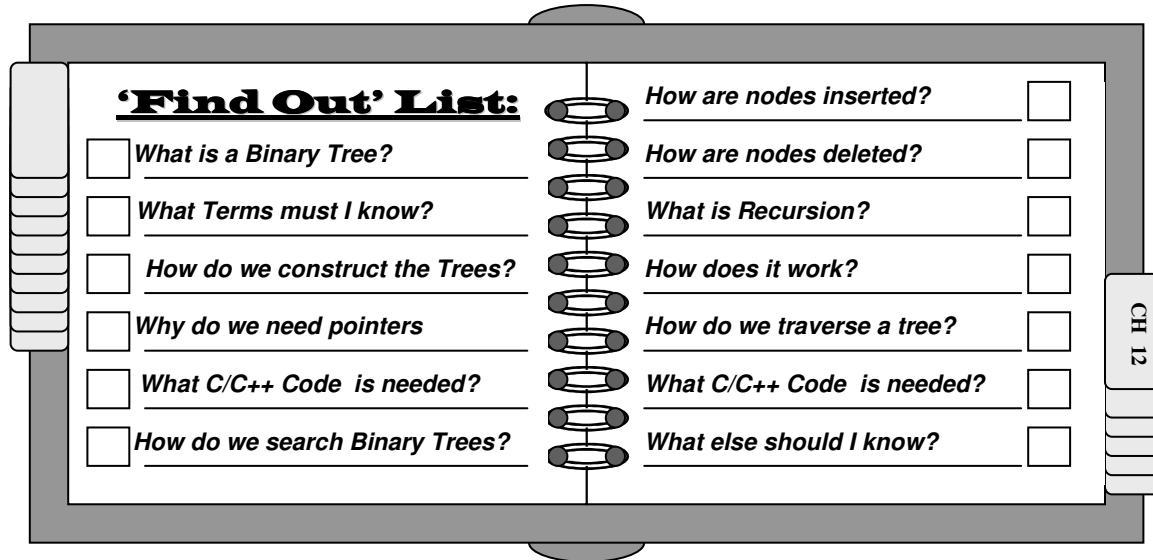


CHAPTER 12: BINARY TREES (and Recursion)

*“A fool sees not the same tree
that a wise man sees”*
William Blake (),
The Marriage of Heaven and Hell, `Proverbs of Hell'

“A tree is recognized by its fruit”
The Bible, Mathew 12:33



Introduction

In the previous chapter, we introduced the concept of allocating memory as we need it, and releasing it when we don't. The approach allowed us to efficiently set up lists which could be easily linked and maintained. However, we still faced one major problem: locating a record quickly. While we can reduce the time needed to find a record in the list through the use of doubly linked lists or leveled lists, we know that as the lists becomes large, searching can still be a prolonged process.

We Already Know

A binary search is the fastest method of finding a record, but requires that records be physically ordered and stored contiguously.

The major problem lies in the manner in which we laid out our records. Even if we physically move the records such that they are ordered in some fashion (i.e., the 1st record is located before the 2nd, the 2nd before the 3rd, and so forth) because we can *not* calculate the base address of any record on the list (as we can with an array), we must still perform a sequential in order to locate a record.

As we will see, a binary tree is intended to approximate the

speed of a binary search while allowing us to dynamically allocate records and store them non-contiguously.

You will notice also that this chapter include the term *and Recursion* (in parentheses) as part of its title. Recursion is necessary to search for items in a binary tree. However, it is not an emphasis of this chapter.

Binary Trees

A binary tree, as the name implies, is a *hierarchical* structure which can have (at most) two branches. It is a *linked list* that incorporates a *binary search* strategy. Def

What is a Binary Tree ?

Assume that we have the following list of integers: **9, 12, 4, 13, 10, 6, 3, 7, 14**

Suppose that we wished to set these elements into a dynamically allocated linked list. First, we would have to set up our structured data object to include a pointer to the next element on the list. Assume the following structure and pointer variables:

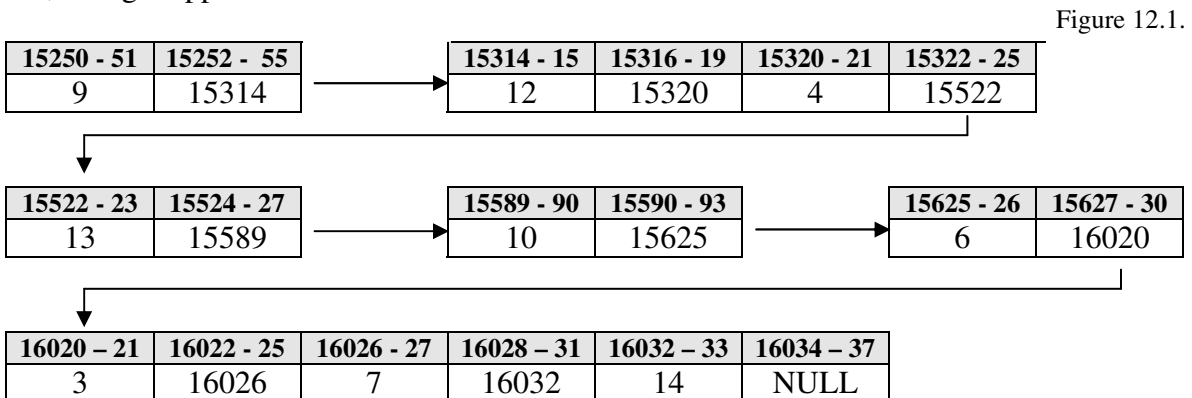
C/C++Code 12.1.

```

struct numbrec
{   int value;
    struct numbrec * next; };
int main()
{   struct numbrec * number, * lastnumber;
    
```

Where pointer variable *number* will point the record we are storing, and *lastnumber* will point to the previous number we stored.

Let's assume that the first available RAM address (when we dynamically request using malloc) is 15250. If we were to look at RAM after we had stored all of the elements on our list, it might appear as:



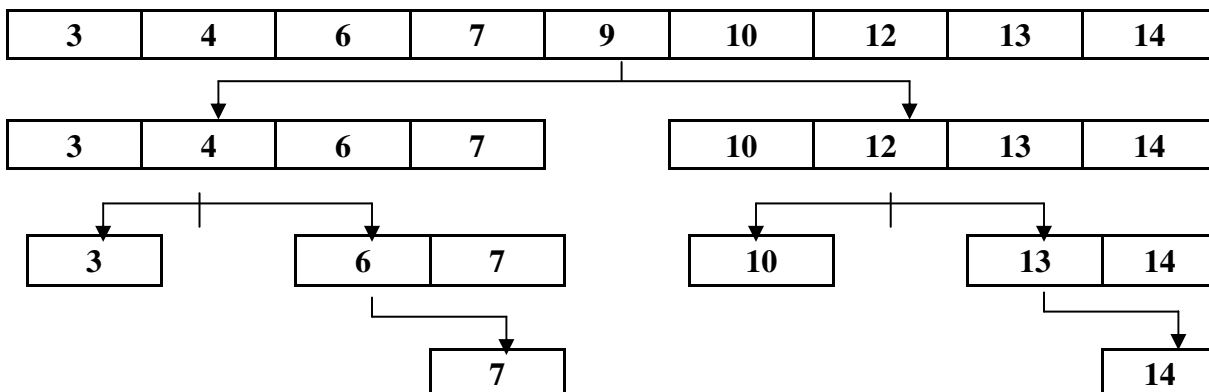
Where, dependent upon the availability of RAM, the elements may, or may not, be stored contiguously.

 ***This is a simple linked list. What's the difference??***

None, right now. In fact a binary tree *is* basically just a linked list. The difference is in how we set up the linkages between the elements on the list.

Recall how we previously performed a binary search. First, the list had to be physically sorted. The main idea was to search the list by splitting it in half by eliminating those elements which we knew could not be the one we were looking for: elements which were too large (or too small) were ignored. We would continue until we either found the element on our list, we there were no more items on the list, meaning that the element we were seeking was not on the list. The procedure would basically appear as below:

Figure 12.2.

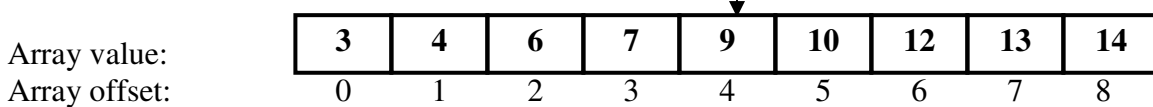


Where the arrows indicate the midpoint of each of the (sub)lists.

When we examine Figure 12.2., we notice that there is a pattern to each of the sublists: sublists containing smaller elements are to the left of the midpoint; sublists containing larger elements are to the right of the midpoint. The figure also corresponds to manner in which we would perform a binary search. For example, suppose we were looking for the numeric value **14** on our list (along with 7, the number requiring the most number of comparisons):

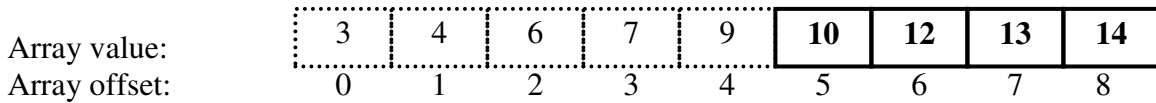
Figure 12.3.

Search 1: The middle of the list is $\lfloor \frac{\text{first} + \text{last}}{2} \rfloor = \lfloor \frac{0 + 8}{2} \rfloor = \lfloor 4 \rfloor = 4$



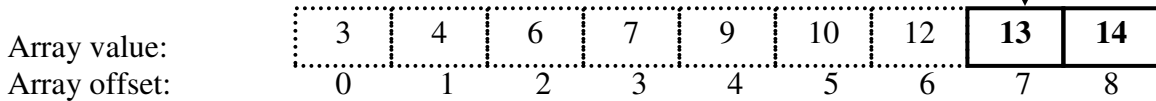
Since $13 > 9$, the value must be to the right of 9.

Search 2: The middle of the list is $\lfloor (first + last)/2 \rfloor = \lfloor (5 + 8)/2 \rfloor = \lfloor 6.5 \rfloor = 6$



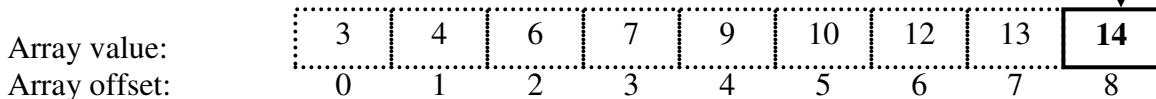
Since $14 > 12$, the value must be to the right of 12.

Search 3: The middle of the list is $\lfloor (first + last)/2 \rfloor = \lfloor (7 + 8)/2 \rfloor = \lfloor 7.5 \rfloor = 7$

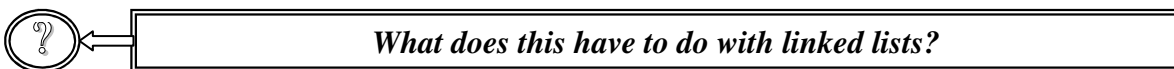


Since $14 > 13$, the value must be to the right of 13.

Search 4: The middle of the list is $\lfloor (first + last)/2 \rfloor = \lfloor (8 + 8)/2 \rfloor = \lfloor 8 \rfloor = 8$

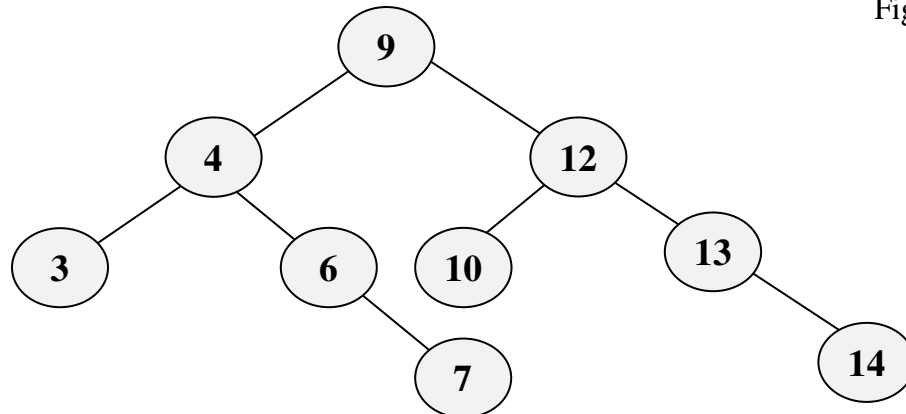


And we have found the number we are looking for



Suppose that we were to set up our linkages in such a fashion as to emulate Figure 12.2. Conceptually, we might set up the following data structure:

Figure 12.4.



We would start by searching the element that *would be* in the midpoint element *if* the lists were sorted. If the element we are looking for is less than that element, we would follow a pointer leading us to all the elements smaller than that element. As with the sorted list, the element found at that address would correspond to the midpoint of all elements smaller than the number we are looking for. If the element we are looking for is greater than that element, we would follow a pointer leading us to all the elements larger than that element.

Let’s look at how we might (once again) find the numeric value **14** on our linked list:

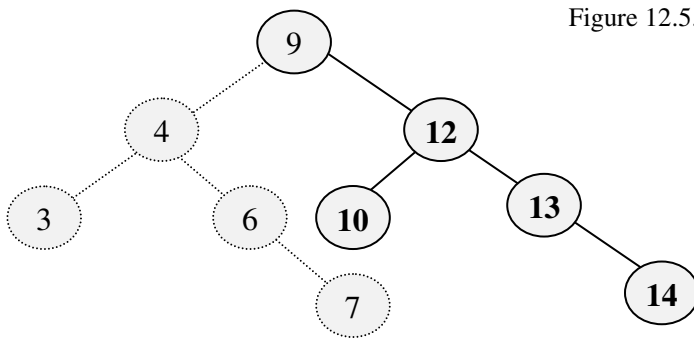


Figure 12.5. **Comparison 1:**

Since 14 is larger than 9, we know that if the value 14 is on the list, it must be to the right of 9; we can ignore the value 9 and all values to the left of 9.

Comparison 2:

Since 14 is larger than 12, we know that if the value 14 is on the list, it must be to the right of 12; we can ignore the value 12 and all values to the left of 12.

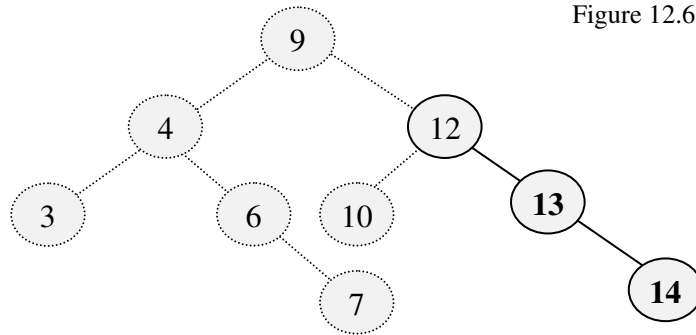


Figure 12.6.

Comparison 3:

Since 14 is larger than 13, we know that if the value 14 is on the list, it must be to the right of 13; we can ignore the value 13 and all values to the left of 13 (in this case, there aren’t any).

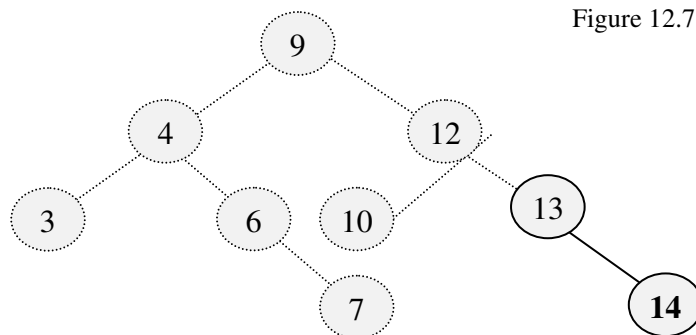


Figure 12.7.

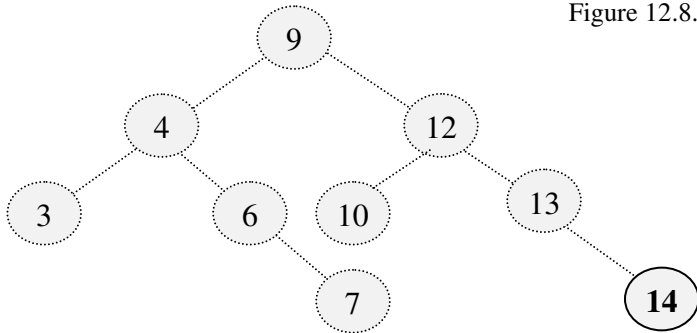


Figure 12.8.

Comparison 4:

We have found the value we were looking for.

This is the same as a binary search!! We found the element in 4 comparisons, just as we did with a binary search!! Will it always be the same??

In this case, yes. But as we have seen before, nothing is entirely free. It is not always possible to construct binary trees such that the number of comparisons needed will be the same as those needed for a binary search. There are also storage, maintenance and programming concerns, but we will address those later in this chapter.

How do we go about constructing a binary tree??

We will see that shortly, but before we do, we need to go over some terminology.

Binary Tree Terminology

Binary trees are associated with some specific definitions, and often rely on a unique set of terms.

- A tree is a *hierarchical* structure.

A hierarchy is a group of persons or things arranged by rank, class, size, or other category.

In our tree, the numbers are arranged by magnitude or size: *Smaller* numbers are to the left and *larger* numbers are to the right.

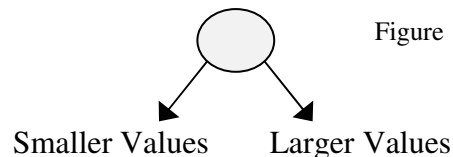


Figure 12.9.

- **The top of the tree is referred to as the root.**

Def

In the example we used above, the value **9** is at the top of the tree (**root**). As noted in our previous definition, because a tree is a hierarchical structure, all numbers smaller than 9 can be found to by following the path(s) to the left; all numbers greater than 9 can be found in the path(s) to the right.

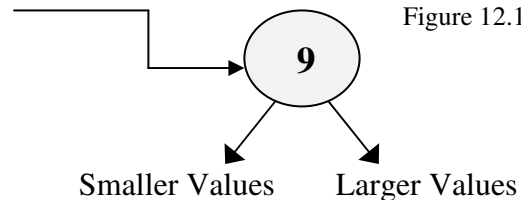


Figure 12.10.

- **Each element (record) in the list is referred to as a node.**

Def

This includes the root, which may also be referred to as the **root node**.

- **Each node in the tree can have (at most) two children. Any node which has a child is a parent**

Def

This is a **binary** situation. Each **node** can have either 0, 1, or 2 **children** (one might say that this is really a **ternary** situation, but we will forgo that discussion). In our example, we saw instances of each of the possibilities. For example, the **nodes** containing the values 9, 4, and 12 each had two **children**. The **nodes** containing the values 6 and 13 had one **child** each. The **nodes** containing the values 3, 7, and 14 had no **children**.

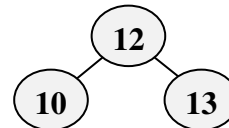
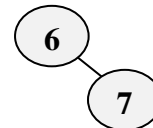


Figure 12.11.

The node containing the value **12** has 2 children



The node containing the value **6** has 1 child



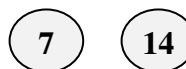
The node containing the value **14** has no children

- **A node which has no children is a leaf.**

Def

Sometimes a **leaf** is also referred to as a terminal node, or as **leaves**, if there is more than one **leaf**.

Figure 12.12.



The nodes containing the values **7** and **14** are both leaves

- **Children of the same parent are said to be siblings.**

Def

For our example, the **nodes** containing the values 3 and 6, and 10 and 13 were each other's **siblings**.

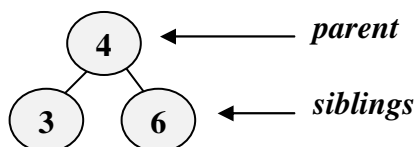


Figure 12.13.

- A child can have at most one parent.**

Def

Self-evident; we *do* live in a monogamous society.

- The nodes superior to a child are their ancestors**

Def

This depends on how it is viewed. In Figure 12.14., the *node* containing the value **9** is the ancestor of the *nodes* containing the values **12**, **10**, **13**, and **14**. The *node* containing the value **12** is the ancestor of the *nodes* containing the values **10**, **13**, and **14**. The *node* containing the value **13** is the ancestor of the *node* containing the value **14** (as well as being the *parent* of that node).

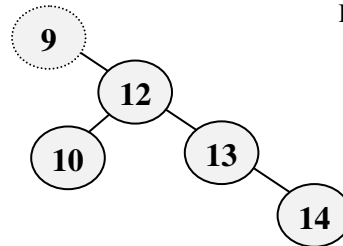


Figure 12.14.

- The nodes inferior to a parent are that node's descendants.**

Def

Once again, this is dependent upon the *nodes* we are addressing. In Figure 12.15., the *nodes* containing the values **4**, **3**, **6**, and **7** are the descendants of the *node* containing the value **9**. The *nodes* containing the values **3**, **6**, and **7** are the descendants of the *node* containing the value **4**. The *node* containing the value **7** is the descendant of the *node* containing the value **6**. The first node in an ordered tree is also referred to as the *oldest* child, while the last is the *youngest*.

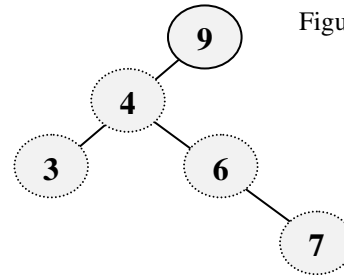


Figure 12.15.

- The level of a node is established by setting the root level at 1 and setting its children to lower levels.**

Def

The *node* containing the value **9** (the *root node*) is at level 1; its ancestors are at levels 2, 3 and 4.

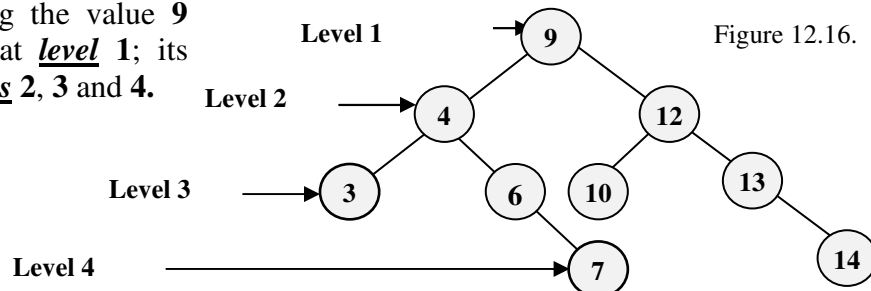


Figure 12.16.

- A tree can be divided into subtrees (which can themselves be divided into subtrees).
Def

For our tree, which has 4 levels, the depth, or height, is four (4)

- A tree can be divided into subtrees (which can themselves be divided into subtrees).
Def

For our example:

Full Tree:

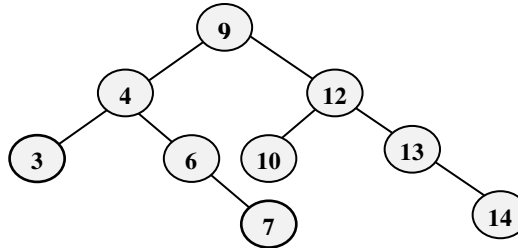
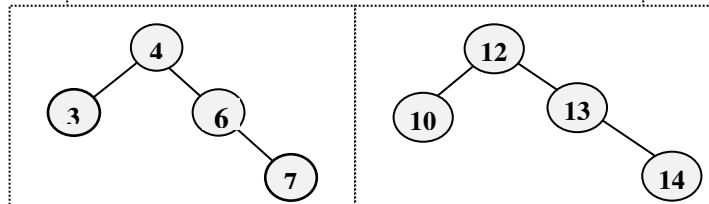


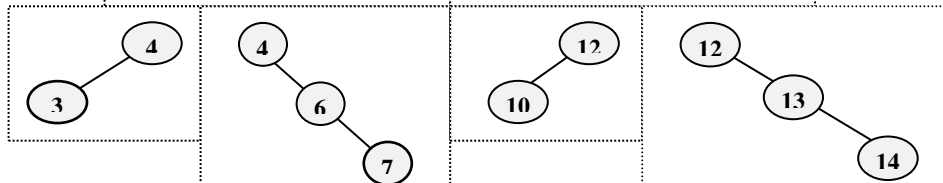
Figure 12.17.

Subtrees:

Levels 2 – 4:



Levels 2 – 3:



Levels 2 – 4:

Levels 3 – 4:



- A forest is the set of disjoint (separate) trees.
Def

For example, we get a *forest* if we remove the root of a tree. In this case, we also have *subtrees*: The *left subtree* and the *right subtree*.

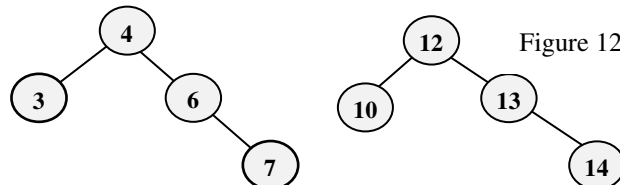


Figure 12.18.

left subtree

right subtree

- **The *degree of a tree* is the maximum degree of nodes in a tree.** ← Def

For our example, the tree has a degree of 3 (levels minus the root).

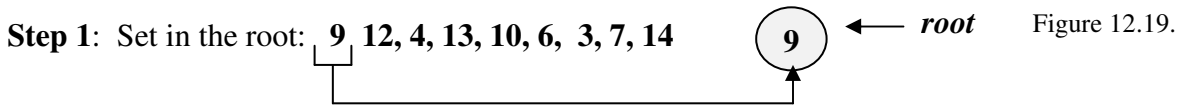
- **A tree is *bushy* if it has (relatively speaking) many leaves** ← Def

As we will see later, this is (generally) a good quality. **What Terms must I Know ?**

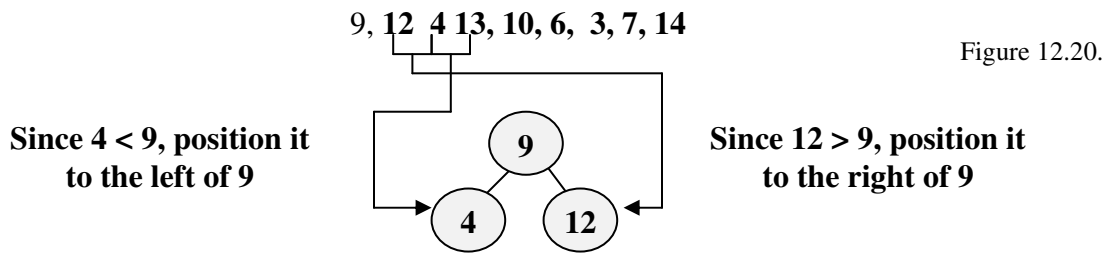
⊙ **NOW, How do we go about constructing a binary tree??**

Constructing Binary Trees

Constructing a binary tree is relatively simple, but constructing a “good” tree (we will see examples of “good” trees vs. “bad” trees later) is often difficult. The manner in which we established our tree was perhaps the simplest way (although we admit that we ‘rigged’ the order in which they were placed in the unordered list). We took the first element in the list and established it as the root. We then took each of the remaining elements (in order; we’ll add them on two at a time to save space) and determined where we should put them (smaller values to the left; larger values to the right):



Step 2: Taking each other element in the list, put it in its appropriate position in the tree. In other words, elements less than the root should be to the *left* of it; elements greater than the root should be to the right of it.



Step 3: Continue until all elements have been added

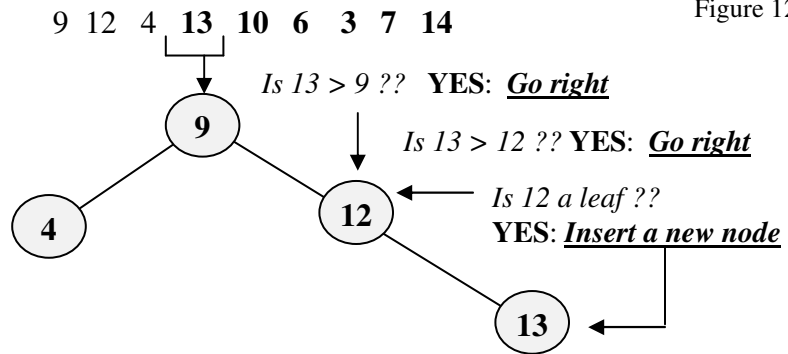


Figure 12.21.

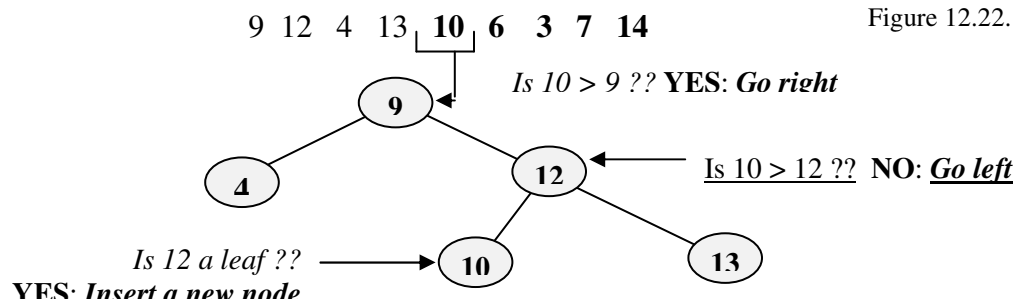


Figure 12.22.

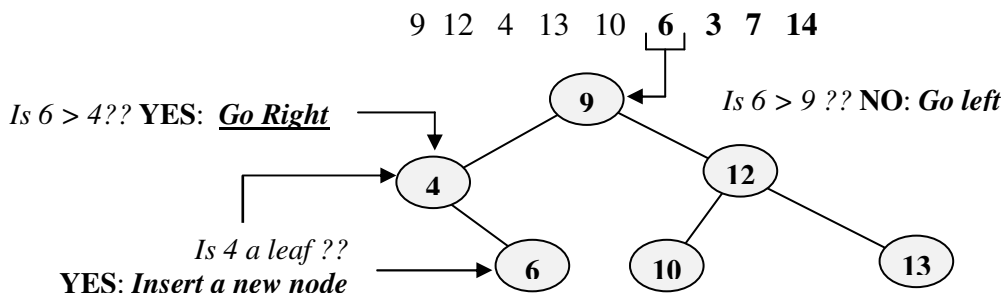


Figure 12.23.

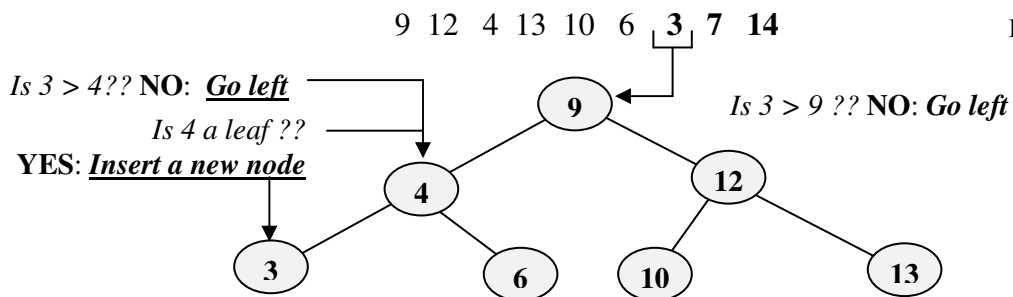


Figure 12.24.

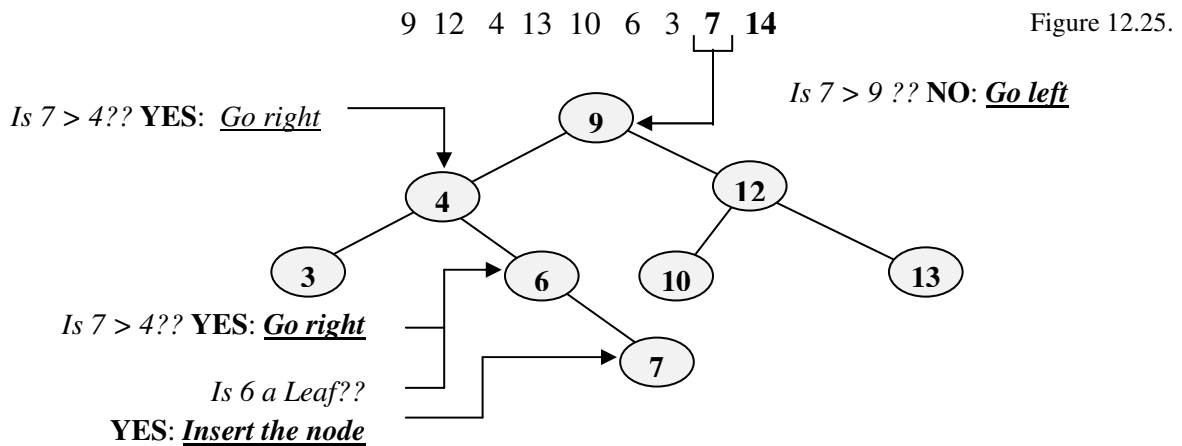


Figure 12.25.

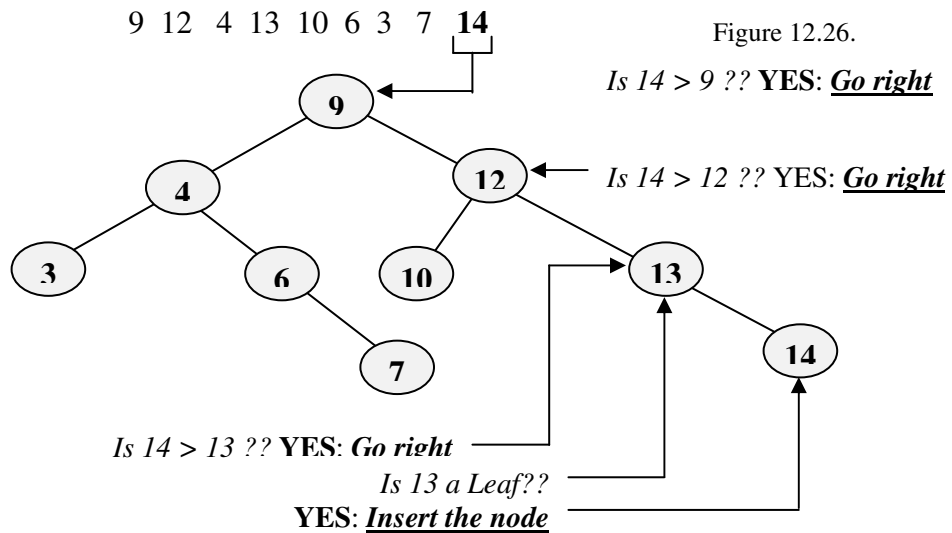


Figure 12.26.

✓ **What is a Binary Tree ?** And the tree is complete.

⊙ **How are the nodes actually linked together??**

Using Pointers in Binary Trees

We already know that trees require pointers, and that the inferior nodes in a subtree are either to the left (if they are smaller) or to the right (if they are greater). Therefore, each node must have *two* pointers: one which points to nodes smaller than it, and one which points to nodes greater than it. Let's call the pointer which points to smaller nodes the *left* pointer, and the pointer which points to nodes greater than it the *right* pointer. If the *left* pointer is NULL, there are no nodes smaller than it in that branch. If the *right* pointer is

NULL, there are no nodes in that branch greater than it. If both the left and right pointers are NULL, the node is a *leaf*.

Consider the following node structure and variable declaration:

C/C++ Code 12.2.

```

struct tree
{   int value;
    struct tree * left, * right; };
int main()
{   struct tree * node, * root;
    
```

In this case, each record will require $2 + 4 + 4 = 10$ -bytes of storage. We will also need two pointer variables: *root*, which will point to the root node, and *node*, which will point to the node we are considering.

If we were display the structure in hierarchical form, it might appear as:

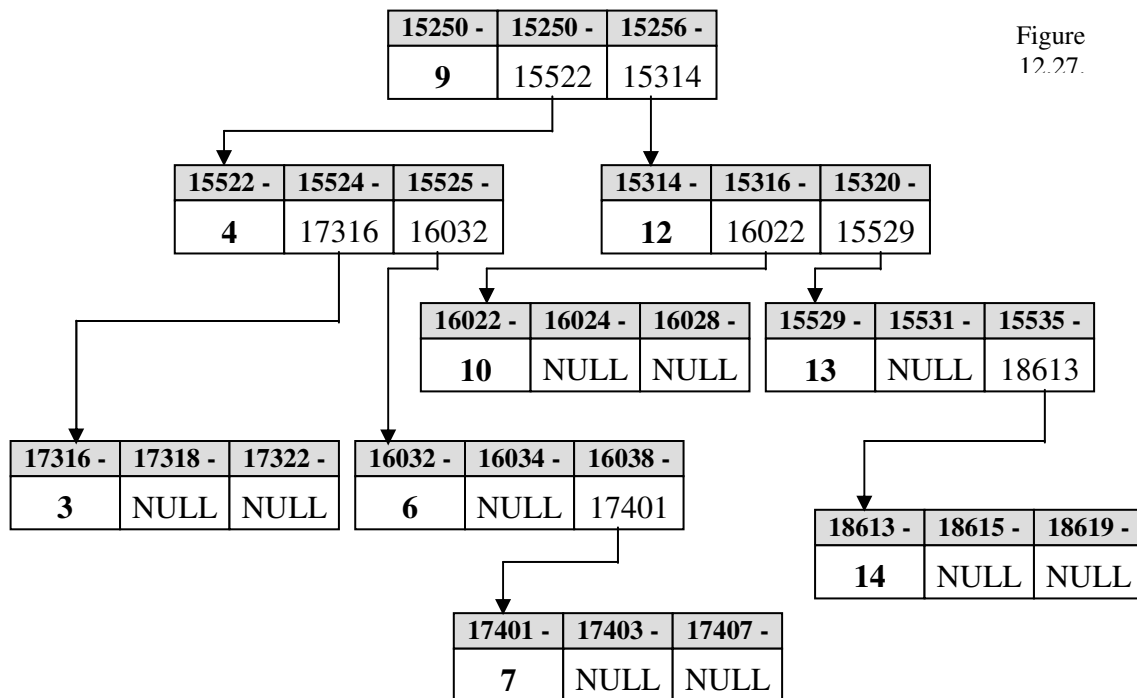
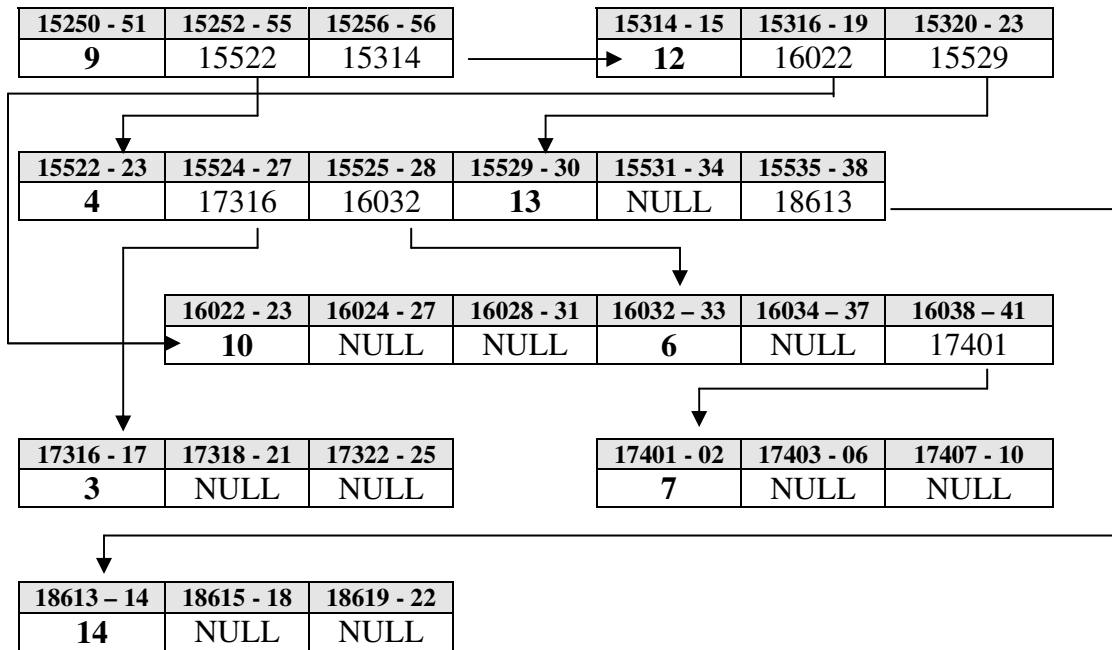


Figure 12.27.

More realistically (since memory is not hierarchical), to store our tree in RAM (as we did as a simple linked list in Figure 12.1.) we might see (Figure 12.28):

Figure 12.28.



✓ Why do we need pointers??



How do we construct the list in C ??

Inserting Nodes into a Binary Tree in C

The procedure follows the steps we used above:

Constructing A Binary Tree

1. Get the first value, dynamically allocate enough RAM for it, store the value, and set the nodes pointer's to NULL. Since this is the 1st node, make it the root.
2. Get the next value. dynamically allocate enough RAM for it, store the value, and set the nodes pointer's to NULL.
3. Starting with the root, compare the value with every node in the tree.
 - a. If the new value is larger than the tree node value, follow the right pointer
 - b. If the new value is smaller than the tree node value, follow the left pointer
4. Continue until the pointer is NULL.
5. Have the NULL pointer point to the new node's address
6. If there are more records, go to step 2.

The relevant code necessary is given in C Code 12.3. The only difference is that we will prompt the user for an integer to insert, and then set up the appropriate links.

C/C++ code 12.3.

```

struct tree                                // Our structure template
{ int   value;                             // The number
  struct tree * left, * right; }           // the pointers

void main()
{ int input, numnodes = 0;
  struct tree * node, * root, * present, * previous; // Our pointer variables
  char number[5];                          // for the user input
  do { printf("Enter an integer to add, Or enter 0 (zero) to quit: "); // Prompt
      if ((input = atoi(gets(number))) > 0); // Get input and convert
      { node = (struct tree *) malloc(sizeof(struct tree)); // Allocate memort
        node -> value = input; // Store value
        node -> left = node -> right = NULL; // initialize pointers to NULL
        if ( numnodes++ == 0) root = node; // If the first entry, make it the root
        else // Otherwise ...
        { present = root; // start at the root, and continue
          while (present != NULL) // until we point to NULL
          { previous = present; // store addr. of the node in the tree
            if (node -> value > present-> value) // is the new value larger?
              present = present -> right; // then go to the right
            else present = present -> left; } // new value smaller?? Go Left
            if (node -> value > previous -> value) // Should we go to the right?
              previous -> right = node; // set in the right pointers ..
            else previous -> left = node; } // .. or set in the left pointers.
          } while (input != 0); } // Keep going until 0 entered

```

As we did when we set up our linked lists, we use the pointer variable *present* to keep track of a node already on the tree, and the pointer variable *previous* to keep track of the node which points to the existing node.

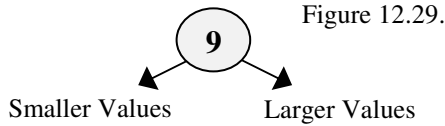
What C/C++ Code is needed??



How do we find a node in a binary tree ??

Searching a Binary Tree

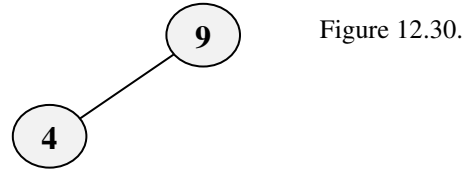
Searching a binary is very similar to our binary search procedure. Suppose we were looking for the value 6 in our tree.



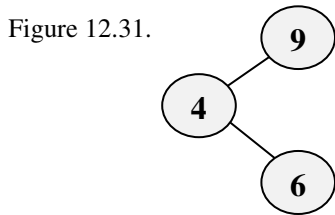
Step 1: Start at the *root* and determine which way to go (unless 6 is at the root): Since smaller than 9, go left.

Step 2: Compare the value at the address contained in the previous pointer. There are four possibilities:

1. The value was found
2. The previous Address was NULL (the value is NOT on the list).
3. The value is greater than the value we are searching for: Go Right.
4. The value is smaller than the value we are searching for: Go Left.



Repeat Step 2 until the record is found or determined not to be on the list.



In our case, since the value 6 is greater than 4, we would go right (where we would find the value we are looking for).

The c code necessary to find a record is also quite simple. In C Code 12.4. we provide the relevant code necessary to find a (user-inputted) value in the tree. We also add one variable (**int nsearches**) which will count (and print out) how many comparisons were necessary to find an element in the tree.

Since we know that using a binary search on a sorted array of this size requires a maximum of:

$$\lceil \log_2 n \rceil = \lceil \log_2 9 \rceil = \lceil 3.169925 \rceil = 4 \text{ comparisons}$$

and an average of $\log_2 n - 1$ or 2.17 comparisons, we are interested in how our binary tree search compares to a binary search. Table 12.1. gives the output from our program, assuming we were searching for the integers from 1 through 15 (inclusive).

C code 12.4

```

int nsearches = 1;                // we always need at least 1 comparison
printf("Enter a number to search for: ");
input = atoi(gets(number));
node = root;                       // start at the top
while ((node != NULL) && (node -> value != input))    // while not found
{ if (input < tree -> value)        // value smaller??
    node = node -> left;           // then go left
  else                             // value greater??
    node = node -> right;
  nsearches++;
}
if (node == NULL)                // NULL encountered??
  printf("The number %d is not on the list (%d searches required)\n",
        input, nsearches);
else                             // otherwise, it was found
  printf("The number %d was found in %d searches\n",
        node -> value, nsearches);

```

The above program produces the output:

Table
12.1.

Search Value	Sequential Searches Needed	Binary Searches Needed	Binary Tree Searches Needed	Search Value	Sequential Searches Needed	Binary Searches Needed	Binary Tree Searches Needed
1	10	3	5	9	1	1	1
2	10	3	5	10	5	3	3
3	7	3	3	11	10	4	4
4	3	2	2	12	2	2	2
5	10	3	4	13	4	3	3
6	6	3	3	14	9	3	4
7	8	4	4	15	10	4	5
8	10	4	5	16	10	4	5
AVE:					7.2	3.1	3.6

Even with a list as small as ours, the number of comparisons needed in for a binary tree search is considerably less than for a sequential search. The number of comparisons needed versus a binary search, however, is slightly more.



Why is it more?? I thought it was the same as a binary search !!

Not quite. For one thing, in a binary search, we can find out if a value is not on the list when we determine that it is not where it supposed to be. In a binary tree, we have to *traverse* the tree and find a NULL pointer

Traverse
To turn; as in a pivot

✓ **How do we search Binary Trees??** before we can determine that the value is not where it should be.

⊙ **Are there other searching disadvantages??**

Yes, some. We admitted early on that our list was contrived. What we meant was that we selected the values, and the order in which they were placed in the tree in a fashion which makes the tree optimal. Suppose that we had chosen the following list: **3, 4, 6, 7, 9, 10, 12, 13, 14.**

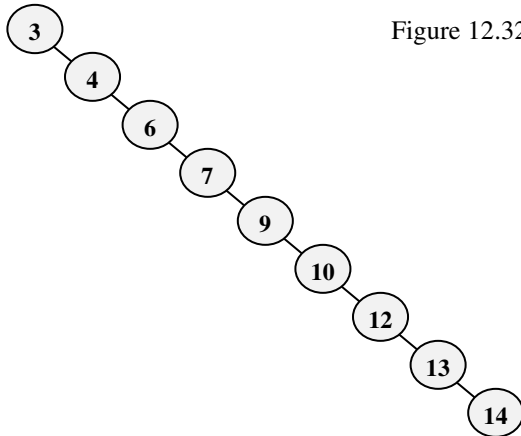


Figure 12.32. This list contains exactly the same elements, BUT it is already in order. If we were to follow the same procedure in constructing the binary tree as we did before, it would appear as it does in Figure 12.32.

Notice that if we were to attempt to find a value in this tree, we would have no better a comparison rate than we would for a simple sequential search. In fact this is nothing more than a simple linked list.

As we mentioned in our definitions section, a good tree is a ‘bushy’ tree. This one looks pretty spindly.

⊙ **Is maintaining the tree as easy as linked lists, or as difficult as sorted arrays??**

A little bit of both.

Inserting Nodes into Binary Trees

Adding a node to a binary tree is generally not a problem, at least if we are only adding a few. Suppose that we wanted to add nodes containing the values 11, 5 and 2 to our tree. It would now appear as:

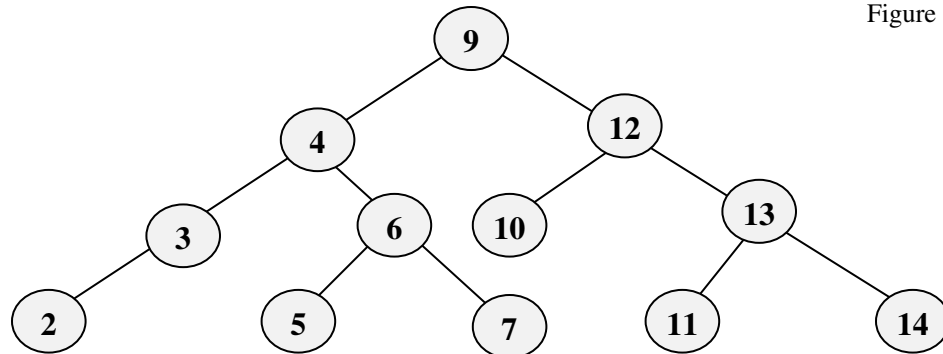


Figure 12.32.

Not a problem. Our tree is actually ‘bushier’ than before. However, if we wished to add the values 15, 18, and 19 (instead of the above values) to our tree. It might now appear as:

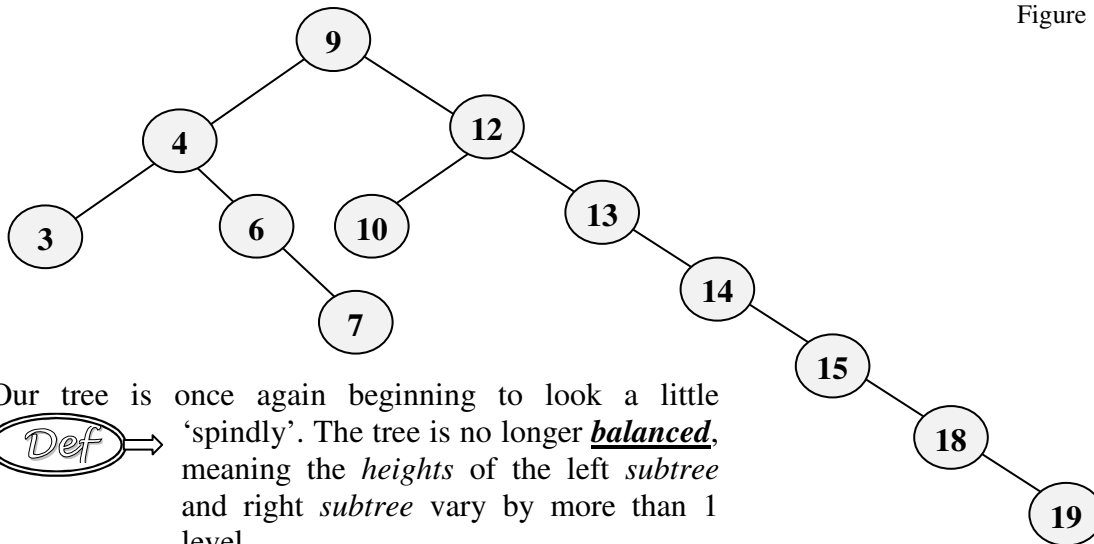


Figure 12.33.

Our tree is once again beginning to look a little ‘spindly’. The tree is no longer ***balanced***, meaning the *heights* of the left *subtree* and right *subtree* vary by more than 1 level.

Optimally, the tree should be rearranged as¹:

¹ Other schemes could also be considered optimal

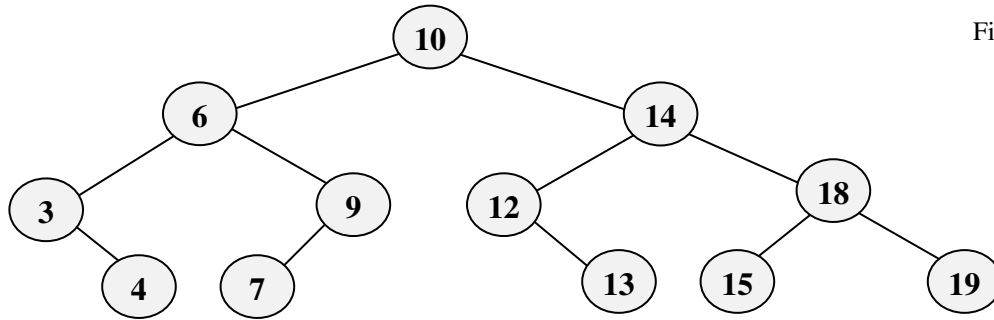
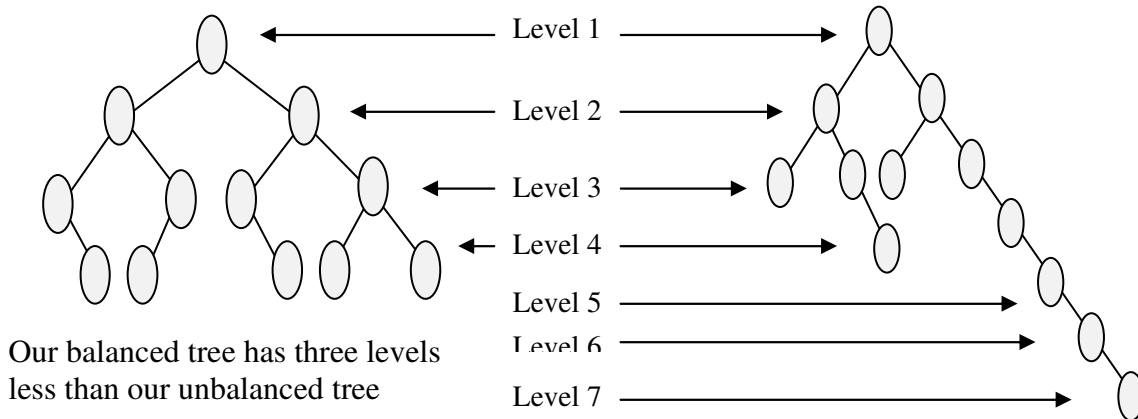


Figure 12.34.

? *What's the difference??*

Consider the following differences in tree height:

Figure 12.35.




? *What difference does it make if the tree is balanced or not??*

As we saw in our ‘worst case’ example, where the tree only had one branch, there is a big difference in terms of the number of comparisons required to find a node. If we used the same search program as we did before (C Code 12.4), and looked at the number of comparisons needed for all integers between 1 and 20, we would find that the following number of comparisons were needed on each of the lists:

Search Value	Unbalanced Tree Searches Needed	Balanced Tree Searches Needed	Search Value	Unbalanced Tree Searches Needed	Balanced Tree Searches Needed
1	5	5	11	4	4
2	5	5	12	2	3
3	3	3	13	3	4
4	2	4	14	4	2
5	4	5	15	5	4
6	3	2	16	7	5
7	4	4	17	7	5
8	5	5	18	6	3
9	1	3	19	7	4
10	3	1	20	8	5
AVE:				4.4	3.8

This may not seem like much of a difference (even though 3.8 is *How do we Insert Nodes into a Binary Trees??* 70% of 4.4), but once again, it is because our list is small. In point of fact, it is a substantial savings.

 **How do we go about balancing the tree??**

There are procedures for balancing the tree – We will see these in the next chapter.

 **How do we delete nodes from the tree??**

Deleting Nodes from a Binary Tree

As with adding nodes from the tree, it depends on which node(s) we wish to delete. Suppose that we wish to delete the elements 7 and 14, such that the tree would appear as:

Once again, not a problem. But that is because the nodes containing the values 7 and 14 were leaves, and thus didn't point to any other nodes. The only change we would have to make is to have the pointers which contained the addresses for the nodes 7 and

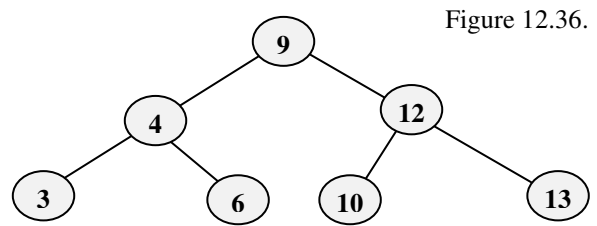


Figure 12.36.

14 (in this case, 6's and 13's right pointers point to NULL). Of course, we could then free the memory which we have been using to store the nodes.

In other cases, it is almost that easy. If, for example, we wished to delete the node containing the value 6 (from our original tree), all we would have to do is have 4's right pointer point to 7 (again, we would free the RAM allocated to hold the node containing the value 6). The tree would appear as it does in Figure 12.37.

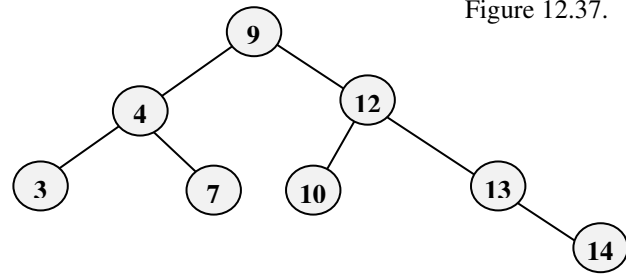


Figure 12.37.

Suppose, however, that we wanted to delete the root. The root has two children, each of whom also have two children. If we try and make the node containing the value 12 the root (which is in the right subtree), how do we include the left subtree? We could, perhaps have 10's left pointer point to 4, which yields the tree given in Figure 12.38., but if we did, it is quite obvious that the tree is not at all balanced.

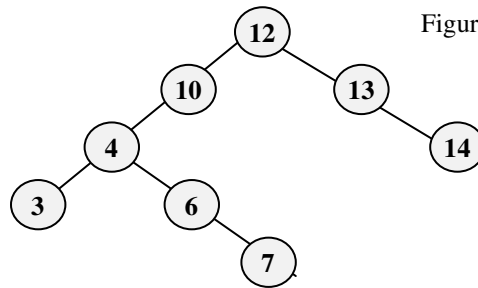


Figure 12.38.

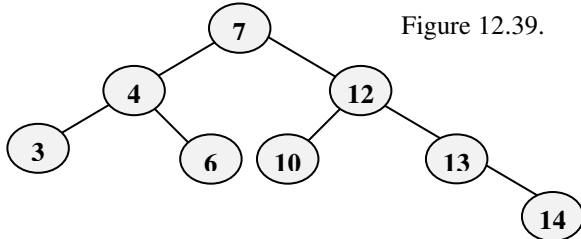




Figure 12.39.

Our only solution is to once again re-structure the tree (balancing as we do), so that it appears as it does in Figure 12.39

How do we Delete Nodes from Binary Trees??

 **What about the procedures necessary to do this??**

As with balancing binary trees, we will see this in the following chapter.

 **How do we list the nodes of a binary tree in order??**

In order to traverse a binary tree, we need to apply a technique called *recursion*.

Recursion

We have actually seen the concept of recursion previously. If you will remember, the recursion was necessary for the quicksort routine (see Chapter 9).

We Already Know.

Quicksort is the fastest internal sort method

- A recursive function is one which calls itself. Def

When a function calls itself, it ‘pops-down’ a level. When it returns from a call, it ‘pops-up’. Initially, this may seem like a strange concept. **What is Recursion??** **Why would a function call itself??**

In fact, it is a very useful concept (and as we will see, the easiest way in which to traverse a binary tree). Before we do that, however, let’s take a look at a simple recursive program. Let’s write a program which will find the factorial of a number². We know that a factorial is:

$n! = n * (n - 1) * (n - 2) \dots * 1$ Where n is the number we wish to find the factorial of

For example, $5! = 5 * 4 * 3 * 2 * 1 = 120$. The recursive code needed might appear as:

C Code 12.5

```

#include <stdio.h>
int factorial(int number);
void main()
{ int i = 5;
  printf("The factorial of %d = %d\n",i, factorial(i));
}
int factorial(int number)
{ if (number == 1)
  return 1;
  else
  return (number * factorial(number - 1)); }

```

Notice that function **main** calls function *factorial* once, passing it the value 5 which will be stored into variable *number* in the function. In function *factorial*, we check to see if the integer passed (the contents of location *number*) is the numeric value 1, otherwise, calls itself, passing the value one less than it received. In other words, each time factorial is

² This program could be written without using recursion

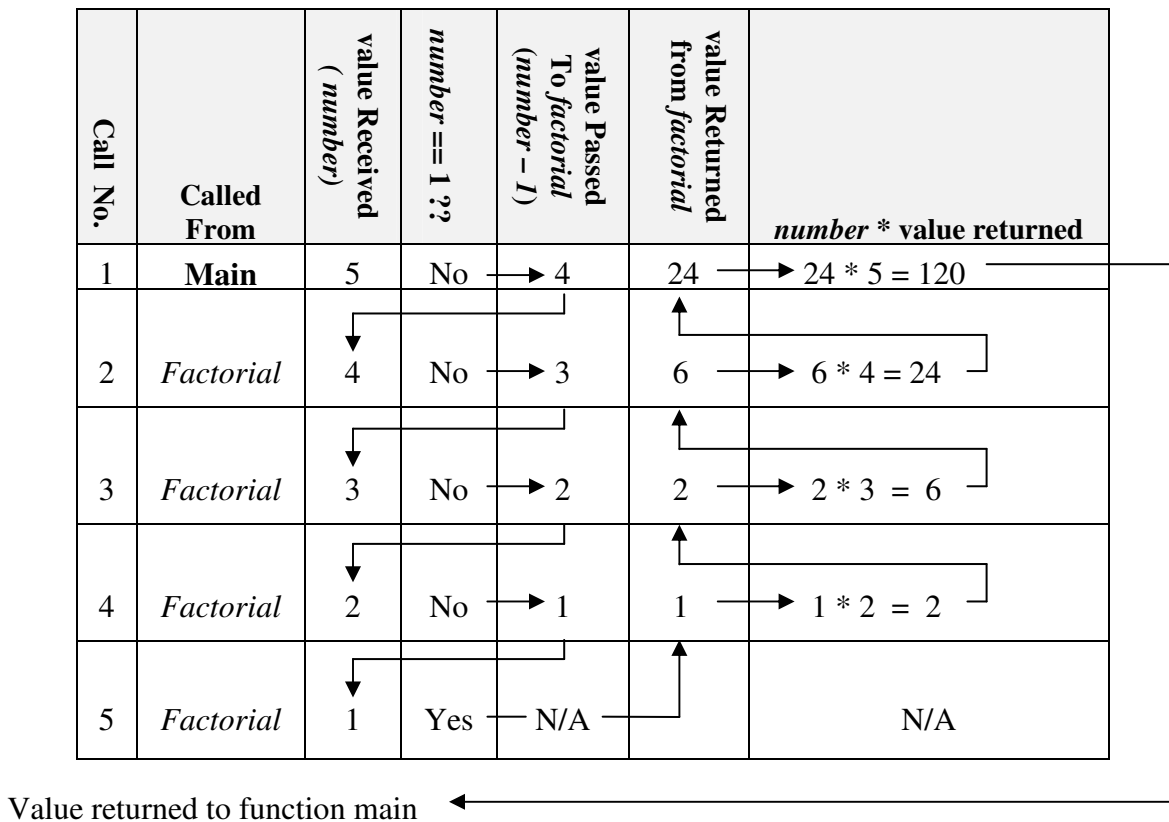
called, it receives the number 5, 4, 3, 2, and 1. When it starts returning values, it returns 1, 2 (1 * 2), 6 (2 * 3), 24 (6 * 4), and finally 120 (24 * 5).

It sounds confusing at first, but lets follow each of the variables each time the function factorial is called. (see Figure 12.40.). In the figure, the arrows pointing down show the values passed as the function ‘pops-down’. The arrows pointing up show the values passed as the function ‘pops-up’. Arrows pointing to the right are the commands being executed by the function while it is at that level.

Let’s start when the function is first called from the function main: *factorial(i)*

In main, I contains the value 5, so that is the value we initially pass to function *factorial*.

Figure 12.40.



In the 1st call to function *factorial*, *number* receives the value 5. We can thus overlook the if statement (**if** (*number* == 1)) and call *factorial* again, this time passing it the value *number* -1 (or the integer 4). As we can see from the above figure, the same is true for the 2nd through 4th calls (where we are ‘popping-down’) to *factorial* where the values 3, 2, and 1 are passed. In the 5th call to function *factorial*, the integer 1 is passed to *number* and we execute the statements:

```

if (number == 1)
    return 1;
    
```


We Already Know.

Upon return from a call to a function, the statement immediately following the call is executed

At which time we stop popping down and start popping up. For our program, this is when all of the calculations will be made. Upon our return to the 4th call to function factorial, we execute the statement: **return** (*number * factorial(number - 1)*);. In other words, the 5th call (our last pop-down) returned the integer 1. Therefore the statement **return** (*number * factorial(number - 1)*) is equivalent to the statement **return** (*number * 1*), where the value of *number* is 2, and the expression evaluates to 2 * 1 or 2.

The same procedure is followed each time the function pops-up (see the rightmost 2 columns in Figure 12.40.). Eventually, when the function pops-back to the main function, the value 120 (5!) is returned. How does Recursion Work??



OK, what does this have to do with listing the nodes of a binary tree in order??

Traversing a Binary Tree

First let's look at the placement of the nodes in our binary tree. Let's also assume that we wish to list the values in ascending order (from smallest to largest). Therefore, if we wish to find the 1st (smallest) element on a list, we have to go as far left as possible. Similarly, the largest element on the list will be as far right as we can go. Our basic algorithm would then be:

We Already Know.

In a binary tree, smaller values are to the left and larger values to the right

Traversing a tree in order

1. 'Pop-down' to the left as far as possible. When we can't go any further, print the contents of the node
2. When we 'Pop-up', print the contents of the node.
3. If we can go right, go right, but go left if we can.
4. Repeat steps 1 – 3.



That's it ??

Essentially, yes. Consider our tree (duplicated in Figure 12.41.). If we go as far left as we can, we find the smallest value (3). When we pop back up, we find the second smallest number (4). We go right (if we can), and then try and go left. When we

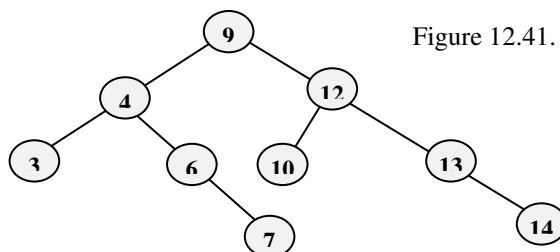


Figure 12.41.

can't, we pop up, and print the contents of the node (6). We then again try and go right (to the node containing the value 7) try to go left (which we can't) and then print the contents of the node upon our return. The procedure continues until we have printed out the entire list: **3 4 6 7 9 10 12 13 14**

How do we really do this ??

Consider the *recursive* function *printtreeinorder* given in C Code 12.6. This code assumes that the data type **struct tree** has associated with a variable *root* (where *root* will hold the address of the *root* node). To display the *tree* in order, all we need to do is pass the address of the *root*:

C/C++ Code 12.6.

```

struct tree
{ int value;
  struct tree * left, * right; };

void main ()
{ struct tree * root, .....
  ...
  printtreeinorder (root);
  ... }

void printtreeinorder(struct tree * node)
{ if (node == NULL)
  return;
  printtreeinorder(node -> left);
  printf("%4d",node -> value);
  printtreeinorder(node -> right);
}
    
```

Function *preinttreeinorder* keeps passing the address of its left pointer until the address is NULL. Upon return (pop-up), it prints the value of the node. The first print would occur as shown in Figure 12.42.

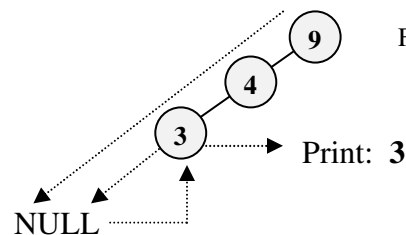
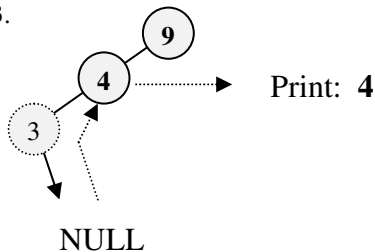


Figure 12.42.

Figure 12.43.



After printing the value, the function again calls itself, this time passing the address of the node to its right. However, since the address is NULL, it returns. Because this was the last statement in the function, it pops-up to the previous call, and again prints the value of the node

The function again calls itself passing the address of the right node. Since the address is not NULL, it immediately passes the address of the left node, continuing until the address is NULL. Upon return, it prints the value of the node.

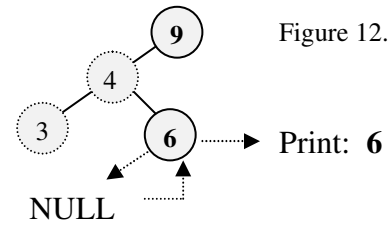


Figure 12.44.

The procedure repeats until all the nodes have been printed.

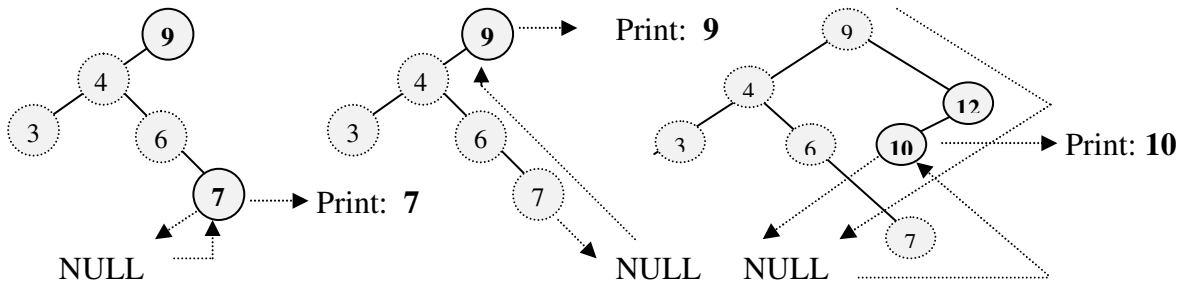
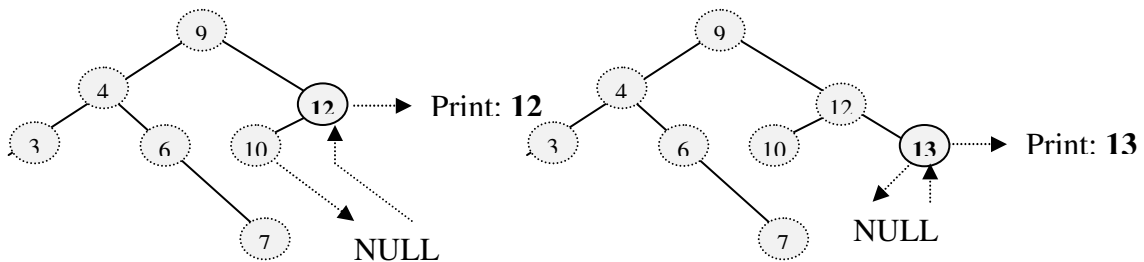
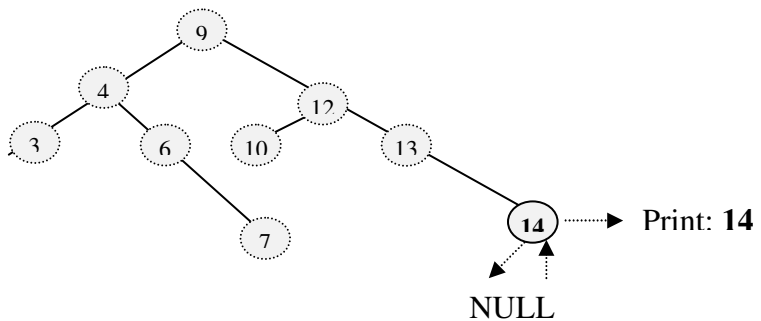


Figure 12.45.



And the final node is printed out. The program will test to see if the node's right pointer is NULL, and since it is, it will pop-up.



How do we Traverse a Tree??

What C/C++ Code is needed??

? Could we print out the tree in a different order ??

Yes, we could print out the tree in reverse order, for example, simply by trying to right first, and then going left when we couldn't go any farther. We could also print the tree from top (i.e., from the root) to bottom (i.e., to the leaves), either going left first, or right first. For our purposes, however, all we need to know is how to print out the tree in order.

Summary

When we began our discussion about searching and sorting, we noted that there were trade-offs involved. Through the use of linked lists, we saw that we were able to maintain some order to our lists, as well as to take advantage of dynamic memory allocation, but finding an element on the list still (basically) required a sequential search. Binary trees are data structures which allow us to use linked lists and to find elements quickly.

Still, there is a price to pay for our flexibility. As we saw, maintenance can be troublesome:

Problems with Binary Trees

1. Adding elements onto a binary tree can quickly lead to a tree losing its 'bushiness' and therefore reducing its searching advantages.
2. Deleting nodes from a binary tree may require a restructuring of the tree.
3. Restructuring a tree can become tedious and complex.

In the next chapter, we will consider some additional trees, and will examine procedures for maintaining balanced trees.

Chapter Terminology: Be Able to Fully Describe these terms

Ancestor	Node
Balanced Tree	NULL Pointer
Binary Tree	Parent
Branch	Popping-down
Bushiness	Popping-up
Children	Printing a tree in order
Degree of a Tree	Recursion
Depth (or Height)	Right Pointer
Descendant	Root
Forest	Sibling
Hierarchical Structures	Subtree
Leaf	Traversing a tree
Left Pointer	Tree

What else should I know??



Review Questions

1. Explain the advantages and disadvantages of binary trees.
2. Given the list of integers: 30, 67, 12, 7, 45, 38, 22, 9, 34, 33, 32
 - A. Construct a binary tree using the method discussed (i.e., the first element becomes the root, the others are added accordingly)
 - B. Identify the left subtree
 - C. Identify the right subtree
 - D. Identify all of the leaves
 - E. Identify all the ancestors of the node containing the value 9
 - F. Identify all of the descendants of the node containing the value 45
 - G. Show the depth of the tree
 - H. Is the tree balanced? Why or why not?
3. For the problem above, assume that the following structure was used:

```

struct tree { int value;
                struct tree * left, * right; };
```

If we look at RAM we see:

5000 - 5015	5016 - 5079	5080 - 5084	5085 - 5178	5179 - 5212
Available	Unavailable	Available	Unavailable	Available
5213 - 5286	5287 - 5339	5340 - 5361	5362 - 5484	
Unavailable	Available	Unavailable	Available	

Assume that RAM is assigned on a first-come-first-served basis.
Show ALL of the pointer assignments.

4. Given the list: **Anita, Juan, Tamara, Olaf, Radhika, Chang, Lambros, Colleen, Marje, Zeke, James**

Set up an OPTIMAL binary tree to print our the names Alphabetically.

5. What is a recursive function?

Review Question Answers (NOTE: checking the answers before you have tried to answer the questions doesn't help you at all)

1. Explain the advantages and disadvantages of binary trees.

Advantages

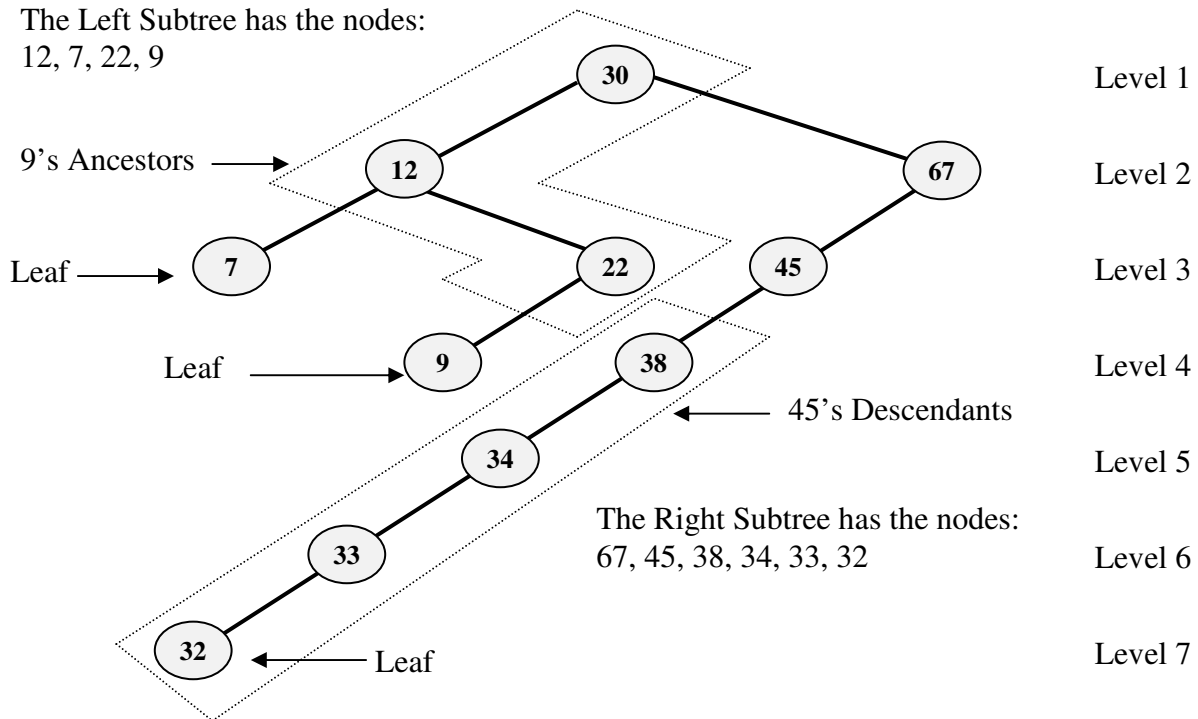
Linked Lists
Dynamic Memory Allocation
Quick Searches

Disadvantages

Maintenance (see list in Summary)

2. Given the list of integers: **30, 67, 12, 7, 45, 38, 22, 9, 34, 33, 32**

The Left Subtree has the nodes:
12, 7, 22, 9

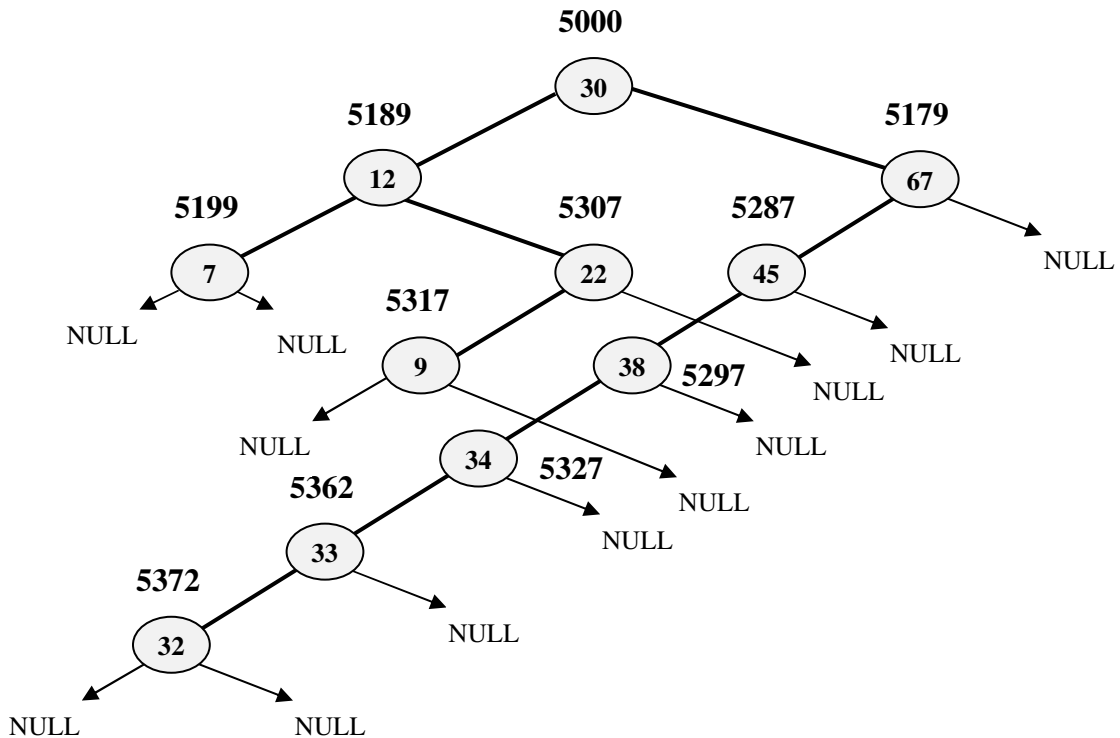


The Tree is NOT balanced since the right subtree has 7 levels and the left subtree has only 3

3. For the given structure, each node will require a total of $2 + 4 + 4 = 10$ -bytes of contiguous storage. Therefore, we must examine RAM to see where we can assign each of the nodes on the list:

Locations	Bytes Available	Assignments
5000 – 5015	16	Node '30' Base address: 5000
5080 - 5084	5	NONE
5179 - 5212	33	Node '67' Base address: 5179 Node '12' Base address: 5189 Node '7' Base address: 5199
5287 – 5339	53	Node '45' Base address: 5287 Node '38' Base address: 5297 Node '22' Base address: 5307 Node '9' Base address: 5317 Node '34' Base address: 5327
5362 – 5484	123	Node '33' Base address: 5362 Node '32' Base address: 5372

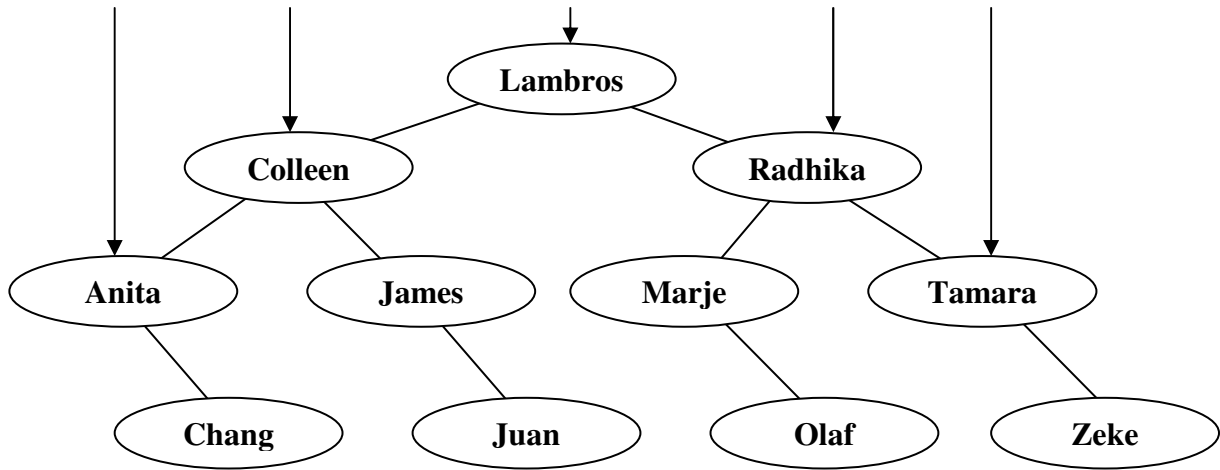
The Addresses assigned to each of the pointers are:



4. Given the list: **Juan, Anita, Tamara, Olaf, Radhika, Chang, Lambros, Colleen, Marje, Zeke, James**

The list can be stored optimally if we would store the list as if we were performing a binary search on it. Therefore, If we first sort it, we could see the path we would take to find any element on the list.

Anita, Chang, Colleen, James, Juan, Lambros, Marje, Olaf, Radhika, Tamara, Zeke, James



4. A Recursive function is one which calls itself.

C/C++ Programming Assignments

1. Assume the following template:

```
struct telephonetemplate
{ char name[30], tel[11];
  struct telephonetemplate *leftname, *rightname };
```

Duplicate the code given in 12.3., arranging the binary by *name*.

2. Add a function to the above program which will allow you to search for any name on the list (make sure that you put in a check to see if the name is NOT on the list).
3. Add a function to the above program which will allow you to print out all the names in the tree IN ORDER.
4. Add a function to the above program which will allow you to print out all the names in the tree IN REVERSE ORDER.
5. Redo exercise 1 above, BUT instead of adding larger names to the right of the root, and smaller names to the left, add the LARGER names to the LEFT, and the SMALLER names to the RIGHT.
6. Using the same template given above, assume that we also have the additional template and declaration:

```
struct myfriends
{ char friendname[30], friendtel[11]; };
int main()
{ struct myfriends buddies[14] =
  {{“Gable, Clark”, “2025551212”},{“Garbo, Greta”, “3056789012”},
  {“Loy, Myrna”, “1230987654”}, {“Leigh, Vivian”, “2348765432”},
  {“Astaire, Fred”, “3457654321”}, {“Dandridge, Dorothy”, “4566543210”},
  {“Romero, Ceasar”, “5675432109”}, {“Crosby, Bing”, “6784321098”},
  {“Harlow, Jean”, “7893210987”}, {“Taylor, Rod”, “891210987”},
  {“Cooper, Gary”, “9010987654”}, {“Waters, Ethel”, “7169876543”},
  {“Miranda, Carmen”, “8595928374”}, {“ Hope, Bob”, “4657382910”}}};
```

Transfer the names from the array buddies to the binary tree. Make sure the tree is *OPTIMALLY BALANCED*.