The only complication can occur when the arrays are of unknown extent or type, but these situations are discussed in later sections.

The two-dimensional case is more interesting, however, and is worth considering for a moment. If we were always reading and writing two-dimensional arrays of the same size (for example, NUM_ROWS by NUM_COLS, where NUM_ROWS and NUM_COLS are fixed numbers that everyone has agreed to), then reading and writing arrays would be a simple matter, as illustrated in figure 6.1.

Figure 6.1: Code to read and write 2-dimensional arrays of fixed size.

```
int array2dRead (int array[NUM_ROWS][NUM_COLS], FILE *in)
{

        int rc;

        rc = fread (array, sizeof (int), NUM_ROWS * NUM_COLS, in);
        return (rc == (NUM_ROWS * NUM_COLS));
}

int array2dWrite (int array[NUM_ROWS][NUM_COLS], FILE *out)
{
        int rc;

        rc = fwrite (array, sizeof (int), NUM_ROWS * NUM_COLS, out);
        return (rc == (NUM_ROWS * NUM_COLS));
}
```

6.2.2 Arrays of Unknown Extent

For many purposes, arrays of a fixed size are sufficient (at least until someone changes NUM_ROWS or NUM_COLS). However, in many cases, it is impossible (or impractical) to know ahead of time what the extents of the array actually are. Therefore, something we need to store (in addition to just the data in the array) is the size of the array itself.

One way to do this is to store the number of rows and columns of the array as the first two integers in the file, followed by the data, row by row.

A sketch of an implementation using the `array2d_t` structure from the *Pointers and Malloc* handout is shown in figure 6.2. This code omits all of the error checking that is generally necessary when using I/O, but gives the basic algorithm.

The first two integers, which represent the number of rows and columns, form a header for this serialized array; they tell the reader the information that they need to know in order to read the array. More information can be added to the header, if necessary; for example, an integer representing the type of each data element (`char`, `int`, or any other C type) could be used if the type of the array was unknown ahead of time.

Similarly, information about the format of the data elements themselves (size and byte-order) can be stored in the header, so that programs on machines with differently sized integers or different binary representations for integers can read the data correctly.

6.2.3 Arrays of Unknown Dimension

The algorithms from the previous sections are jury-rigged to work with two-dimensional arrays. They can be extended to work with arrays of higher dimension, using similar techniques, but there is another problem yet to be solved– how can we deal with an array that has an unknown number of dimensions?

One problem that would immediately arise is how to represent such an object in C in the first place– all the arrays we have seen so far have had a static number of dimensions, known at compile-time, and this is generally the way C programs are written. It is possible to write code that does otherwise, but I will not pursue this topic in this chapter. Instead of focusing on the C tricks necessary to use such arrays in C, I will focus on the methods used to read and write these arrays.

If the number of dimensions is known to be small, then a fixed-size header can be used, similar to the header used in the previous section to handle arrays of unknown extent. This header can list the extent of each dimension, with a sentinel value of zero to indicate the highest dimension, if it was less than the limit.

This technique works well if the number of dimensions can be bounded. However, this

Figure 6.2: Code to read and write 2-dimensional arrays of unknown size. Note the complete lack of error checking.

```
array2d_t *array2dReadDynamic (FILE *in)
{
        int rows, cols, i;
        array2d_t *array;

        fread (&rows, sizeof (int), 1, in);
        fread (&cols, sizeof (int), 1, in);

        array = arrayCreate (rows, cols);

        for (i = 0; i < rows; i++)
                fread (array->elems [i], sizeof (int), cols, in);

        return (array);
}

void array2dWriteDynamic (FILE *out, array2d_t *array)
{
        int i;

        fwrite (&array->rows, sizeof (int), 1, out);
        fwrite (&array->cols, sizeof (int), 1, out);

        for (i = 0; i < array->rows; i++)
                fwrite (array->elems [i], sizeof (int), array->cols, out);

        return ;
}
```

is not always the case. In this case, the approach is to use a dynamic header– the list of dimensions and their extents is *itself* a dynamic 1-dimensional array, represented in the manner shown in the previous section. In this representation, the input file contains:

1. Any information about the byte-order, word-size, etc of the represented data.
2. An integer specifying the number of dimensions.
3. An array of integers representing the extent in each dimension.
4. The data itself.

6.2.4 Arrays of Unknown Type

We have made a pair of simplifying assumptions in our discussion up to this point: first, we have assumed that both the reader and the writer implicitly know what the type of the array is (i.e. `int`, `double`, etc), and second that both the reader and writer use the same representation for these types.

This is not always the case. For example, different machines use different byte orders or different numbers of bytes to represent multi-byte objects, and different machines may use different binary representations to represent bytes (including bytes of different sizes).[1]

In order to successfully read data stored in the native binary representation of the writer, the reader must know what that format is (and how a translation into its own native format can be performed). Alternatively, all writers (no matter what their native representation) can write in a canonical format that all readers (no matter what their native representation) can read.

If writers always write in their own native format (and the readers perform whatever translation is necessary), then a marker must be added to the data that indicates exactly what representation was used to write it, so that the reader can choose the appropriate reading routines. This marker must be chosen in such a way (and written in a canonical, universally agreed-upon manner) so that all readers can unambiguously determine the way that the file was written.

[1] This is less and less of a worry each year; currently nearly every CPU uses 8-bit bytes, two's complement integers, and IEEE floating point.

6.2.5 Data Definition Languages

We began this section with an example where the reader and the writer implicitly agreed on the dimension, size, and type of the arrays, as well as the underlying representation of that type. By the final example in the previous section, all of the implicit assumptions had been removed, except one– the writer and reader have implicitly agreed that what is being written and read are arrays.

In some applications, even this assumption is invalid– the reader may have *no* idea what kind of thing the writer has left in the file, and may have to learn the type and format of the data from the data itself. This problem can be solved by adding still more information to the header, so that the data includes a description of its format, explaining how it should be interpreted by any readers. This description, however, must be in a form that is implicitly shared by the readers and writers– if the writer doesn't describe the data in a way that the reader can understand, then the data is useless to the reader.

6.3 Strings and String Fields

For this case study, we'll investigate how we can read and write records containing information represented by strings. In a general sense, strings are simply one-dimensional arrays, and can be handled as such, using the methods that we have already seen. Typically, however, there is useful information (such as information about the representation of the string) encoded in the string itself, and therefore we will consider strings as a separate case.

6.3.1 Delimited Strings

Imagine that we have a set of records that we wish to store in a file. Each record consists of a fixed number of fields, and each field consists of a string. Since some (or all) of the fields are of varying length, however, each record itself is represented by a string. How can we tell where each record and each field in the record begins and ends?

There are a number of approaches to solving this problem. The most common one, at least in the UNIX world, is to have each line in the file represent a single record, and have the boundaries between each field in the record denoted by a special string (usually just a single character like "`:`") called a *field delimiter* (or just a *delimiter*). For example, here's what a small file that represents a phone book might look like:

```
Ellard:Dan:496-6246
Ellard:Penny:643-9644
```

Writing records to a file in this format is simple; for each record, just do something along the lines of:

```
fprintf (fout, "%s:%s:%s\n", lastname, firstname, telephone);
```

Figure 6.3 is an example of an implementation of some code to read in a file of this form. You should note the following things about this code:

- It allows "comments" to be intermixed with the data. A comment line is any line that has a '#' as its first character.
- The logic for figuring out where each field begins and ends could be done with a single call to `sscanf`, and in fact the entire process of reading in each line and chopping it up into fields could be accomplished with a single call to `fscanf`. It might be interesting for you to figure out what the format string you would need to pass in to `fscanf` would be in order to accomplish this. Alternatively, finding the start and end of each field individually can be accomplished via repeated calls to `scanf` (or `fscanf`).

Given that I *could* have used `fscanf` (or some member of the `scanf` family, or `strtok`), it may seem odd that I did not. (In general, doing things the easy way is always my choice.) The reason is that these functions, while handy for many simple purposes, are hard to extend. In the next section, we'll extend these ideas significantly.

6.3.2 Escaped Strings

In the code in figure 6.3, there is no way to represent a field that contains the delimiting character itself. For example, if you ever try to enter a last name that has a ':' in it, our phone book routines will fail. In some applications, this is not a problem; if there exists a delimiter character that can be chosen that will not, due to some property of the data, ever appear in the data, then this approach is sufficient.

In the more general case, however, it might not be possible to choose a delimiter character at all, or at the very least it may not be possible to choose one that will work for all applications. Therefore, in this section we will discuss how delimited strings can be implemented in a manner which allows all of the characters in the character set to appear in any field.

The usual way to do this is to *escape* the field delimiting character by adding a special character or string (the *escape character*) before each character that needs to be "protected". You have already seen this technique in practice; `printf` uses the % character to escape the meaning of other characters.

Figure 6.3: Code to split a string into fields.

```
char **stringSplit (char *line, int field_count, char delim)
{
        char *eol, *start, *end;
        char **fields;
        int i;

        eol = strchr (line, '\n');
        if (eol != NULL)
                *eol = '\0';

        fields = (char **) malloc (field_count * sizeof (char *));

        for (i = 0; i < field_count; i++)
                fields [i] = NULL;

        start = line;
        for (i = 0; i < field_count; i++) {
                end = strchr (start, delim);
                if ((i == field_count - 1) && (end == NULL))
                        end = strchr (start, '\0');
                assert (end != NULL);

                fields [i] = malloc (sizeof (char) * (1 + end - start));

                strncpy (fields [i], start, end - start);
                fields [i][end - start] = '\0';

                if (*end != '\0')
                        start = end + 1;
                else
                        start = end;
        }
        assert (*start == '\0');
        return (fields);
}
```

Unfortunately, often it is unacceptable to use exactly the same scheme as `printf` (which uses % as the escape character, and escapes % itself with %%), since we may well want to allow two adjacent instances of the delimiter character to signify an empty field. For example, we might want "::" to signify an empty field, rather than a single ":" (but of course, we still want a way to represent a single ":"). What can be done is to use *another* character or string as the escape character, instead of using the delimiter as the escape.

The character that is most typically used in UNIX is backslash (\). So, if we ever want to represent the name of some eccentric named "`foo:bar`" in our system, his name could be stored as "`foo\:bar`".

Of course, this immediately leads to another problem (that might not be immediately obvious). What if we then have to deal with another eccentric whose last name ends in "\"? The answer, again, is that we need to escape the escape character itself! Luckily, we *can* use the escape character to escape itself (unlike the delimiter character, which cannot escape itself), so there the saga ends.

In our example code, we actually have *two* delimiting characters: the ":", and the newline. If we want to allow the user to embed newlines in their strings, then we can still accomplish this with the same method of escaping the newline- but we'd need to modify the code somewhat, because it uses `fgets` everywhere, and `fgets` treats newlines specially. However, despite the fact that it would require a little more coding to get working, algorithmically we now know exactly what we would need to do.

6.3.3 Annotated Strings

In many cases, it may be necessary to escape sequences of multiple characters. These substrings are *annotations* of the text. (In different contexts, different terms are used to describe this concept, but I'm going to use this single term to try to be consistent).

For example, in HTML the formatting commands are embedded directly in the text of the web pages, as annotations. In HTML, annotations generally begin with a "$<$" and end with a "$>$". In C, you can think of comments as annotations that begin with `/*` and end with `*/` (although of course this fails to consider what happens when these sequences occur inside C strings). In these cases, the start and ending delimiters of annotations are different, but in some applications the same character is used to delimit both the start and end of an annotation (as double quotes are used to represent both the start and end of strings, for example).

Annotated strings are slightly more complicated to deal with than escaped strings, because there is (in the general case) no way to know ahead of time how long each annotation is. However, a state machine can be used to process the string and keep track of what sections represent normal text and what sections represent any annotations.

Note that annotated strings are often escaped as well: otherwise, there would be no way to represent the annotation string itself in the text. For example, in HTML, the escape character is "&".

Note that although HTML is a convenient (and ubiquitous) example of annotated text, it is also not a very good example, and certainly not a clean design. Instead of simply escaping the delimiters, each has its own special escape (for example, the string "<" is the escaped form of "<"). In addition, despite the fact that the escape character can be used to escape itself, the HTML escape character is escaped differently, as "&".

6.4 Binary Trees

The next problem we will tackle is how to store binary trees in a file. As always, there are a vast number of different situations that might arise, but luckily in the case of binary trees, these can divided into two general cases- binary search trees, and unordered binary trees.

6.4.1 Binary Search Trees

Imagine that we have an inorder binary tree consisting of integers- each node contains an integer value; the left subtree (if any) contains only keys less than the key of the node, and the right subtree (if any) contains keys that are greater. Imagine that we want to store this binary tree in a file in such a way that we can reconstruct it *exactly* as it was. In particular, our goal is to store the keys in such an order that when if we start with an empty tree, and we read them in again in the order they appear in the file and insert them into the tree, the resulting tree has exactly the same form as the original.

This problem may seem somewhat contrived, given the binary search trees that we have seen– the key properties of binary search trees (at least as far as we have discussed them so far) is that the keys are inorder, and (in order to minimize search time) the tree is reasonably balanced. Both of these properties can be attained by constructing the tree in a proper manner, regardless of the order that the keys are inserted into the tree. However, there are other kinds of binary search trees where there are other relationships between the nodes, and the exact shape of the tree actually has a great deal importance.[2]

[2] For example, multi-key search trees, where each level of the tree uses a different key.

Choosing a Traversal

Your first thought might be to write a routine that walks through the tree using one of the traversals that we've seen in class (inorder or depth-first, preorder, postorder, or level-order), writing out the value of each node as you encounter it- and you'd be right, but only if you picked the correct traversal!

If we use an inorder traversal, we'll put the values onto the disk in a sorted order- and as we've seen, if we build a binary tree from sorted keys (assuming that no balancing operations are performed), then we will always get a tree that looks like a linked list. Unless by some incredible coincidence all the trees we want to store happen to look like this, the resulting trees will not be identical to the originals.

If we use a postorder traversal, it won't work either (although at least the trees won't always come out looking like linked lists). If we try a few different examples, we can see see that it just doesn't do what we need, but after some fooling around you might start to catch on to *why* neither inorder or postorder does the right thing- which leads us to a preorder or level-order traversal.

Recall that when we build a binary search tree using the standard insertion routines, we build from the top down– nodes are always added as leaves. Therefore, in order to reconstruct the original tree, the following property must hold- *before a key is inserted into the tree, the key which was the key of its parent node in the original tree must already be in the tree.* Preorder or level-order traversal will do what we need, since each always visits parents before their children, and so parents will always appear before their children in the file.

Since preorder is trivial to code and efficient (while level-order slightly more complex and may require more space), it makes sense to use preorder. It's interesting to note that there are a number of other traversals that share this property.[3] However, since preorder is the easiest to code and understand, there is no reason not to use it.

For an example of an implementation, see figure 6.4.

This code is very simple– partly because it doesn't do much error checking or attempt to recover from any errors, but mostly because the algorithm is very short and simple. Note that the `treeInsert` function can be any ordinary tree insertion routine, but it must not be one that attempts to balance or otherwise modify the existing shape of the tree in any way other than creating a new leaf!

The code in figure 6.4 only reads and writes integer keys. It could be extended to work with binary search trees that use some other kind of key, and have values composed of

[3] In fact, preorder isn't the first one that I thought of when I first had to solve this problem myself, although I wish it had been, because it would have saved me a lot of work!

Figure 6.4: Routines to read and write binary search trees with integer keys.

```
void treeWrite (tree_t *tree, FILE *fout)
{
        int rc;

        if (tree == NULL)
                return ;

        /* Write the key (and value, if any). */
        rc = fwrite (&tree->key, sizeof (int), 1, fout);
        assert (rc == 1);

        treeWrite (tree->left, fout);
        treeWrite (tree->right, fout);
        return ;
}

tree_t *treeRead (FILE *fin)
{
        tree_t *tree = NULL;
        int key, rc;

        for (;;) {
                /* Read the key (and value, if any). */
                rc = fread (&key, sizeof (int), 1, fin);
                if (rc != 1)
                        break;

                tree = treeInsert (tree, key);
        }
        return (tree);
}
```

arbitrary records of strings or other types, using the same sort of methods that we used in the previous section that described how to deal with strings.

6.4.2 Unordered Binary Trees

Now that we've conquered binary search trees, let's tackle something a little more complicated- "unordered" binary trees, such as expression trees or animals trees (like those that we used in the `animals` program).

The crucial difference between this type of tree and a binary search tree is that there isn't any general way to compare any two arbitrary keys in order to determine which one should come before the other. We know at a glance whether one integer is larger or smaller than another in a binary search tree with integer keys, but in an animals tree there's no way to tell at a glance whether "cat" will appear to the left or right of "dog" or "bear"- we must traverse the tree, answering questions about these beasts in order to know where they will be located.

So, how can we proceed? There are several possibilities.

An Explicit Reconstruction

One approach is to encode the explicit sequence of operations that was used to construct the tree itself.

For an animals tree, this reduces to storing an explicit sequence of "yes", "no", questions, and animal names that can be used to reconstruct the original tree. This does *not* mean that the original game itself needs to be saved: only the information represented by the tree. In effect, we would be recording an abbreviated game of animals, in such a way that reading it back in would simulate replaying the game. This requires storing the relationship between each node and all of its ancestors (i.e. the sequence of "yes" and "no"s that lead to each node).

Like in the previous case, we would need to do a preorder walk of the tree when writing it, in order to ensure that the ancestors of each node are installed before we try to install the node. However, depending on how the tree is actually constructed, it is not always the case that the ancestor questions of an animal node preceeded the animal in the game– in fact, the opposite is the case. Therefore, some care is needed.

For animals, this seems like a complex and inefficient task, but for other problems, this is not necessarily a bad approach. If constructing the tree is not terribly inefficient, or the contents of the tree can be represented easily in some other manner, then this approach works well. For example, in an expression tree, it is very efficient to convert an infix expression into an expression tree (or vice versa)– and the infix expression is a more compact representation of the expression (which makes it more efficient to store).

An Implicit Representation

A better approach (in most cases) is to record the structure of the tree implicitly instead of explicitly recording all of the information used to construct the tree. This requires less storage and is somewhat more flexible.

For example, to write out an animals tree, we do a preorder walk, but along with the contents of each node we record whether the node is a question node or an animal node. When we read the tree back in, we read it again in preorder, and know where we are in the recursion based on whether the node is a question node (in which case we need to recursively read in the "yes" and "no" children) or an animal (the base case of the recursion).

An example of this algorithm is given in C in figure 6.5. The reading algorithm, though also quite simple, is quite a bit less obvious than the algorithm for reading a binary search tree, and you may need to try it once or twice to convince yourself that it actually works.

Some particular details to note about the code in figure 6.5:

- This code assumes that the tree is exactly like the trees we used in `animals`- each node is either an internal node with two children, or a leaf with zero children. This code could be extended to handle trees where nodes can have a single left or right child.

  This can be done by modifying the `aniTreeWrite` function to denote "question" nodes with a single left child with `QUESTION_LEFT_CHAR`, and nodes with a single right child with `QUESTION_RIGHT_CHAR`, and then modify the case statement in `aniTreeRead` as shown in figure 6.6.

- When the original call to `aniTreeRead` returns, it is assumed that the entire tree has been read in. Real programs might do some error checking to make sure that there isn't any more data in the file, and indicate some sort of error. (In general, this code is almost entirely lacking in error checking.)

Animal trees have a relatively simple structure, so these routines are quite simple. In the more general case, binary trees can be represented by strings using the method sketched in algorithm 6.1:

An example of a tree represented this way is shown in figure 6.7. Note that this method can easily be extended to work for M-ary trees as well.

Figure 6.5: Routines to read and write an Animals tree.

```
#define ANIMAL_NODE_CHAR        'A'
#define QUESTION_NODE_CHAR      'Q'

void aniTreeWrite (tree_t *tree, FILE *fout)
{
        if (tree == NULL)
                return ;
        else if ((tree->yes == NULL) && (tree->no == NULL)) {
                fprintf (fout, "%c%s\n", ANIMAL_NODE_CHAR, tree->animal);
        }
        else {
                fprintf (fout, "%c%s\n", QUESTION_NODE_CHAR, tree->question);
                aniTreeWrite (tree->yes, fout);
                aniTreeWrite (tree->no, fout);
        }
}

tree_t *aniTreeRead (FILE *fin)
{
        tree_t *node = aniNodeCreate (NULL, NULL, NULL, NULL);
        char type = getChar (fin);      /* get the type char.   */
        char *text = getLine (fin);     /* read a line...       */

        switch (type) {
                case ANIMAL_NODE_CHAR   :
                        node->animal = text;
                        break;
                case QUESTION_NODE_CHAR :
                        node->question = text;
                        node->yes = aniTreeRead (fin);
                        node->no = aniTreeRead (fin);
                        break;
        }
        return (node);
}
```

Figure 6.6: Dealing with question nodes that have only a left or right child.

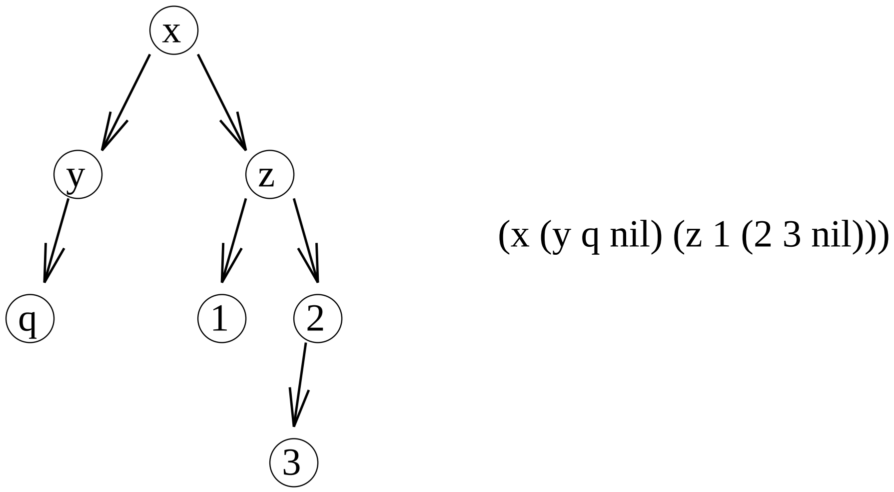
```
case QUESTION_LEFT_CHAR : {
        node->question = text;
        node->yes = aniTreeRead (fin);
        node->no  = NULL;
        break;
}
case QUESTION_RIGHT_CHAR: {
        node->question = text;
        node->yes = NULL;
        node->no  = aniTreeRead (fin);
        break;
}
```

Algorithm 6.1 (Representing a Binary Tree as a String) *An algorithm to generate a string that represents a binary tree.*

Beginning with an empty string, perform a preorder traversal of the tree. At each node, add the representation of that node to the end of the string, according to the following rules:

- If the node is a leaf, then it represented by the string representation of its key (and value, if any). Any instances of any of the delimeters used by the tree representation must be escaped if they appear in this string.
- If the node is NULL, it is represented by a special annotation. In this example, the string `nil` will be used. (Note that any instances of the string `nil` must be escaped.)
- An internal node is represented by a left node delimeter (in this example, a left parenthesis), followed by the string represenation of its key (and value, if any), a field delimeter (in this example, a space), the string representation of its left descendants, a field delimeter, the string representation of its right descendants, and finally a right delimeter (in this example, a right parenthesis). Note that any instances of the left or right node delimeters or the field delimeter in the key or value must be escaped.

Figure 6.7: String representation of a binary tree.



Adding Inorder Keys

The simplest (and perhaps most flexible) approach is to add an extra integer "position" field to each node in the tree. This field is used *only* by the I/O routines. This field is filled in by an inorder traversal of the tree immediately before it is written, and then used as the key to reconstruct the tree.

The algorithms for reading and writing are given in algorithms 6.2 and 6.3.

Algorithm 6.2 (Unordered Binary Tree Write) *Algorithm to write an unordered binary tree to file.*

1. Do an inorder traversal of the entire tree, filling in the position field with the inorder position of each node in the tree.

2. Write the tree as an inorder tree, using this field as the key.

This approach is simple and fast, but requires some storage overhead. The fact that it requires storing information in the tree itself means that this approach can only be used when the tree can be modified (and fields for this purpose were set aside when the data

Algorithm 6.3 (Unordered Binary Tree Read) *Algorithm to read an unordered binary tree (written by algorithm 6.2) from file.*

Read the tree as a normal inorder tree (using the algorithm described earlier), using the position field as the key for insertion.

---

structure was created). If this is not the case, then this algorithm cannot be used as stated here, although its central idea can be used. (See the exercises.)

6.5 Exercises

1. Prove the statement (from section 6.3.2) that it is always possible that the escape character can be used to escape itself (assuming that the delimiting character is different from the escape character).

2. Our method of reading a binary search tree is $O(n \log n)$, since it inserts n items into the tree, with an expected time of $O(\log n)$ per insertion. (In the worst case, of course, each insertion may be $O(n)$, so that the reconstruction requires time $O(n^2)$). This is not usually a matter of concern, since the $O(n)$ disk operations necessary will be so much slower than the tree insertion. For very large and unbalanced trees, however, this could present a problem.

   Show how it is possible, by storing an additional $O(n)$ items of data with the tree (and using $O(n)$ additional memory while the tree is being reconstructed) to *always* reconstruct the tree in time $O(n)$. (If you find a way achieve time $O(n)$ with less then $O(n)$ additional storage, feel free to explain how this can be accomplished– but be careful!)

   Hint– consider the techniques for storing unordered binary trees.

3. Show how it is possible to use the idea behind algorithms 6.2 and 6.3 so that unordered trees can be written and read using the basic ideas behind these algorithms but *without* modifying the original tree or marking any of its nodes in any way. $O(n)$ extra storage space can be used (both in the file and in memory).