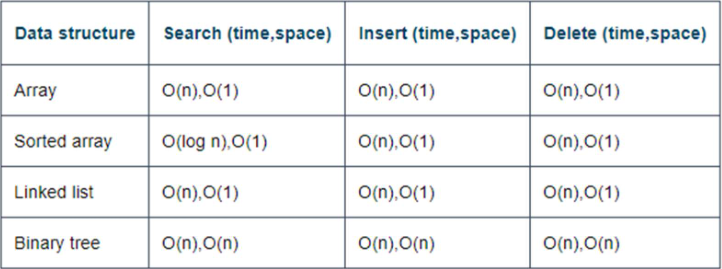
Lecture 21 Binary Search Trees

Design efficient data structures for search

* Insert and delete are fundamental operations to build and maintain data structures. Search is the common operation to use a data structure.



**Concept of Binary search trees**

* A BIST is a binary tree with the following properties

1. A field or a node property is used as a key for search comparison, i.e., a tree node is associated with a key value
2. For any node of the tree, the key values of all nodes in the left sub tree are less than the key value of the node, and the key values of all nodes in the right sub tree are equal or greater than the key value of the node

Note: for simplicity, we restrict that keys are distinct

**Example of node structure**

Typedef struct node{

Int data; //this variable can be used as a key

Struct node \*left;

Struct node \*right;

}TNODE;

Typedef struct node {

Char \*name[20];//key var

Float score;

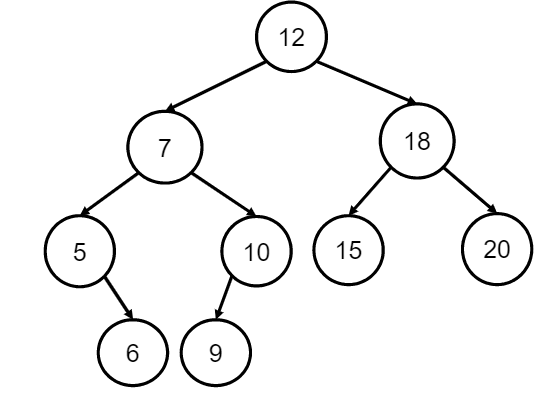
Struct node \*left;

Struct node \*right;

}RECORD;

**Sorted ordering of nodes of BST**

* The in-order traversal of BST gives the increasing ordering of nodes sorted by the key values, such BST is also called ordered binary tree.
* Ex. In-order traversal: 5,6,7,9,10,12,15,18,20



**BST search**

* Searching BST is to find a node with key value matching with the given search value
* Idea of the BST search algorithm: At a node of BST if the search key is the node key then return the node; else if the search key is less than the node key, then search the left sub-tree; else search the right sub-tree
* No need to traverse all nodes of the tree, such save time by visiting much less number of nodes

**Recursive search**

TNODE \*recursive\_search(TNODE \*root, int key){

If (root==NULL) return NULL;

Else if (key==root->data) return root;

Else if (key<root->data)

Return recursive\_search(root->left, key);

Else

Return recursive\_search(root->right, key);

}

Time: O(h), space: O(h)

**Iterative choice**

TNODE \*iterative\_search (TNODE \*root, int key){

While(root){

If(key==root->data)

Return root;

Else if (key<root->data)

Root = root->left

Else

Root = root->right;

}

Return NULL;

}

Time: O(h), Space: O(1)

**BST recursive insert**

Void recursive\_insert(TNODE \*\*rootp, TNODE \*newnp){

If(\*rootp==NULL){

\*rootp = newnp;

}else{

If(newnp->data == (\*rootp)->data){

Printf(“The same key value is found”);

Return;

} else if (newnp->data < (\*rootp)->data)

Recursive\_insert(&(\*rootp)->left, newnp);

Else

Recursive\_insert(&(\*rootp)->right, newnp);

}

}

Time: O(h), space: O(h)

**BST insertion by iterative algorithm**

Void iterative\_insert(TNODE \*\*rootp, TNODE \*newnp){

If(\*rootp==NULL)

\*rootp = newnp;

Else{

TNODE \*p = \*rootp;

While(1){

If(newnp->data == p->data){

Printf(“The same key value is found”);

Break;

} else if(newnp->data < p->data){

If(p->left == NULL){ p->left = newnp; break;}

Else p = p->left;

} else{

If(p->right == NULL) { p->right = newnp; break;}

Else p = p->right;

}

}

}

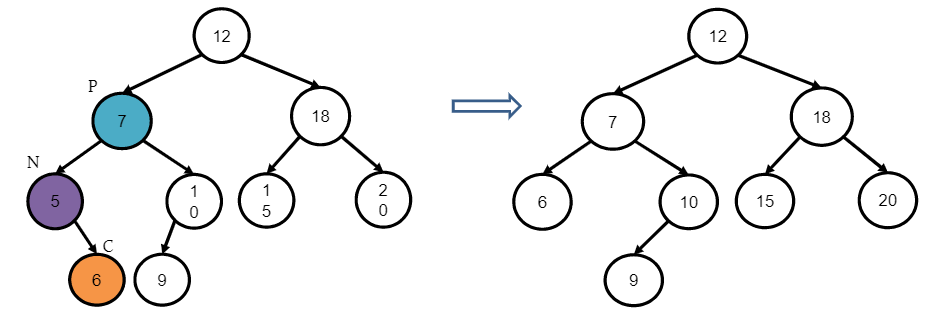
}

Time: O(h), space: O(1)

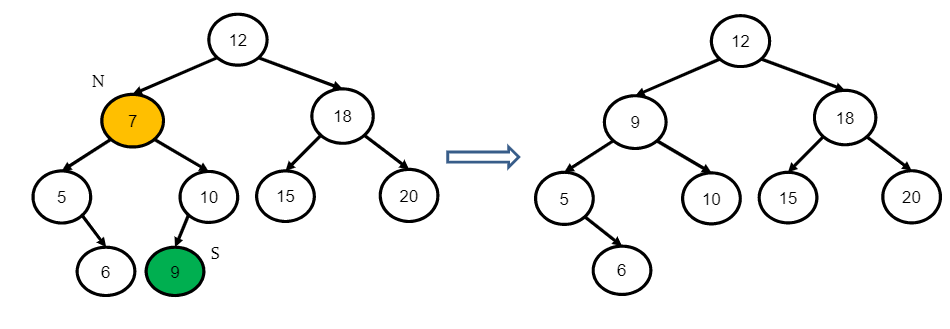
**Delete operation**

Three cases:

1. The delete node has no children, just delete it
2. The delete node has only one child (either left or right)
   1. Suppose N is the delete node, P is the parent of N, and C is the child node of N. If N is the left child if P, make C the left child of P. If N is the right child of P, make C the right child of P, Delete N.
   2. Ex. Delete node 5

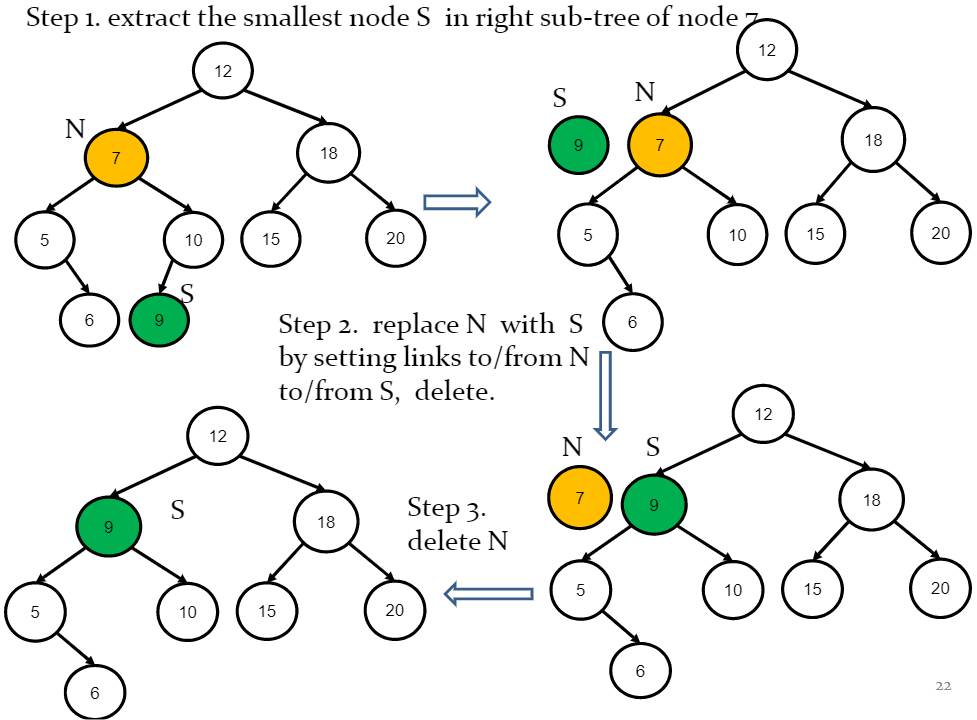


1. The delete node has two children
   1. Replace the node’s value with the smallest value in the right subtree (or the largest value in the left sub tree)
   2. Delete the smallest value node in the right sub tree (or largest value node in the left sub tree)
   3. Ex. Delete node 7



**Better solution for case 3**

1. Extract the smallest(or largest) node in right (or left) subtree
2. Replace the delete node with the extracted node
3. Free the delete node



**Find the smallest key node**

TNODE \*recursive\_find\_smallest\_node(TNODE \*root){

If(root==NULL || root->left == NULL) return root;

Else return recursive\_find\_smallest\_element(root->left);

}

Time: O(h), space: O(h)

TNODE \*iterative\_find\_smallest\_node(TNODE \*root){

If(root==NULL) return root;

While(root->left !=NULL) root = root->left;

Return root;

}

Time: O(h), space: O(1)

**Delete by recursive algorithm**

Void recursive\_delete(TNODE \*\*rootp, int key){

TNODE \*root - \*rootp, \*tnp;

If(root==NULL) return;

Else if(key==root->data){

If(root->left == NULL && root ->right == NULL){ //case 1

Free(root); \*rootp = NULL

}

Else if(root->left != NULL && root ->right ==NULL){ //case 2

Tnp = root->left; \*rootp = tnp; free(root);}

Else if( root->left == NULL && root->right != NULL){ //case 2

Tenp = root->right; \*rootp = tnp; free (root);}

Else if(root->left !=NULL &&root->right !=NULL){ //case 3

Tnp = extract\_smallest\_node(&root->right);

Tnp ->left = root->left;

Tnp->right = root->right;

\*rootp = tnp; free (root);

}

} else{

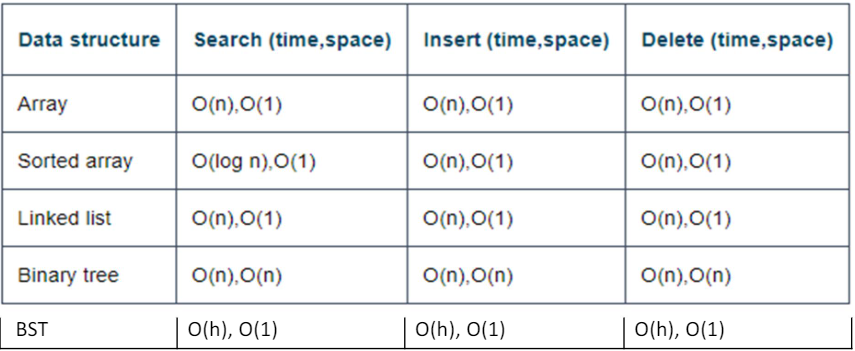
If(key<root->data) delete (&root->left, key);

Else delete (&root->right, key);

}

} time: O(h), space: O(h)

See code example 23 for BST delete by iterative algorithm



**Balanced BST**

* A BST is not valanced in general. In worst case the height of a BST is O(n). The next effort is to create valanced BST with height o(log n), so as to improve the performance of search, insert and delete time operations to O(log n).
* Balanced BST, i.e. height of left and right subtrees are equal or not much differences at any node
* Ex. A perfect binary BST of n nodes. The search/delete/insert operations can be done in time O(log n)