Problem Set 4

Problem 1 Consider a tree storing 100,000 entries. What is the worst-case height of T in the following cases?

a. T is an AVL tree.
b. T is a (2,4) tree.
c. T is a binary search tree.
d. T is a splay tree.

Solution:

a. Let $n(h)$ be the minimum number of nodes of an AVL tree with height h. We have:
$n(h)=n(h-1)+n(h-2)+1$ $(h\geq2)$, where $n(0)=1$ and $n(1)=2$.

The first 6 $n(h)$'s ($h=0, 1, \ldots, 5$) are: 1, 2, 4, 7, 12, 20.
Recall that the first 9 Fibonacci numbers F_i ($i=0, 1, \ldots 8$) are: 0, 1, 1, 2, 3, 5, 8, 13, 21.

Notice that $n(h) = F_{h+3} - 1$, where F_{h+3} is a Fibonacci number with $F_{h+3} = F_{h+2} + F_{h+1}$.
It is known that $F_h = [\varphi^h/5^{1/2}]$, where $\varphi=(1+5^{1/2})/2$ is the golden ratio.
Therefore, we have $n(h) = [\varphi^{h+3}/5^{1/2}] - 1$, and $h = \log(5^{1/2}(n(h)+1))/\log \varphi - 3$.
By applying the above formula, we get $h = 22$ for an AVL tree storing 100,000 entries.

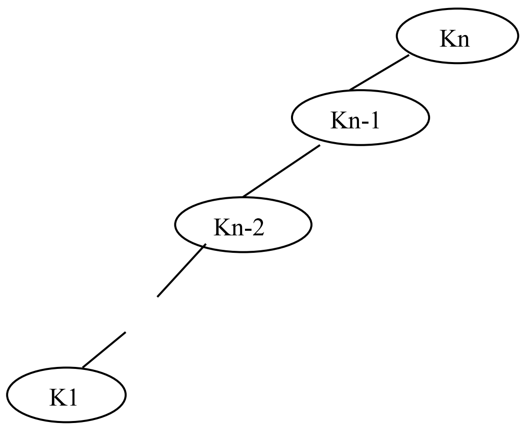
b. In the worst-case height, a (2, 3) tree is a complete binary search tree, where there are 2^i nodes at depth i. Let n be the number of nodes, the height h satisfies the following constraint:
$n=2^0 + 2^1 +2^2+ \ldots + 2^h$ (1)
Multiplying both sides by 2, we have
$2n=2^1 + 2^2 +2^3+ \ldots + 2^h + 2^{h+1}$ (2)
So, we have $n=2^{h+1} -1$. Therefore, $h=\log (n+1)-1$.
Hence, the height of a (2,4) tree storing n entries is at most floor(log(n+1)-1), where floor(x) is the largest integer that is no larger than x. Therefore, the worst-case height of T is floor(log(100,001)-1) =15.

c. The worst-case height of T is 99,999.

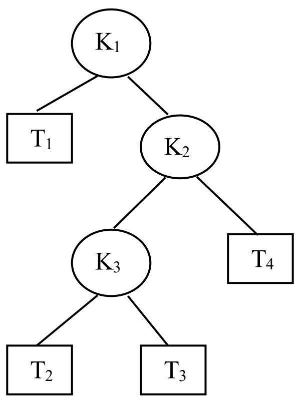
d. The worst-case height of T is 99,999.

Problem 2 What does a splay tree look like if its entries are accessed in increasing order of their keys?

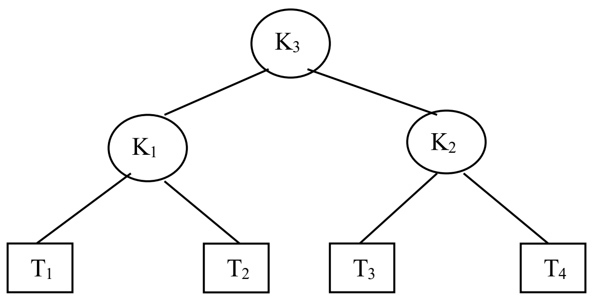
Solution: Assume that there are n entries in the splay tree and $k_i \leq k_{i+1}$ ($i=1, 2, n-1$), where k_i is the key of entry i. After entry 1 is assessed, it is moved to the root. Since k_1 is the smallest key, all other entries are in the right subtree of entry 1. Similarly, after entry 2 is accessed, entry 2 is moved to the root and entry 1 becomes the left child of entry 2. After entry n is accessed, entry n is moved to the root and entry i-1 is the left child of entry i (i=2, 3, …, n). The resulting splay tree is shown as follows.



Problem 3 Assuming that *T*1, *T*2, *T*3, and *T*4 are AVL trees of height *h*, alter the following binary search tree to yield an AVL tree.



Solution:



Problem 4 Design an algorithm findAll(k) for finding all the entries with the key k in a binary search tree T, and show that it runs in time O(h+s), where h is the height of T and s is the size of the collection returned.

Solution: In a binary search tree, we need to make a rule for placing entries with duplicate keys. Entries with duplicate keys can be inserted in the left subtree or the right subtree of a node with the same key. Therefore, when we find the first node with the same key, we can either search left subtree or the right subtree for all the entries with the same key. However, in case of AVL trees, tri-node restructuring may spoil this order, that is, entries with duplicate keys may be stored in both the left subtree and the right subtree of the node with the same key.

The key idea of the algorithm for finding all the entries with the key k is that after we find the first node, we search the subtree rooted at the first node that contains all the entries with the key k. The algorithm is shown as follows:

```
Algorithm findAll(k)
      Input: The search key k, the root v of the binary search tree T
      Output: A list L containing all the entries with the key k
      {
        Create an empty list L;
        while ( v != NULL )
           {
            if (v.key=k)
               {
                findAllEntries(v, L);
                return L;
               }
            else
              if ( v.key <k)
                 v=v.left;
              else
                 v=v.right;
           }
          return L;
      }

Algorithm findAllEntries(v, L)
      Input: A node v with the key k and a List L
      Output: All the entries with the key k in the subtree rooted at v
      {
         // We use recursive pre-order traversal to find all the entries with the key k
in the subtree rooted at v
         if (v.key=k)
            {
             Add v to L;
             if ( v.left != NULL)
                FindAllEntries(v.left, L)
             if (v.right !=NULL)
                FindAllEntries(v.right, L);
            }
        return; // Traversal returns when v does not contain the key k
      }
```

Time complexity analysis:

1. Finding the first node with the key k takes O(h) time.
2. Since at 2s nodes are visited by findAllEntries(), traversing the subtree rooted at the first node takes O(s) time.

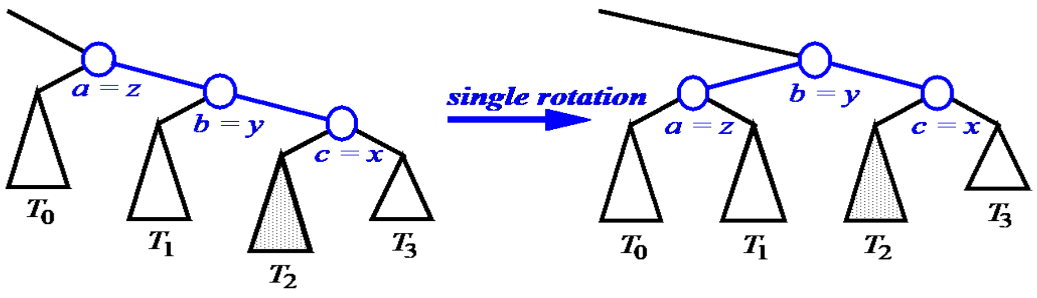
Therefore, this algorithm takes O(h+s) time.

Problem 5 Show that after the tri-node restructuring operation on the subtree rooted at the node z, the new subtree is an avl tree.

Proof: Tri-node restructuring occurs only when a new node is inserted into an avl tree or when a done is removed from an avl tree. Next, we consider the cases for inserting a node. The proof for deleting a node is similar.

Let height (v) be the height of a subtree v or the height of the subtree rooted at v. There are four cases:

Case 1:



By the definition of x, y, z, before tri-node restructuring we have the following equations:

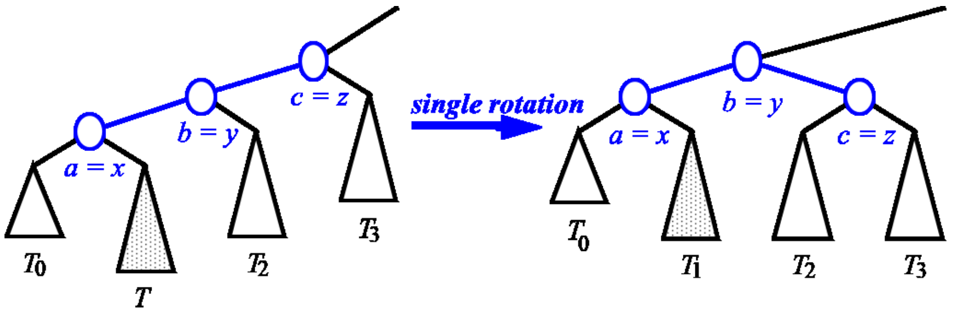
height(y)=height(T_0)+2 (1)
height(x)≤height(T1)+1 (2)
|height(T3)-height(T2)|≤1. (3)

From the above equations, we have:

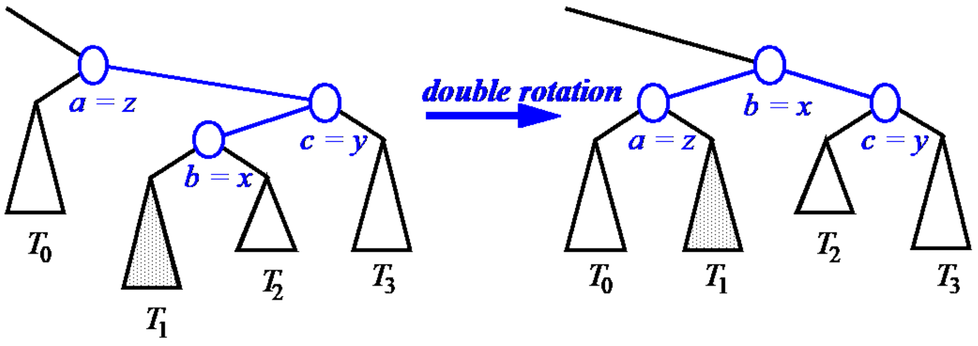
|height(T0)-height(T1)|≤1 (4)
|max{height(T0), height(T1)}-max{height(T2), height(T3)}|≤1 (5)

Equations (3) and (4) imply that in the new tree both z and x satisfy the height difference constraint. Equation (5) implies that in the new subtree, the root y satisfies the height difference constraint. Therefore, the new subtree rooted at y is an avl tree.

Case 2: This case is symmetrical to Case 1.



Case 3:



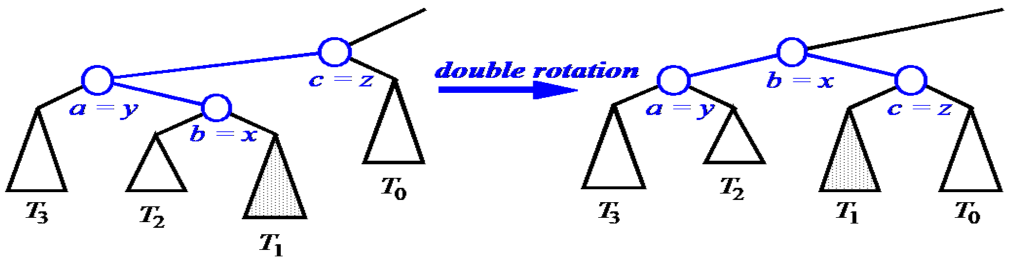
By the definition of x, y, z, before tri-node restructuring we have the following equations:

height(y)=height(T0)+2 (1)
height(x)⩽height(T3)+1 (2)
|height(T1)-height(T2)|⩽1. (3)

From the above equations, we have:

|height(T0)-height(T1)|⩽1 (4)
|max{height(T0), height(T1)}-max{height(T2), height(T3)}|⩽1 (5)

Case 4: This case is symmetrical to Case 3.



Problem 6 Let T and U be (2,4) trees with n and m entries, respectively, such that all the entries in T have keys less than the keys of all the entries in U. Describe an O(log n + log m)-time algorithm for merging T and U into a single (2,4) tree.

Solution: Let h_t and h_u be the heights of T and U, respectively. Consider two possible cases:

Case 1. $h_t \geq h_u$. Do the following:

a. Find the entry e with the smallest key in U.
b. Remove the entry e from U.
c. Let h'_u be the new height of U.
d. If $h_t > h'_u$, do the following:
   i. Insert the entry e into the rightmost node v of T at height $h'_u + 1$, and make the root of U the rightmost child of v.
   ii. If there is an overflow, handle the overflow.
e. If $h_t = h'_u$, do the following:
   i. Create a new node v, and insert the entry e into v.
   ii. Make U the left child of v and T the right child of v.

Case 2. $h_t < h_u$. This case is symmetrical to Case 1. Do the following:

f. Find the entry e with the largest key in T.
g. Remove the entry e from T.
h. Let h'_t be the new height of T
i. Insert the entry e into the leftmost node u of U at height $h'_t + 1$, and make the root of T the leftmost child of u.
j. If there is an overflow, handle the overflow.

Time complexity analysis:
Case 1:

a. Finding the entry e with the smallest key in U takes O(log m) time.
b. The remove operation on U takes O(log m) time.
c. The insert operation on T, including handling overflows, takes O(log n) time.

Therefore, the time complexity for this case is O(log n + log m).

Case 1:

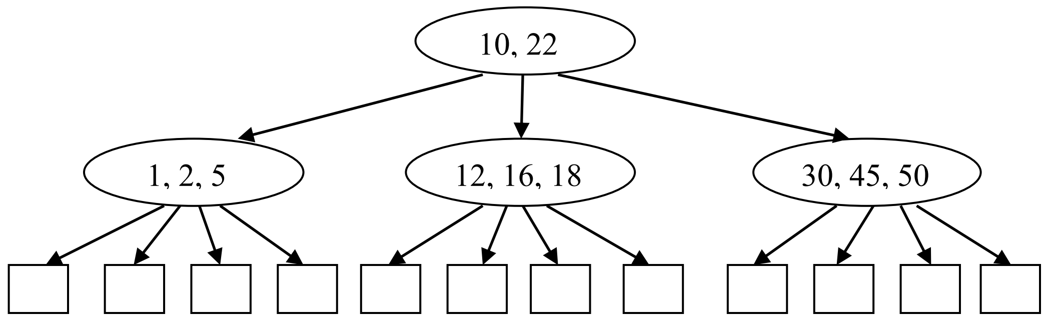
a. Finding the entry e with the largest key in T takes O(log n) time.
d. The remove operation on T takes O(log n) time.
e. The insert operation on U, including handling overflows, takes O(log m) time.

Therefore, the time complexity for this case is O(log n + log m).
Based on the above time complexity analysis, we can conclude that this algorithm takes O(log n + log m) time.

Problem 7 Consider a sequence of keys (5, 16, 22, 45, 2, 10, 18, 30, 50, 12, 1). Draw the final result of inserting the entries with these keys in the given order into

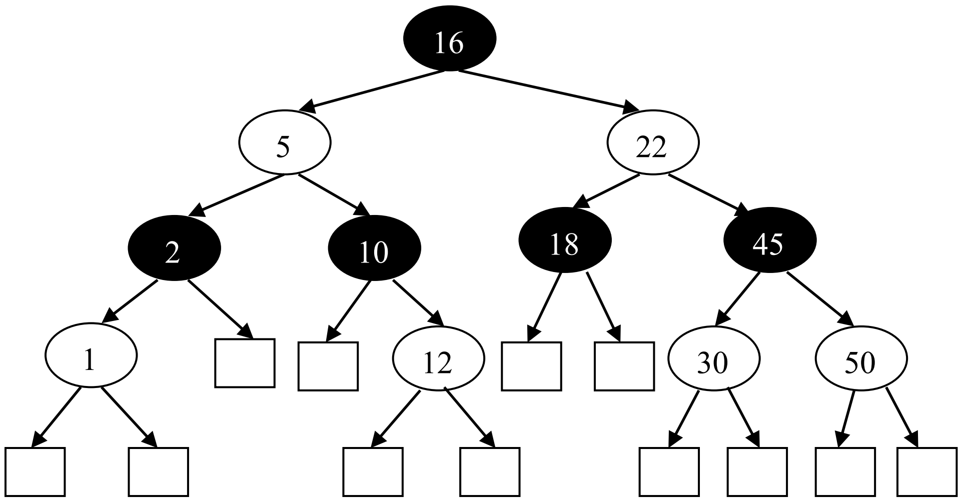
a. An initially empty (2, 4) tree.
b. An initially empty red-black tree.

Solution: a. The final (2, 4) tree:



Notice that the answer is not unique. When an overflow occurs, we can push either the second entry or the third entry to the parent, resulting in different trees.

b. The final red-black tree:



Problem 8 Consider a binary search tree where all the keys are distinct. Describe a modification to the binary search tree that supports the following two index-based operations in O(h) time, where h is the height of the tree:

- atIndex(i): return the entry in the binary search tree with index i.
- indexOf(e): return the index of entry e.

The index of an entry is the sequence number of the entry in the inorder traversal on the tree. For example, the indices of the first two entries with the smallest key and the second smallest key are 0 and 1, respectively.

Solution: For each node, we introduce a field named *size*, denoting the number of entries stored in the subtree rooted at this node. The field *size* of each relevant internal node needs to be updated when a new entry is inserted or an entry is removed.

- Key idea with atIndex(i): We use a recursive algorithm findIndex(i,v), where i is the relative index in the subtree rooted at v, and v is the current node. Given the current node v, if the target entry is in the left subtree, the relative index is unchanged. If the target entry is in the right subtree, the relative index in the right subtree is i-left(v).size-1.

- Key idea with indexOf(e): Since the index of an entry is its sequence number in the inorder traversal on the tree, the algorithms for indexOf(e) dynamically keeps track of the running number of inorder predecessors of the target entry.

```
Algorithm atIndex(i)
    Input: The index i
    Output: The position in the binary search tree of the entry at index i.
        {
          if ( i>= the total number of entries in the tree )
              return IndexOutOfBound(); // error
          return findIndex(i, root);
        }

Algorithm findIndex(i, v)
    Input: The index i and the current node v
    Output: The position in the binary search tree of the entry at index i.
        {
          if ( (i=0 and left(v)=null) or left(v).size=i)
              return v;   // found
          if (i<left(v).size)   // in the left subtree and the relative index in the left
                                 // subtree remains the same
              return findIndex(i, left(v));
          else   // in the right subtree and its relative index in the right subtree is
                 //  i-left(v).size-1
              return findIndex(i-left(v).size-1, right(v));
        }

Algorithm indexOf(e)
    Input: An entry e
    Output: The index of e.
        {
          if ( root=null )
              return EntryDoesNotExist();  //  e is not found
          return indexOf(e, root, 0);
        }

Algorithm indexOf(e, v, i)
    Input: An entry e, the current node v, and the running number of inorder
           predecessors of v
    Output: The index of e.
        {
          if (v is an external node)
              return EntryDoesNotExist();  // e is not found
          if (v.key =  the key of e)
              return i+left(v).size;   //  e is found
          if (v.key<the key of e)
             // e is in the left subtree, so the running number of inorder predecessors of
             // e is unchanged
              return indexOf(e, left(v), i);
          else   // e is in the right subtree and the running number of inorder
```

```
            // predecessors of e increases by left(v).size+1
        **if**  (left(v)≠null)
            **return** indexOf(e, right(v), i+left(v).size+1);
        **else**
            **return** indexOf(e, right(v), i+1);
}
```

Time complexity analysis: Both algorithms traverse the tree along one path from the root to a leaf node in the worst case, and it takes O(1) time to visit each node. Therefore, both algorithms take O(h) time, where h is the height of the tree.