**Guohua Jiang**

**Homework 2**

**3.7** What is the running time of the following code?

public static List<Integer> makeList( int N )

{

ArrayList<Integer> lst = new ArrayList<>( );

for( int i = 0; i < N; i++ )

{

lst.add( i );

lst.trimToSize( );

}

}

First, we have to understand what trimToSize method does. The trimToSize method basically trims the capacity of the arraylist to the current size. It copy the entire arrayList to a new array; therefore the runtime of trimToSize method is O(N).

There are two operations inside the for loop. Each iteration, add method tales O(1), and trimToSize method is O(N) takes O(N). The for loop runs N times. Therefore the running time for this algorithm is O(N2).

**3.8** The following routine removes the first half of the list passed as a parameter: public static void removeFirstHalf( List<?> lst )

{

int theSize = lst.size( ) / 2;

for( int i = 0; i < theSize; i++ )

lst.remove( 0 );

}

1. Why is theSize saved prior to entering the for loop?

By dividing half of the list, we can eliminate a half of the list size before stepping in the for loop.

1. What is the running time of removeFirstHalf if lst is an ArrayList?

Each iteration, the size of the list will be divided by half; therefore the loop will takes O(logN) running time. If the lst is an ArrayList, after removing the first element, all the remaining elements have to shift 1 position to the front; therefore, each remove operation takes O(N) running time.

Thus, the overall running time for this algorithm is O(NlogN).

1. What is the running time of removeFirstHalf if lst is a LinkedList?

For the LinkedList, eletion in the fitst element takes constant time.

Thus the running time would be O(logN).

1. Does using an iterator make removeHalf faster for either type of List?

In this case, iterator would not make faster for ArrayList or LinkedList. We are always removing the first element in the list; therefore we can find the element instantly.

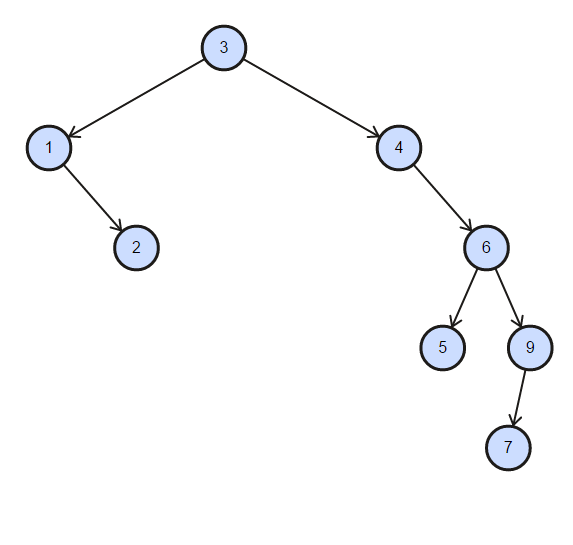
**4.4** Show that in a binary tree of *N* nodes, there are *N* + 1 null links representing children.

For a N nodes binary tree, there are will be 2N links in total. Beside the root, each node will has one link comes from its parent, which means N – 1 links are not null.

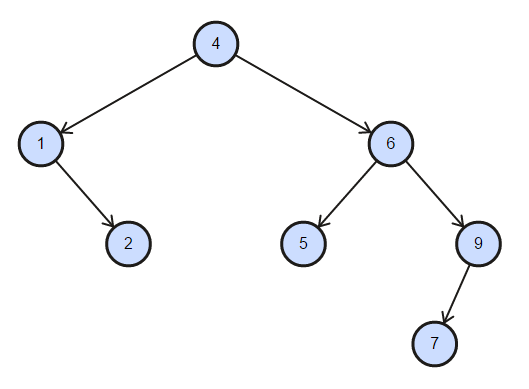
2N – (N - 1) = N + 1

Thus, there are N + 1 null links in a binary tree of N nodes.

**4.9** a. Show the result of inserting 3, 1, 4, 6, 9, 2, 5, 7 into an initially empty binary search tree.



b. Show the result of deleting the root.



**4.18** a. Give a precise expression for the minimum number of nodes in an AVL tree of height *h*.

*S*(*h*) = *S*(*h* − 1)+ *S*(*h* − 2) + 1

For *h* = 0, *S*(*h*) = 1

and *h* = 1, *S*(*h*) = 2

b. What is the minimum number of nodes in an AVL tree of height 15?

S(1) = 1

S(2) = 2

S(3) = S(2) + S(1) + 1 = 4

S(4) = S(3) + S(2) + 1 = 7

S(5) = S(4) + S(3) + 1 = 12

S(6) = S(5) + S(4) + 1 = 20

S(7) = S(6) + S(5) + 1 = 33

S(8) = S(7) + S(6) + 1 = 54

S(9) = S(8) + S(7) + 1 = 88

S(10) = S(9) + S(8) + 1 = 143

S(11) = S(10) + S(9) + 1 = 232

S(12) = S(11) + S(10) + 1 = 376

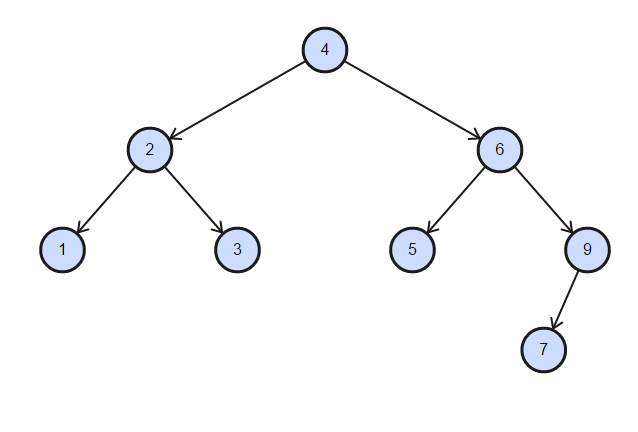
S(13) = S(13) + S(12) + 1 = 609

S(14) = S(13) + S(12) + 1 = 986

S(15) = S(14) + S(13) + 1 = 1596

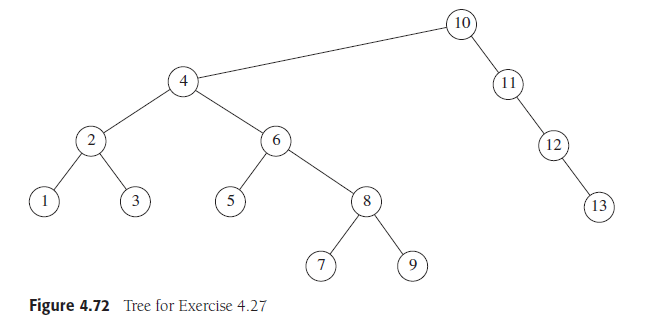
There for the minimum number of nodes in an AVL tree of height 15 is 1596.

**4.19** Show the result of inserting 2, 1, 4, 5, 9, 3, 6, 7 into an initially empty AVL tree.

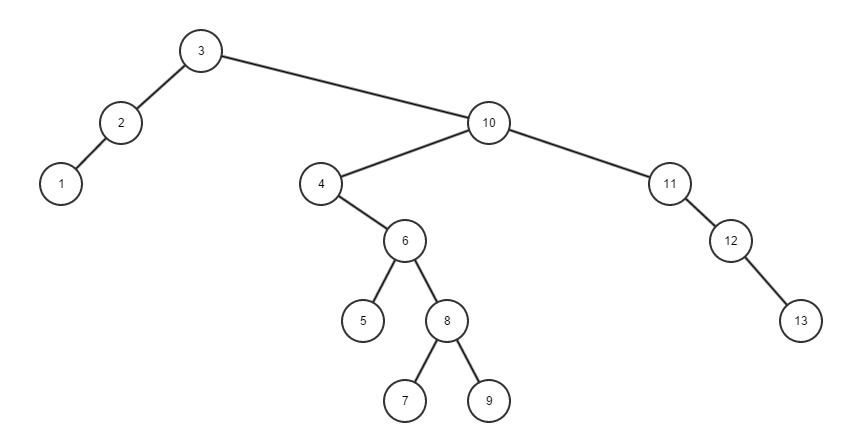


**4.27** Show the result of accessing the keys 3, 9, 1, 5 in order in the splay tree in

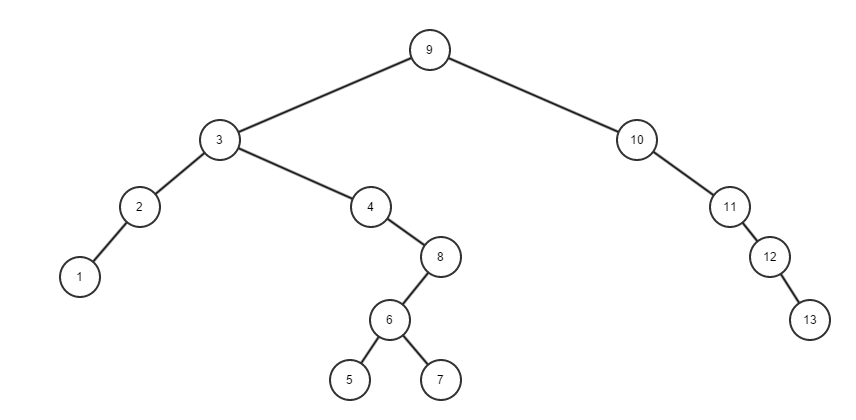
Figure 4.72.



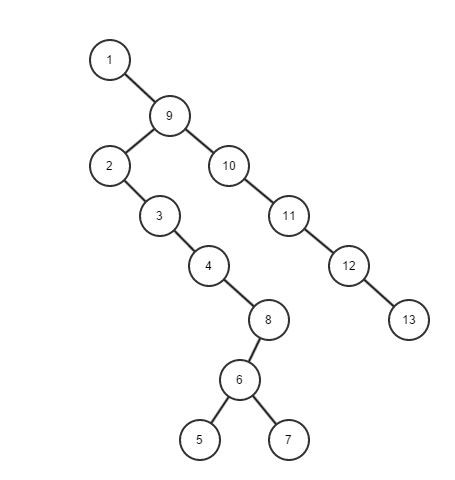
After accessing the key 3:



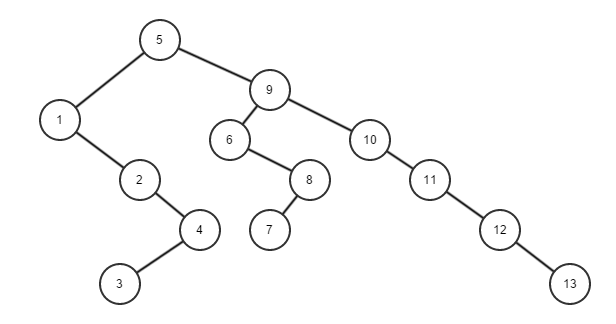
After accessing the key 9:



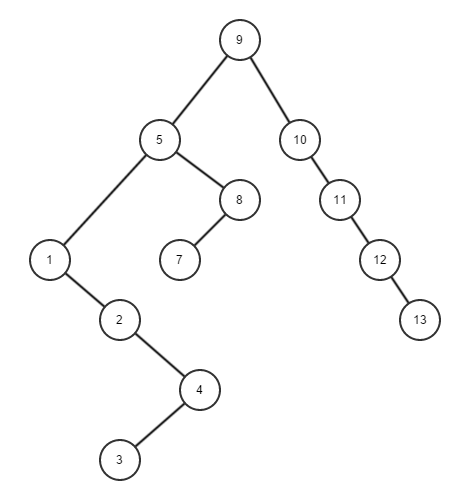
After accessing the key 1:



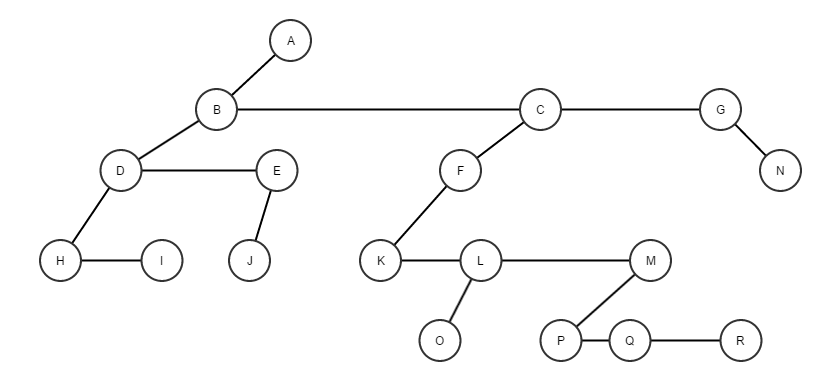
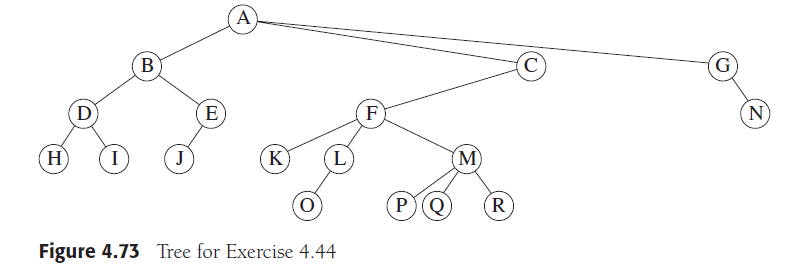
After accessing the key 5:



**4.28** Show the result of deleting the element with key 6 in the resulting splay tree for the previous exercise.



**4.44** Show how the tree in Figure 4.73 is represented using a child/sibling link implementation.



**4.51** Let *f* (*N*) be the average number of full nodes in a binary search tree.

a. Determine the values of *f*(0) and *f* (1).

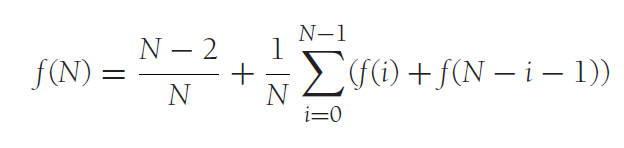
When N = 0, the tree is an empty tree.

Thus *f* (0) = 0.

When N = 1, the tree has only 1 node which happens to be to the root of the tree with on child; therefore it is not a full node.

Thus *f* (1) = 0.

b. Show that for *N >* 1



Base case: One full node require at least 3 nodes; therefore when N = 2, f(2) must be 0. Substituting 2 in the function, we have f(2) = 0.

Thus, the formula works for the N =2.

Assume: the formula works for some K, where K ≤ N.

Prove: the formula works for K+1.