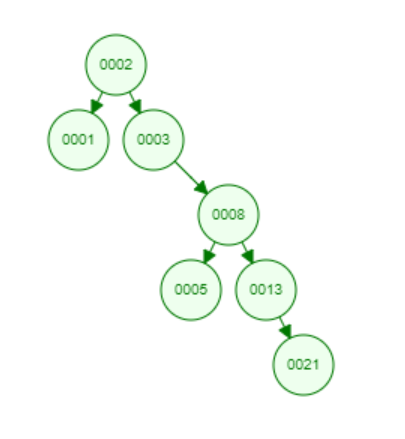
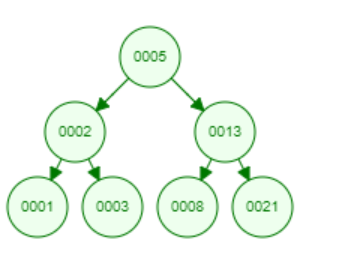
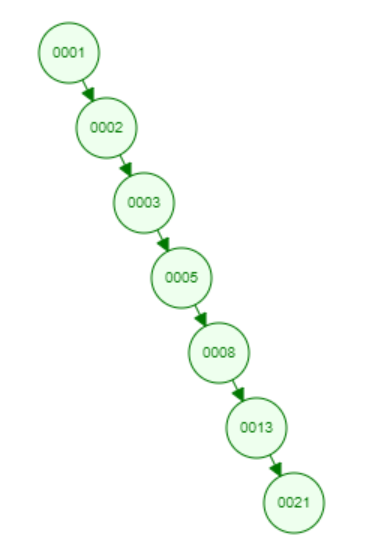
1. 
2. A. 2, 252, 401, 398, 330, 344, 397, 363. This sequence can be used.

b. 924, 220, 911, 244, 898, 258, 362, 363. This sequence can be used.

c. 925, 202, 911, 240, 912, 245, 363. This sequence cannot be used because we will never get to node 912 in the search because it is greater than node 911.

d. 2, 399, 387, 219, 266, 382, 381, 278, 363. This sequence can be used.

e. 935, 278, 347, 621, 299, 392, 358, 363. This sequence cannot be used because we will never get to node 299 because it is less than node 347.

1. For a binary search tree T of n nodes, determine the number of internal nodes in O(n) running time in psuedo code.

Give a recursive algorithm.

countInternalNodes(node\* root){

count=0

if(root->left != NULL || root->right != NULL){

count =1

if(root->left != NULL){

count+= countInternalNodes(root->left)

}

If(root->right != NULL){

Count+= countInternalNodes(root->right)

}

}

Return count

}

(2) Give a non-recursive algorithm.

ountInternalNodes(node\* root){

count=0

current=root

while(root->left != NULL || root->right != NULL){

count++

if(current->left == NULL){

current=current->right

}

Else{

Pre=current->left

 while(pre->right != NULL && pre->right != current){

         pre = pre->right;

if(pre->right == NULL){

pre->right=current

current=current->left

}

Else{

Pre->right=NULL

Current=current->right

}

}

}

}

Return count

}

1. Given a Binary Search Tree T and two integers a and b on the tree, print all elements in T between a and b in psuedo code.

printNodes(node\* T, int a, int b){

if(T==NULL){

return

}

If(a < T->data){

printNodes(T->left, a, b)

}

If(a <= T->data && b >= T->data){

Print(T->data)

}

If(b > T->data){

printNodes(T->right, a, b)

}

}

1. Suppose that instead of each node x keeping the attribute x:p, pointing to x’s

parent, it keeps x:succ, pointing to x’s successor. Give pseudocode for SEARCH,

INSERT, and DELETE on a binary search tree T using this representation. These

procedures should operate in time O(h), where h is the height of the tree T . (Hint:

You may wish to implement a subroutine that returns the parent of a node. This is the same question of 12.3-5 in the textbook.)

findParent(x){

if(x==root)

return NULL

max=TREE\_MAXIMUM(x)

parent=max->succ,

if(parent)

t=parent->left

else

t=root

while(t !+ x){

parent=t

t=t->right

}

Return parent

}

Search(node\* root, int key)

{

if (root == NULL || root->key == key)

return root

if (root->key < key)

return search(root->right, key);

return search(root->left, key);

}

Insert(node\* root, int key){

if (node == NULL)

return newNode(key)

if (key < node->key)

node->left = insert(node->left, key)

else if (key > node->key)

node->right = insert(node->right, key);

return node;

}

DeleteNode(node\* root, int key){

if (root == NULL)

return root;

if (key < root->key)

root->left = deleteNode(root->left, key);

else if (key > root->key)

root->right = deleteNode(root->right, key);

else

{

if (root->left == NULL)

{

node \*temp = root->right;

free(root);

return temp;

}

else if (root->right == NULL)

{

node \*temp = root->left;

free(root);

return temp;

}

root->key = root->succ->key;

root->right = deleteNode(root->right, temp->key);

}

return root;

}