Chapter 10: Efficient Collections (skip lists, trees)

If you performed the analysis exercises in Chapter 9, you discovered that selecting a bag-like container required a detailed understanding of the tasks the container will be expected to perform. Consider the following chart:

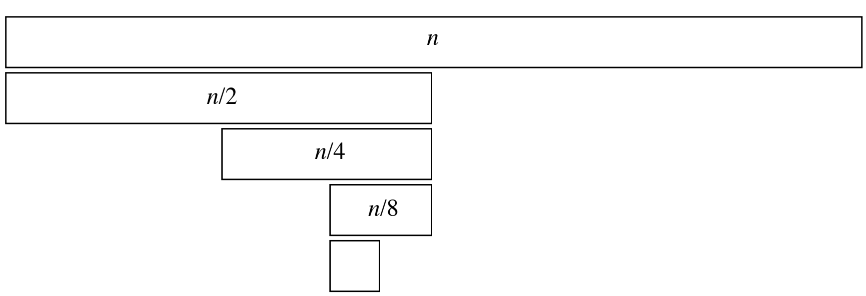
	Dynamic array	Linked list	Ordered array
add	O(1)+	O(1)	O(n)
contains	O(n)	O(n)	O(log n)
remove	O(n)	O(n)	O(n)

If we are simply considering the cost to insert a new value into the collection, then nothing can beat the constant time performance of a simple dynamic array or linked list. But if searching or removals are common, then the O(log n) cost of searching an ordered list may more than make up for the slower cost to perform an insertion. Imagine, for example, an on-line telephone directory. There might be several million search requests before it becomes necessary to add or remove an entry. The benefit of being able to perform a binary search more than makes up for the cost of a slow insertion or removal.

What if all three bag operations are more-or-less equal? Are there techniques that can be used to speed up all three operations? Are arrays and linked lists the only ways of organizing a data for a bag? Indeed, they are not. In this chapter we will examine two very different implementation techniques for the Bag data structure. In the end they both have the same effect, which is providing O(log n) execution time for all three bag operations. However, they go about this task in two very different ways.

Achieving Logarithmic Execution

In Chapter 4 we discussed the log function. There we noted that one way to think about the log was that it represented “the number of times a collection of n elements could be cut in half”. It is this principle that is used in binary search. You start with n elements, and in one step you cut it in half, leaving n/2 elements. With another test you have reduced the problem to n/4 elements, and in another you have n/8. After approximately log n steps you will have only one remaining value.

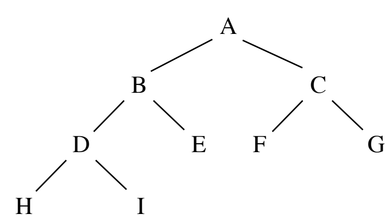


There is another way to use the same principle. Imagine that we have an organization based on layers. At the bottom layer there are n elements. Above each layer is another that has approximately half the number of elements in the one below. So the next to the bottom layer has approximately n/2 elements, the one above that approximately n/4 elements, and so on. It will take approximately log n layers before we reach a single element.

How many elements are there altogether? One way to answer this question is to note that the sum is a finite approximation to the series n + n/2 + n/4 + … . If you factor out the common term n, then this is 1 + ½ + ¼ + … This well known series has a limiting value of 2. This tells us that the structure we have described has approximately 2n elements in it.

The *skip list* and the *binary tree* use these observations in very different ways. The first, the skip list, makes use of non-determinism. Non-determinism means using random chance, like flipping a coin. If you flip a coin once, you have no way to predict whether it will come up heads or tails. But if you flip a coin one thousand times, then you can confidently predict that about 500 times it will be heads, and about 500 times it will be tails. Put another way, randomness in the small is unpredictable, but in large numbers randomness can be very predictable. It is this principle that casinos rely on to ensure they can always win.

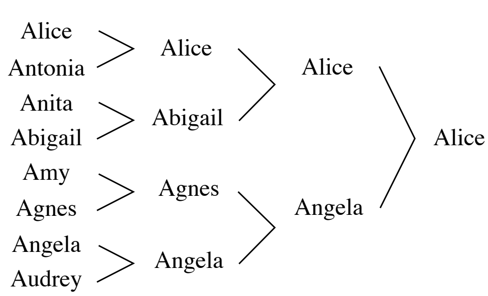
The second technique makes use of a data organization technique we have not yet seen, the binary tree. A binary tree uses nodes, which are very much like the links in a linked list. Each node can have zero, one, or two children.



Compare this picture to the earlier one. Look at the tree in levels. At the first level (termed the root) there is a single node. At the next level there can be at most two, and the next level there can be at most four, and so on. If a tree is relatively “full” (a more precise definition will be given later), then if it has n nodes the height is approximately log n.

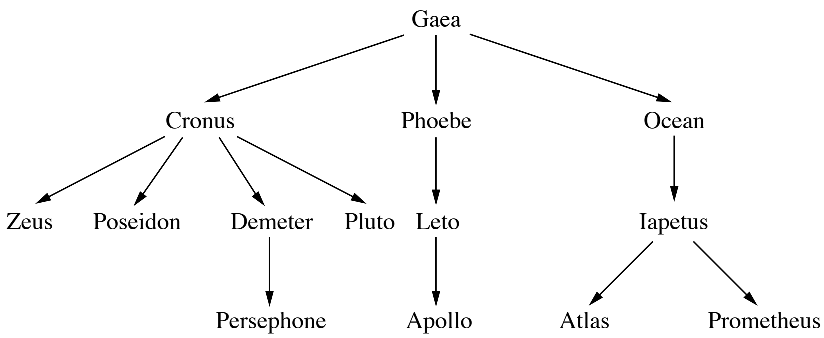
Tree Introduction

Trees are the third most used data structure in computer science, after arrays (including dynamic arrays and other array variations) and linked lists. They are ubiquitous, found everywhere in computer algorithms.

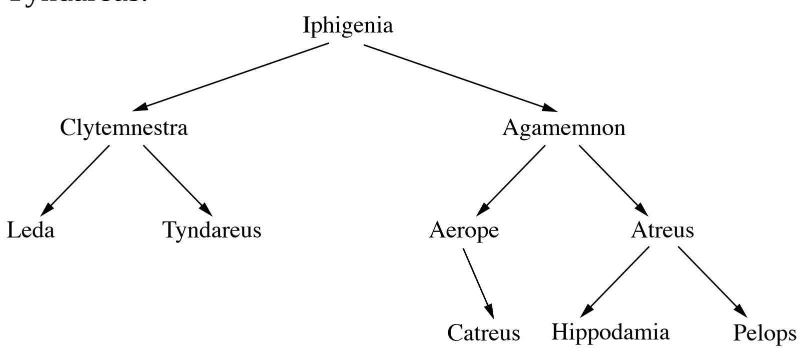


Just as the intuitive concepts of a stack or a queue can be based on everyday experience with similar structures, the idea of a tree can also be found in everyday life, and not just the arboreal variety. For example, sports events are often organized using binary trees, with each node representing a pairing of contestants, the winner of each round advancing to the next level.

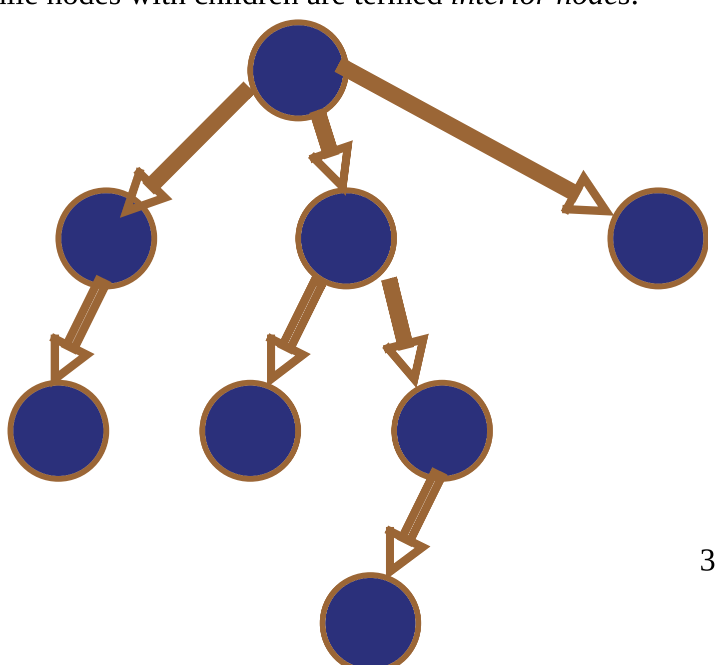
Information about ancestors and descendants is often organized into a tree structure. The following is a typical family tree.



In a sense, the inverse of a family tree is an ancestor tree. While a family tree traces the descendants from a single individual, an ancestor tree records the ancestors. An example is the following. We could infer from this tree, for example, that Iphigenia is the child of Clytemnestra and Agamemnon, and Clytemnestra is in turn the child of Leda and Tyndareus.



The general characteristics of trees can be illustrated by these examples. A tree consists of a collection of *nodes* connected by directed *arcs*. A tree has a single *root node*. A node that points to other nodes is termed the *parent* of those nodes while the nodes pointed to are the *children*. Every node except the root has exactly one parent. Nodes with no children are termed *leaf nodes*, while nodes with children are termed *interior nodes*. Identify the root, children of the root, and leaf nodes in the following tree.

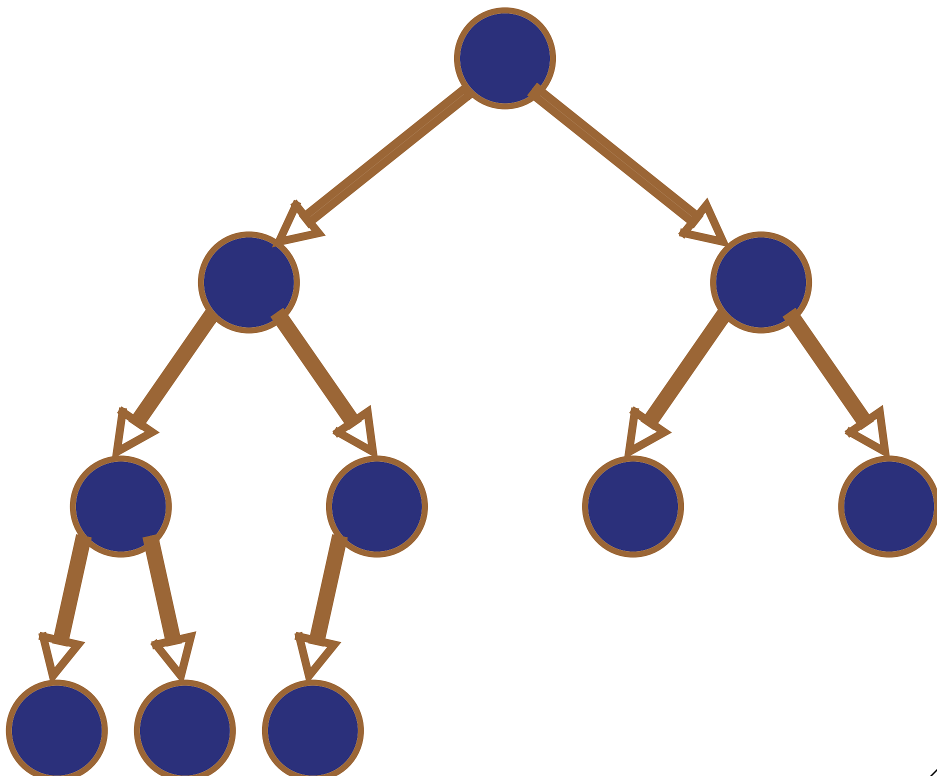


There is a single unique path from the root to any node; that is, arcs don't join together. A path's *length* is equal to the number of arcs traversed. A node's *height* is equal to the maximum path length from that node to a left node. A leaf node has height 0. The height of a tree is the height of the root. A nodes depth is equal to the path length from root to that node. The root has depth 0.

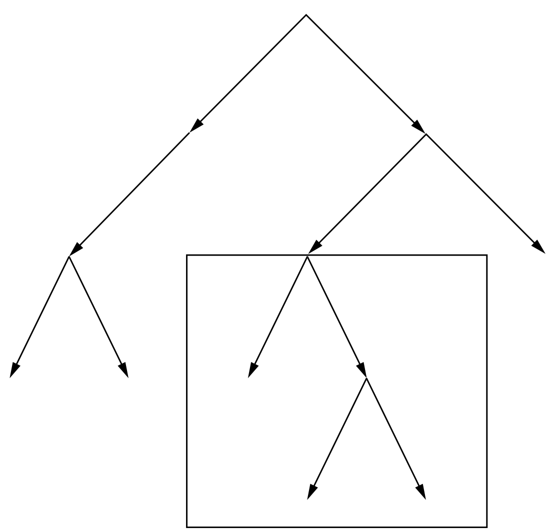
A *binary tree* is a special type of tree. Each node has at most two children. Children are either left or right.

Question: Which of the trees given previously represent a binary tree?

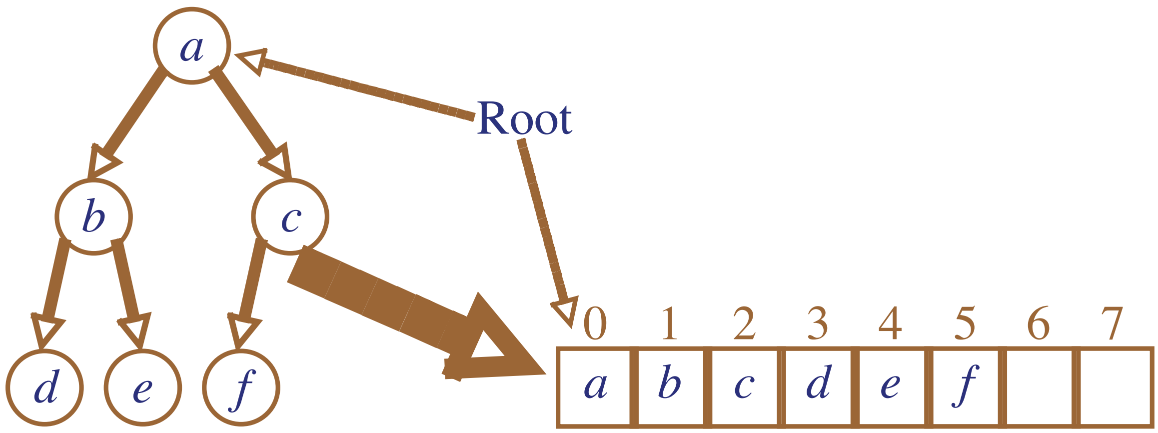
A *full* binary tree is a binary tree in which every leaf is at the same depth. Every internal node has exactly 2 children. Trees with a height of n will have $2^{n+1} – 1$ nodes, and will have 2^n leaves. A *complete* binary tree is a full tree except for the bottom level, which is filled from left to right. How many nodes can there be in a complete binary tree of height n? If we flip this around, what is the height of a complete binary tree with n nodes?



Trees are a good example of a *recursive* data structure. Any node in a tree can be considered to be a root of its own tree, consisting of the values below the node.



A binary tree can be efficiently stored in an array. The root is stored at position 0, and the children of the node stored at position i are stored in positions 2i+1 and 2i+2. The parent of the node stored at position i is found at position floor((i-1)/2).



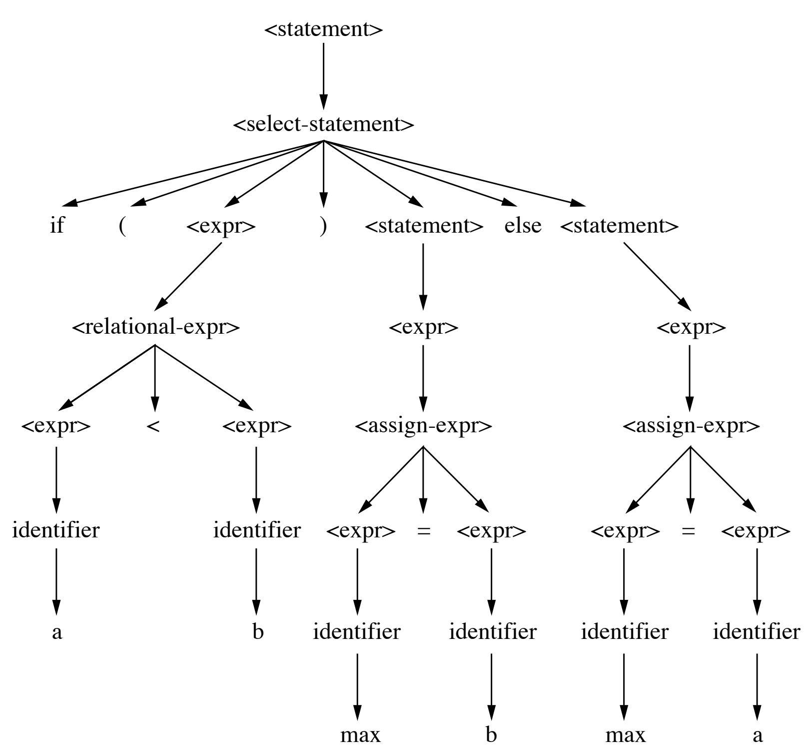
Question: What can you say about a complete binary tree stored in this representation? What will happen if the tree is not complete?

```
struct node {
   EleType value;
   struct node * left;
   struct node * right;
}
```

More commonly we will store our binary trees in instances of class **Node**. This is very similar to the Link class we used in the linked list. Each node will have a value, and a left and right child.

Trees appear in computer science in a surprising large number of varieties and forms. A common example is a parse tree.

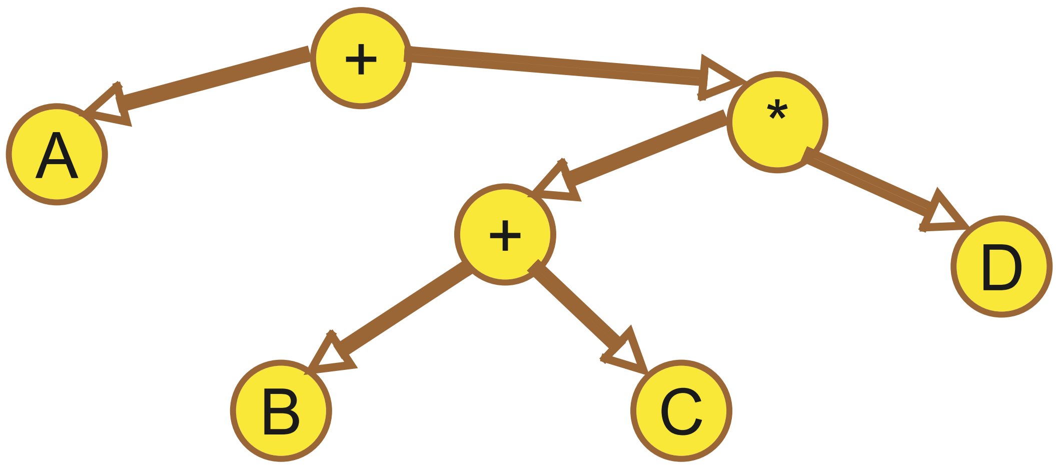
Computer languages, such as C, are defined in part using a grammar. Grammars provide rules that explain how the tokens of a language can be put together. A statement of the language is then constructed using a tree, such as the one shown.



Leaf nodes represent the tokens, or symbols, used in the statement. Interior nodes represent syntactic categories.

Syntax Trees and Polish Notation

A tree is a common way to represent an arithmetic expression. For example, the following tree represents the expression A + (B + C) * D. As you learned in Chapter 5, Polish notation is a way of representing expressions that avoids the need for parentheses by writing the operator for an expression first. The polish notation form for this expression is + A * + B C D. Reverse polish writes the operator after an expression, such as A B C + D * +. Describe the results of each of the following three traversal algorithms on the following tree, and give their relationship to polish notation.

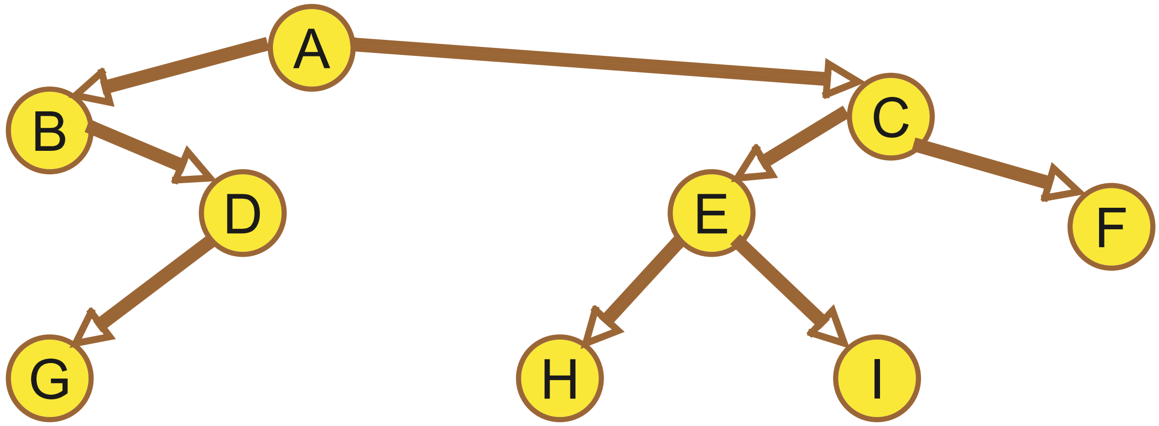


Tree Traversals

Just as with a list, it is often useful to examine every node in a tree in sequence. This is a termed a *traversal*. There are four common traversals:

preorder: Examine a node first, then left children, then right children
Inorder: Examine left children, then a node, then right children
Postorder: Examine left children, then right children, then a node
Levelorder: Examine all nodes of depth n first, then nodes depth n+1, etc

Question: Using the following tree, describe the order that nodes would be visited in each of the traversals.



In practice the inorder traversal is the most useful. As you might have guessed when you simulated the inorder traversal on the above tree, the algorithm makes use of an internal stack. This stack represents the current path that has been traversed from root to left. Notice that the first node visited in an inorder traversal is the leftmost child of the root. A useful function for this purpose is slideLeft. Using the slide left routine, you should verify that the following algorithm will produce a traversal of a binary search tree. The algorithm is given in the form of an iterator, consisting of tree parts: initialization, test for completion, and returning the next element:

```
slideLeft(node n)
  while n is not null
    add n to stack
    n = left child of n
```

```
initialization:
   create an empty stack

has next:
  if stack is empty
    perform slide left on root
  otherwise
    let n be top of stack. Pop topmost element
    slide left on right child of n
  return true if stack is not empty

current element:
  return value of top of stack
```

You should simulate the traversal on the tree given earlier to convince yourself it is visiting nodes in the correct order.

Although the inorder traversal is the most useful in practice, there are other tree traversal algorithms. Simulate each of the following, and verify that they produce the desired traversals. For each algorithm characterize (that is, describe) what values are being held in the stack or queue.

```
PreorderTraversal
 initialize an empty stack
```

```
  has next
     if stack is empty then push the root on to the stack
     otherwise
       pop the topmost element of the stack,
       and push the children from left to right
     return true if stack is not empty

  current element:
     return the value of the current top of stack

PostorderTraversal
  intialize a stack by performing a slideLeft from the root

 has Next
     if top of stack has a right child
       perform slide left from right child
     return true if stack is not empty

  current element
     pop stack and return value of node

LevelorderTraversal
  initialize an empty queue

 has Next
     if queue is empty then push root in to the queue
     otherwise
       pop the front element from the queue
       and push the children from right to left in to the queue
     return true if queue is not empty

  current element
     return the value of the current front of the queue
```

Euler Tours

The three common tree-traversal algorithms can be unified into a single algorithm by an approach that visits every node three times. A node will be visited before its left children (if any) are explored, after all left children have been explored but before any right child, and after all right children have been explored. This traversal of a tree is termed an *Euler tour*. An Euler tour is like a walk around the perimeter of a binary tree.

```
void EulerTour (Node n ) {
   beforeLeft(n);
   if (n.left != null) EulerTour (n.left);
```

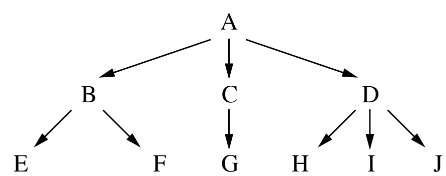
```
    inBetween (n);
    if (n.right != null) EulerTour (n.right);
    afterRight(n);
  }

  void beforeLeft (Node n) { … }
  void inBetween (Node n) { … }
  void afterRight (Node n) { … }
```

The user constructs an Euler tour by providing implementations of the functions beforeLeft, inBetweeen and afterRight. To invoke the tour the root node is passed to the function EulerTour. For example, if a tree represents an arithmetic expression the following could be used to print the representation.

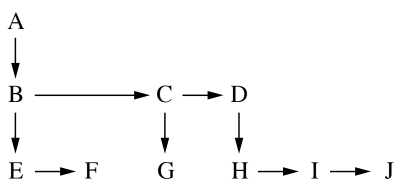
```
  void beforeLeft (Node n) { print("("); }
  void inBetween (Node n) { printl(n.value); }
  void afterRight (Node n) { print(")"); }
```

Binary Tree Representation of General Trees

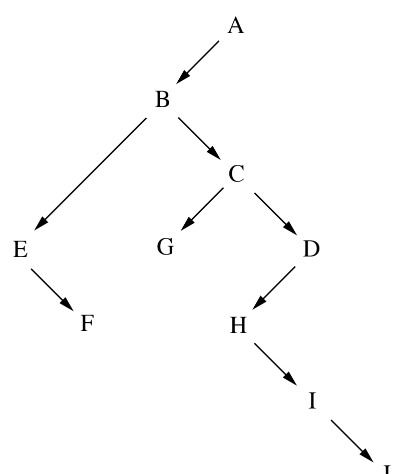
The binary tree is actually more general than you might first imagine. For instance, a binary tree can actually represent any tree structure. To illustrate this, consider an example tree such as the following:



To represent this tree using binary nodes, use the left pointer on each node to indicate the first child of the current node, then use the right pointer to indicate a "sibling", a child with the same parents as the current node. The tree would thus be represented as follows:



Turning the tree 45 degrees makes the representation look more like the binary trees we have been examining in earlier parts of this chapter:



Question: Try each of the tree traversal techniques described earlier on this resulting tree. Which algorithms correspond to a traversal of the original tree?

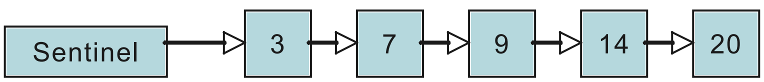
Efficient Logrithmic Implementation Techniques

In the worksheets you will explore two different efficient (that is, logarithmic) implementation techniques. These are the *skip list* and the *balanced binary tree*.

The Skip List

The SkipList is a more complex data structure than we have seen up to this point, and so we will spend more time in development and walking you through the implementation. To motivate the need for the skip list, consider that ordinary linked lists and dynamic arrays have fast (O(1)) addition of new elements, but a slow time for search and removal. A sorted array has a fast O(log n) search time, but is still slow in addition of new elements and in removal. A skip list is fast O(log n) in all three operations.

We begin the development of the skip list by first considering an simple ordered list, with a sentinel on the left, and single links:



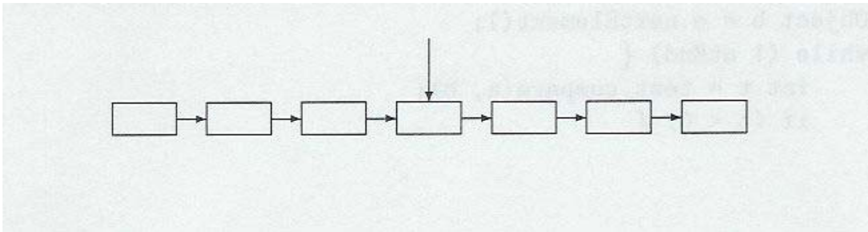
To add a new value to such a list you find the correct location, then set the value. Similarly to see if the list contains an element you find the correct location, and see if it is what you want. And to remove an element, you find the correct location, and remove it. Each of these three has in common the idea of finding the location at which the operation will take place. We can generalize this by writing a common routine, named **slideRight**. This routine will move to the right as long as the next element is smaller than the value being considered.

```
slide right (node n, TYPE test) {
  while (n->next != nil and n->next->value < test)
    n = n->next;
  return n;
}
```

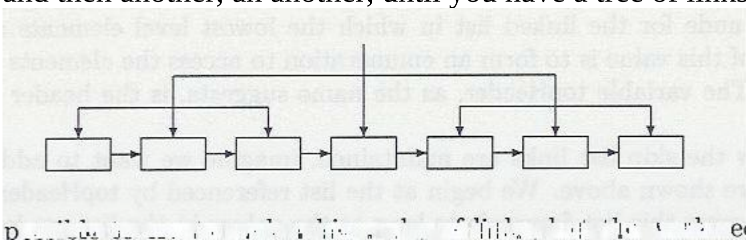
Try simulating some of the operations on this structure using the list shown until you understand what the function slideRight is doing. What happens when you insert the value 10? Search for 14? Insert 25? Remove the value 9?

By itself, this new data structure is not particularly useful or interesting. Each of the three basic operations still loop over the entire collection, and are therefore O(n), which is no better than an ordinary dynamic array or linked list.

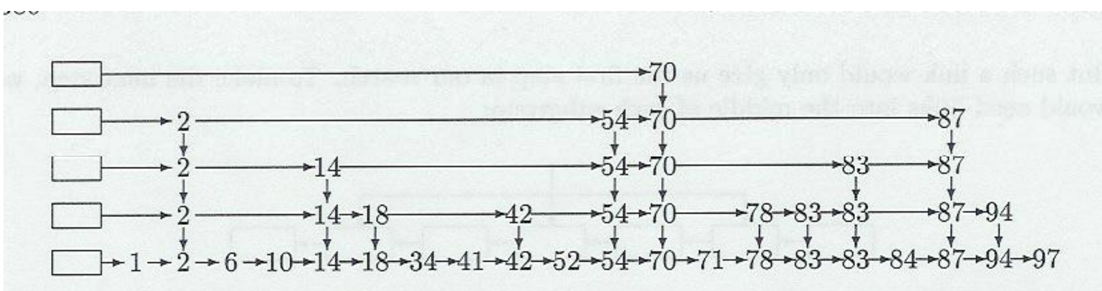
Why can't one do a binary search on a linked list? The answer is because you cannot easily reach the middle of a list. But one could imagine keeping a pointer into the middle of a list:



and then another, an another, until you have a tree of links



In theory this would work, but the effort and cost to maintain the pointers would almost certainly dwarf the gains in search time. But what if we didn't try to be precise, and instead maintained links *probabistically*? We could do this by maintaining a stack of ordered lists, and links that could point down to the lower link. We can arrange this so that each level has approximately half the links of the next lower. There would therefore be approximately log n levels for a list with n elements.



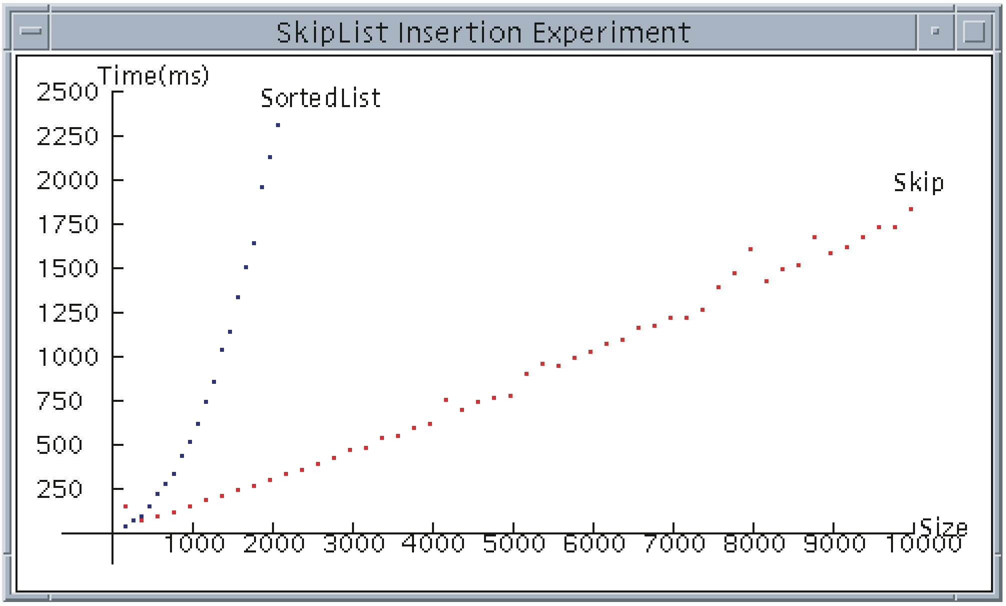
This is the basic idea of the skip list. Because each level has half the elements as the one below, the height is approximately log n. Because operations will end up being proportional to the height of the structure, rather than the number of elements, they will also be O(log n).

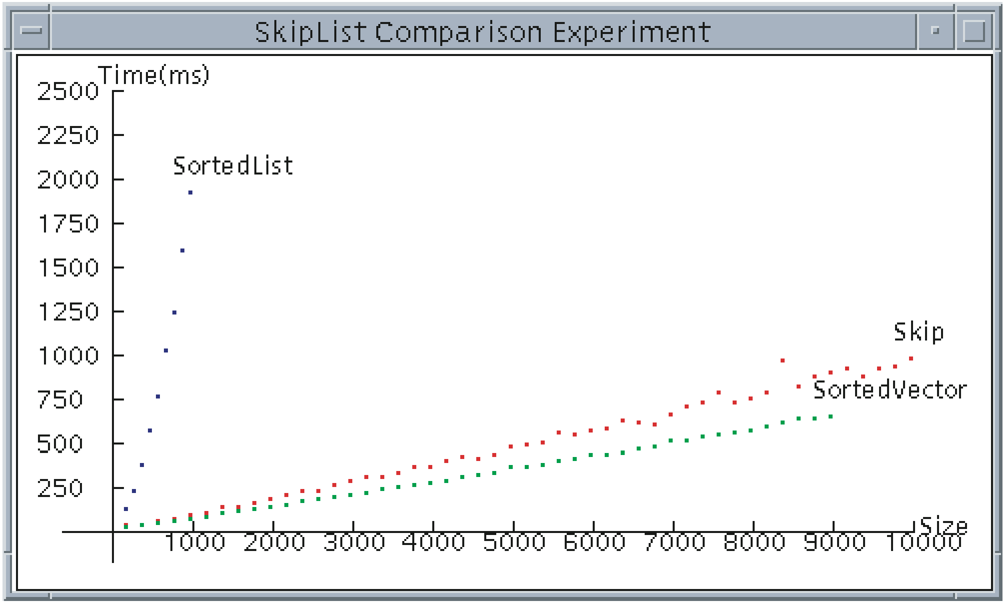
Worksheet 28 will lead through through the implementation of the skip list.

Other data structures have provided fast implementations of one or more operations. An ordinary dynamic array had fast insertion, a sorted array provided a fast search. The skip list is the first data structure we have seen that provides a fast implementation of all three of the Bag operations. This makes it a very good general purpose data structure. The one disadvantage of the skip list is that it uses about twice as much memory as needed by the bottommost linked list. In later lessons we will encounter other data structures that also have fast execution time, and use less memory.

The skip list is the first data structure we have seen that uses *probability*, or *random chance*, as a technique to ensure efficient performance. The use of a coin toss makes the class non-deterministic. If you twice insert the same values into a skip list, you may end up with very different internal links. Random chance used in the right way can be a very powerful tool.

In one experiment to test the execution time of the skip list we compared the insertion time to that of an ordered list. This was done by adding n random integers into both collections. The results were as shown in the first graph below. As expected, insertion into the skip list was much faster than insertion into a sorted list. However, when comparing two operations with similar time the results are more complicated. For example, the second graph compares searching for elements in a sorted array versus in a skip list. Both are O(log n) operations. The sorted array may be slightly faster, however this will in practice be offset by the slower time to perform insertions and removals.



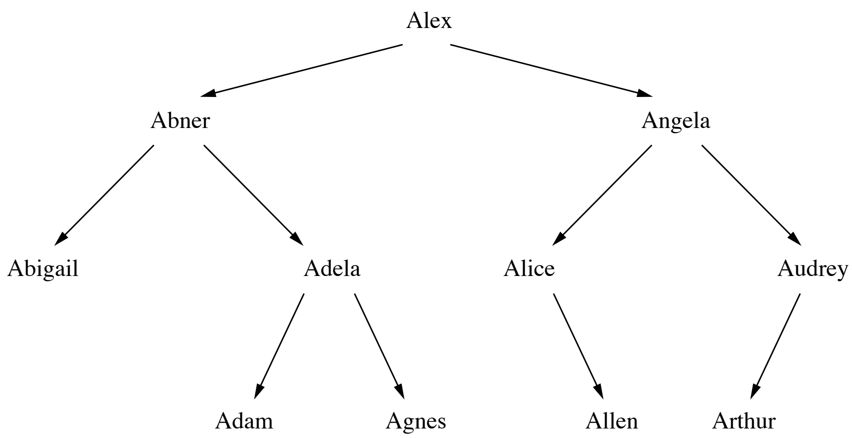


The bottom line says that to select an appropriate container one must consider the entire mix of operations a task will require. If a task requires a mix of operations the skip list is a good overall choice. If insertion is the dominant operation then a simple dynamic array or list might be preferable. If searching is more frequent than insertions then a sorted array is preferable.

The Binary Search Tree

As we noted earlier in this chapter, another approach to finding fast performance is based on the idea of a binary tree. A binary tree consists of a collection of nodes. Each node can have zero, one or two children. No node can be pointed to by more than one other node. The node that points to another node is known as the parent node.

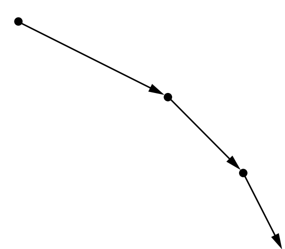
To build a collection out of a tree we construct what is termed a binary search tree. A *binary search tree* is a binary tree that has the following additional property: for each node, the values in all descendants to the left of the node are less than or equal to the value of the node, and the values in all descendants to the right are greater than or equal. The following is an example binary search tree:



Notice that an inorder traversal of a BST will list the elements in sorted order. The most important feature of a binary search tree is that operations can be performed by walking the tree from the top (the root) to the bottom (the leaf). This means that a BST can be used to produce a fast bag implementation. For example, suppose you wish to find out if the name “Agnes” is found in the tree shown. You simply compare the value to the root (Alex). Since Agnes comes before Alex, you travel down the left child. Next you compare “Agnes” to “Abner”. Since it is larger, you travel down the right. Finally you find a node that matches the value you are searching, and so you know it is in the collection. If you find a null pointer along the path, as you would if you were searching for “Sam”, you would know the value was not in the collection.

Adding a value to a binary search tree is easy. You simply perform the same type of traversal as described above, and when you find a null value you insert a new node. Try inserting the value “Amina”. Then try inserting “Sam”.

The development of a bag abstraction based on these ideas occurs in two worksheets. In worksheet 29 you explore the basic algorithms. Unfortunately, bad luck in the order in which values are inserted into the bag can lead to very poor performance. For example, if elements are inserted in order, then the resulting tree is nothing more than a simple linked list. In Worksheet 30 you explore the AVL tree, which rebalances the tree as values are inserted in order to preserve efficient performance.



As long as the tree remains relatively well balanced, the addition of values to a binary search tree is very fast. This can be seen in the execution timings shown below. Here the time required to place n random values into a collection is compared to a Skip List, which had the fastest execution times we have seen so far.

Functional versus State Change Data Structures

In developing the AVL tree, you create a number of functions that operate differently than those we have seen earlier. Rather than making a change to a data structure, such as modifying a child field in an existing node, these functions leave the current value unchanged, and instead create a new value (such as a new subtree) in which the desired modification has been made (for example, in which a new value has been inserted). The methods **add**, **removeLeftmostChild**, and **remove** illustrate this approach to the manipulation of data structures. This technique is often termed the *functional* approach, since it is common in functional programming languages, such as ML and Haskell. In many situations it is easier to describe how to build a new value than it is to describe how to change an existing value. Both approaches have their advantages, and you should add this new way of thinking about a task to your toolbox of techniques and remember it when you are faced with new problems.

Self Study Questions

1. In what operations is a simple dynamic array faster than an ordered array? In what operations is the ordered array faster?

2. What key concept is necessary to achieve logarithmic performance in a data structure?

3. What are the two basic parts of a tree?

4. What is a root note? What is a leaf node? What is an interior node?

5. What is the height of a binary tree?

6. If a tree contains a node that is both a root and a leaf, what can you say about the height of the tree?

7. What are the characteristics of a binary tree?

8. What is a full binary tree? What is a complete binary tree?

9. What are the most common traversals of a binary tree?

10. How are tree traversals related to polish notation?

11. What key insight allows a skip list to achieve efficient performance?

12. What are the features of a binary search tree?

13. Explain the operation of the function **slideRight** in the skip list implementation. What can you say about the link that this method returns?

14. How are the links in a skip list different from the links in previous linked list containers?

15. Explain how the insertion of a new element into a skip list uses random chance.

16. How do you know that the number of levels in a skip list is approximately log n?

Analysis Exercises

1. Would the operations of the skip list be faster or slower if we added a new level one-third of the time, rather than one-half of the time? Would you expect there to be more or fewer levels? What about if the probability were higher, say two-thirds of the time? Design an experiment to discover the effect of these changes.

2. Imagine you implemented the naïve set algorithms described in Lesson 24, but used a skip list rather than a vector. What would the resulting execution times be?

3. Prove that a complete binary tree of height n will have 2^n leaves. (Easy to prove by induction).

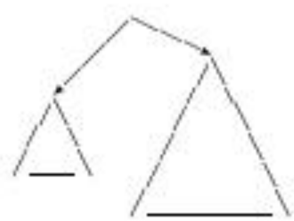
4. Prove that the number of nodes in a complete binary tree of height n is $2^{n+1} – 1$.

5. Prove that a binary tree containing n nodes must have at least one path from root to leaf of length floor(log n).

6. Prove that in a complete binary tree containing n nodes, the longest path from root to leaf traverses no more than ceil(log n) nodes.

7. So how close to being well balanced is an AVL tree? Recall that the definition asserts the difference in height between any two children is no more than one. This property is termed a *height-balanced tree*. Height balance assures that locally, at each node, the balance is roughly maintained, although globally over the entire tree differences in path lengths can be somewhat larger. The following shows an example height-balanced binary tree.

A complete binary tree is also height balanced. Thus, the largest number of nodes in a balanced binary tree of height h is $2^{h+1}-1$. An interesting question is to discover the *smallest* number of nodes in a height-balanced binary tree. For height zero there is only one tree. For height 1 there are three trees, the smallest of which has two nodes. In general, for a tree of height h the smallest number of nodes is found by connecting the smallest tree of height h-1 and h-2.



If we let M_h represent the function yielding the minimum number of nodes for a height balanced tree of height h, we obtain the following equations:

$M_0 = 1$
$M_1 = 2$
$M_{h+1} = M_{h-1} + M_h + 1$

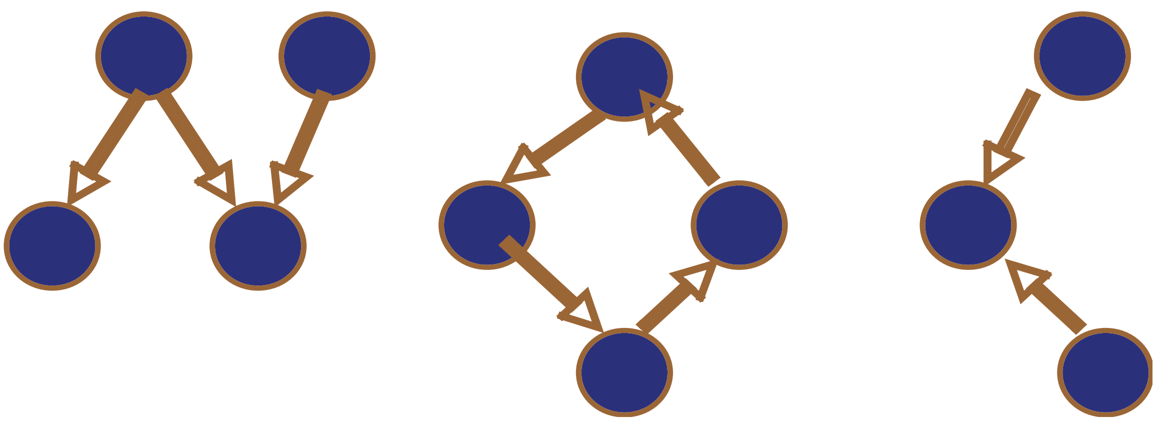
These equations are very similar to the famous *Fibonacci numbers* defined by the formula $f_0 = 0$, $f_1 = 1$, $f_{n+1} = f_{n-1} + f_n$. An induction argument can be used to show that $M_h = f_{h+3} - 1$. It is easy to show using induction that we can bound the Fibonacci numbers by 2^n. In fact, it is possible to establish an even tighter bounding value. Although the details need not concern us here, the Fibonacci numbers have a closed form solution; that is, a solution defined without using recursion. The value F_h is approximately $\frac{\Phi^i}{\sqrt{5}}$, where Φ is the golden mean value $(1+\sqrt{5})/2$, or approximately 1.618. Using this information, we can show that the function M_h also has an approximate closed form solution:

$$M_h \approx \frac{\Phi^{h+3}}{\sqrt{5}} - 1$$

By taking the logarithm of both sides and discarding all but the most significant terms we obtain the result that h is approximately 1.44 log M_h. This tells us that the longest path in a height-balanced binary tree with n nodes is at worst only 44 percent larger than the log n minimum length. Hence algorithms on height-balanced binary trees that run in time proportional to the length of the path are still O(log n). More importantly, preserving the height balanced property is considerably easier than maintaining a completely balanced tree.

Because AVL trees are fast on all three bag operations they are a good general purpose data structure useful in many different types of applications. The Java standard library has a collection type, TreeSet, that is based on a data type very similar to the AVL trees described here.

Explain why the following are not legal trees.



Programming Projects

1. The bottom row of a skip list is a simple ordered list. You already know from Lesson 36 how to make an iterator for ordered lists. Using this technique, create an iterator for the skip list abstraction. How do you handle the remove operation?

2. The smallest value in a skip list will always be the first element. Assuming you have implemented the iterator described in question 6, you have a way of accessing this value. Show how to use a skip list to implement a priority queue. A priority queue, you will recall, provides fast access to the smallest element in a collection. What will be the algorithmic execution time for each of the priority queue operations?

```
void skipPQaddElement (struct skipList *, TYPE newElement);
TYPE skipPQsmallestElement (struct skipList *);
void skipPQremoveSmallest (struct skipList *);
```

On the Web

The wikipedia has a good explanation of various efficient data structures. These include entries on skip lists, AVL trees and other self-balancing binary search trees. Two forms of tree deserve special note. Red/Black trees are more complex than AVL trees, but require only one bit of additional information (whether a node is red or black), and are in practice slightly faster than AVL trees. For this reason they are used in many data structure libraries. B-trees (such as 2-3 B trees) store a larger number of values in a block, similar to the dynamic array block. They are frequently used when the actual data is stored externally, rather than in memory. When a value is accessed the entire block is moved back into memory.