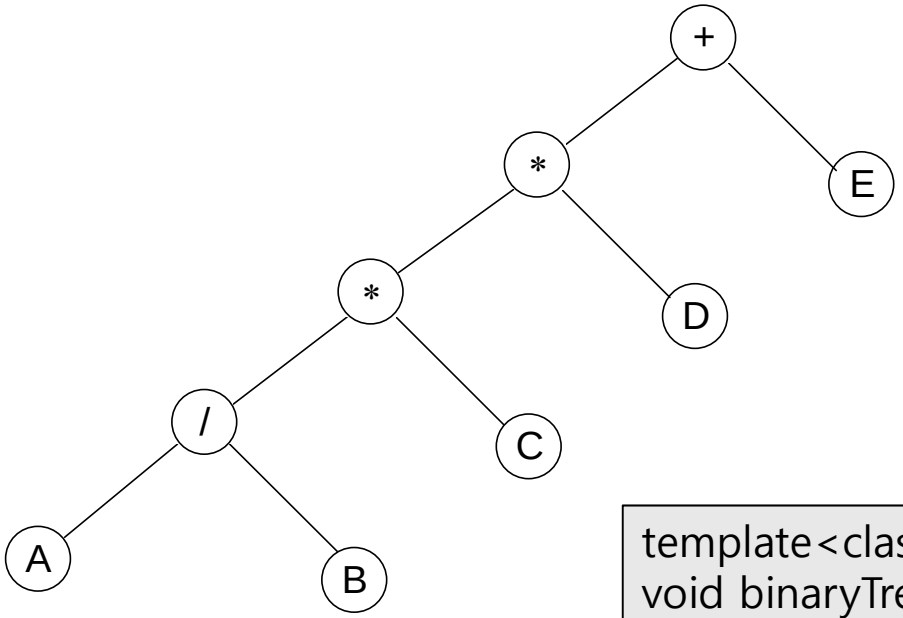
Tree Traversal

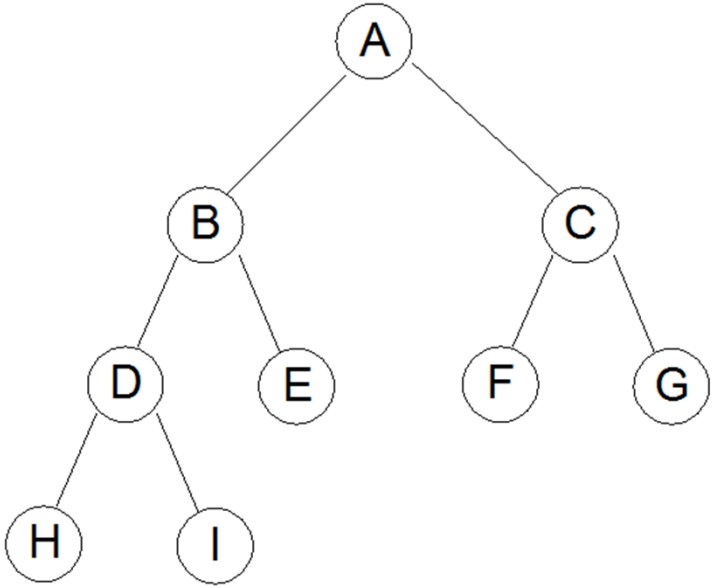


What's the output?

Running time?

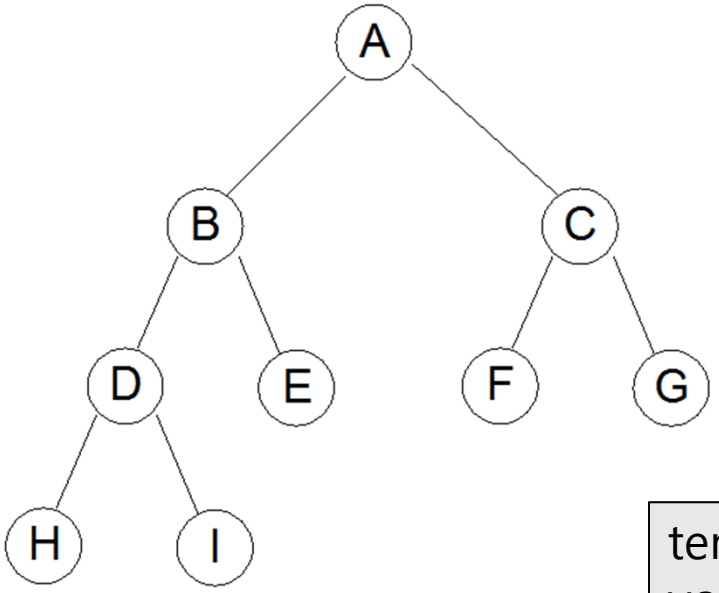
```
template<class T>
void binaryTree<T>::preOrder(treeNode * curr_node) {
      if (curr_node != NULL){
            print(curr_node->data);
            preOrder(curr_node->left);
            preOrder(curr_node->right);
      }
}
```

Traversal in a broader view



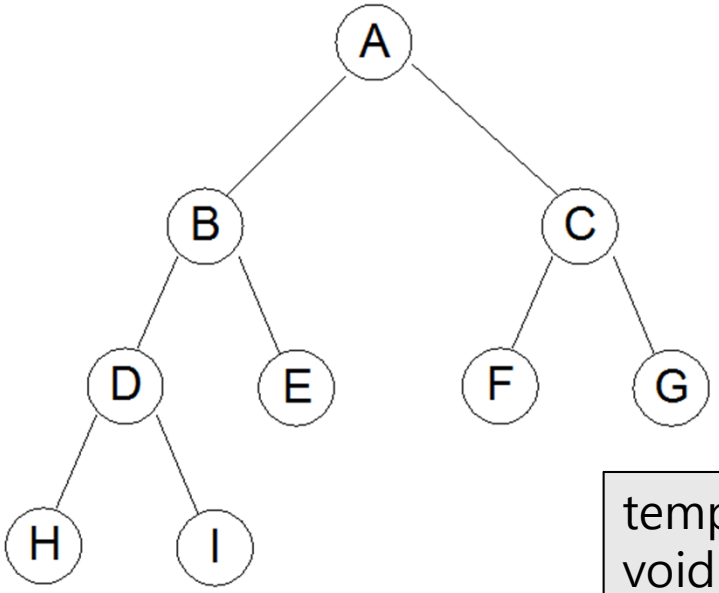

```
template<class T>
treeNode * binaryTree<T>::copy(treeNode * curr_node)




}
```

Traversal in another view




```
template<class T>
void binaryTree<T>::clear(treeNode *  curr_node) {
      if (curr_node != NULL) {
            clear(curr_node>left);
            clear(curr_node->right);
            delete curr_node;
            curr_node = null;
      }
}
```

Traversal in some different way…



A B C D E F G H I

```
template<class T>
void binaryTree<T>::levelOrder(treeNode * curr_node {




}
```

Dictionary

Example: <course number, course title>

ID (key)	Data (value)
EE105	Intro. Programming
EE220	Logic Design
EE240	Data Structure
CS300	Algorithms
MATH225	Discrete Math
.....	.....

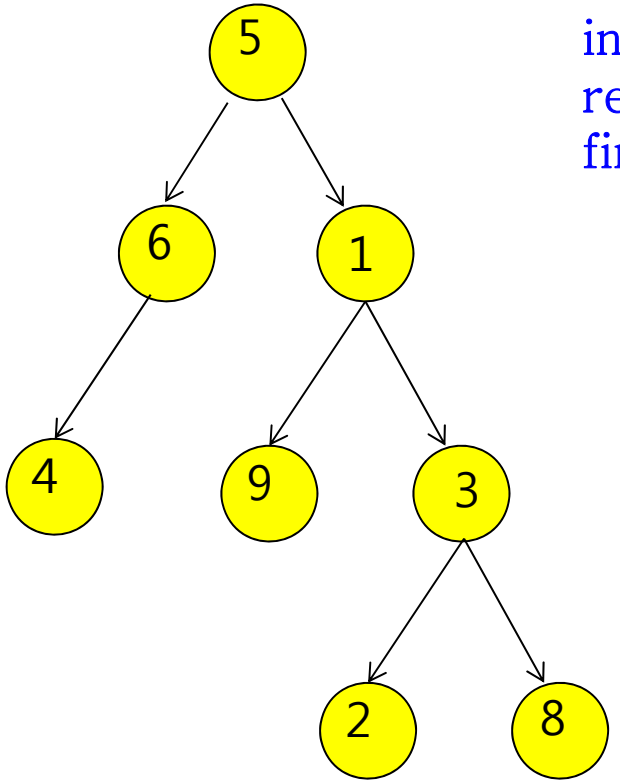
Note: A large portion of data is structured and stored in a form of "relational" database.

Dictionary is a basic form of the relational database.

A dictionary is a structure supporting the following functions:

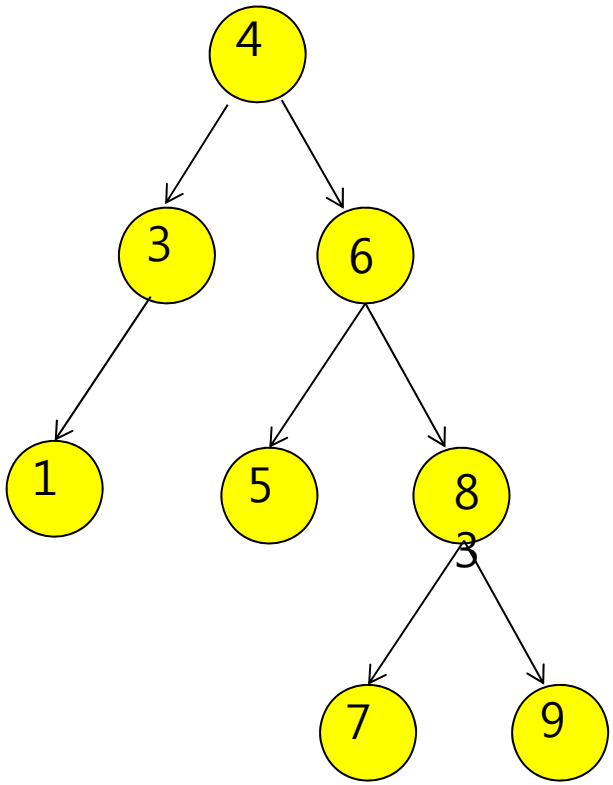
```
void insert(keyType & k, dataType & d);
dataType remove(keyType & k);
dataType find(keyType & k);
```

Implementing Dictionary as a binary tree



insert(k, d);
remove(k);
find(k);

Binary search tree



A Binary Search Tree (BST) is a binary tree, T, such that:

- _____, OR
- T = {r, T_L , T_R } and

  x ∈ T_L → _____

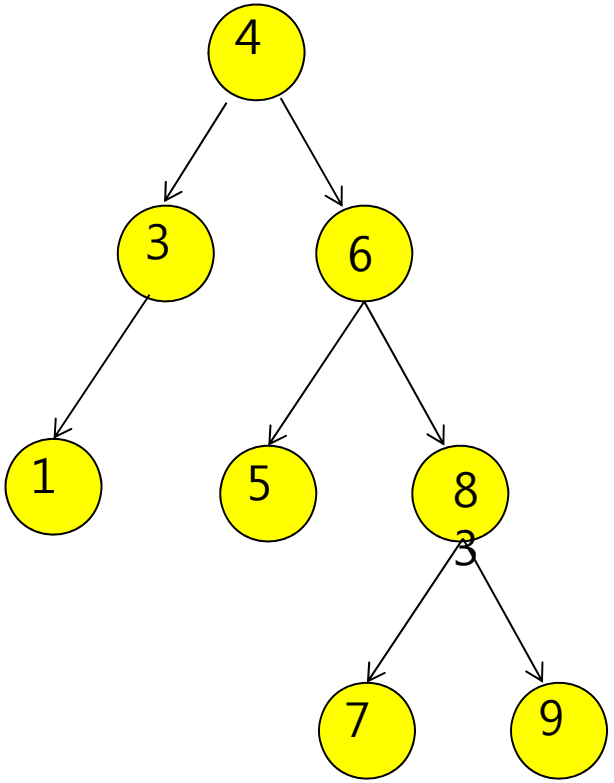
  x ∈ T_R → _____

and

Dictionary ADT with BST implementation

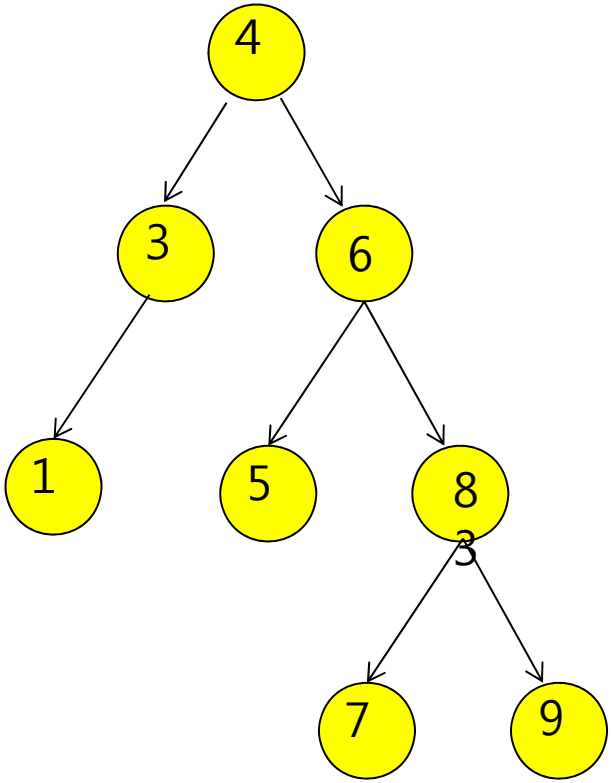
```
template <class K, class D>
class BST {
public: // Driver
     void insert(const K&, const D&);
     void remove(const K&);
     D find(const K&);
     void traversal();
private:
    // Workhorse
    void insert(treeNode *&, K &, D &);
     ……
    class treeNode {
    public:
         K key;
         D data;
         treeNode * left;
         treeNode * right;
    };
    treeNode * root;
};
```

find( ):



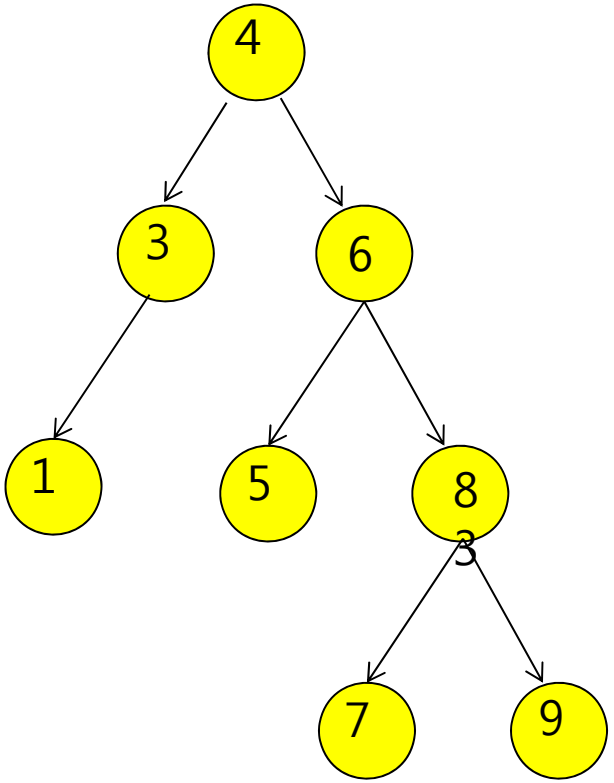

```
_____________________(treeNode * curr_node,
                                    const K & key) {
    if (curr_node == NULL)

    else if (curr_node->key == key)


    else if


    else

}
```

insert( ):



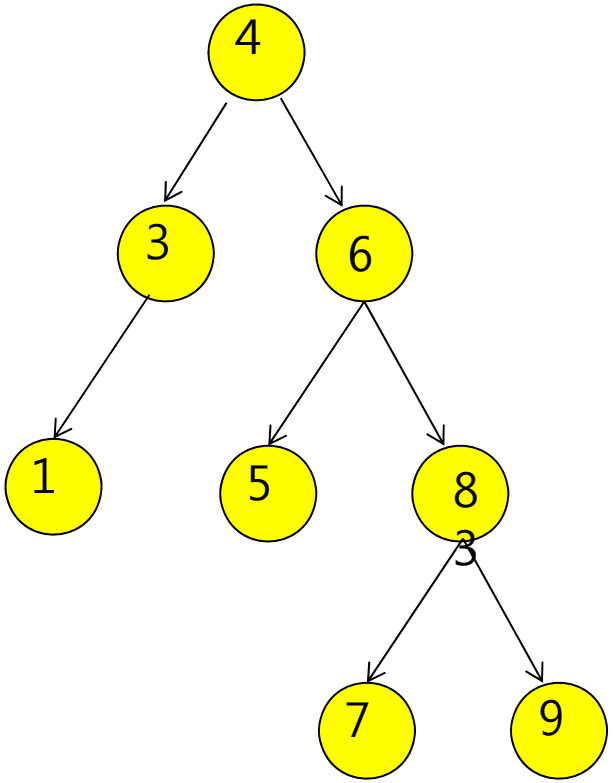

```
_____________________________(treeNode * curr_node,
                        const K & key, const D & data){
    if (curr_node == NULL)

    else if (curr_node->key == key)

    else if (key < curr_node->key)

    else

}
```

remove( ):




```
_____(treeNode * & curr_node,
                                  const K & k) {
    if (curr_node != NULL) {
        if (k == curr_node->key)

            doRemoval(curr_node);

        else if (k < curr_node->key)

            remove(curr_node->left, k);

        else

            remove(curr_node->right, k);
    }
}
```

remove( )… examples

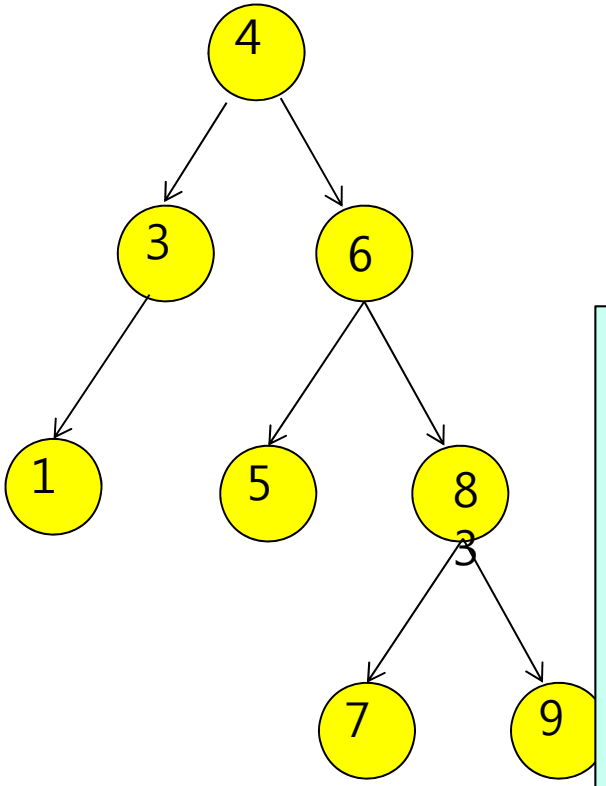


Tree.remove(7); →

Tree.remove(3); →

Tree.remove(6); →

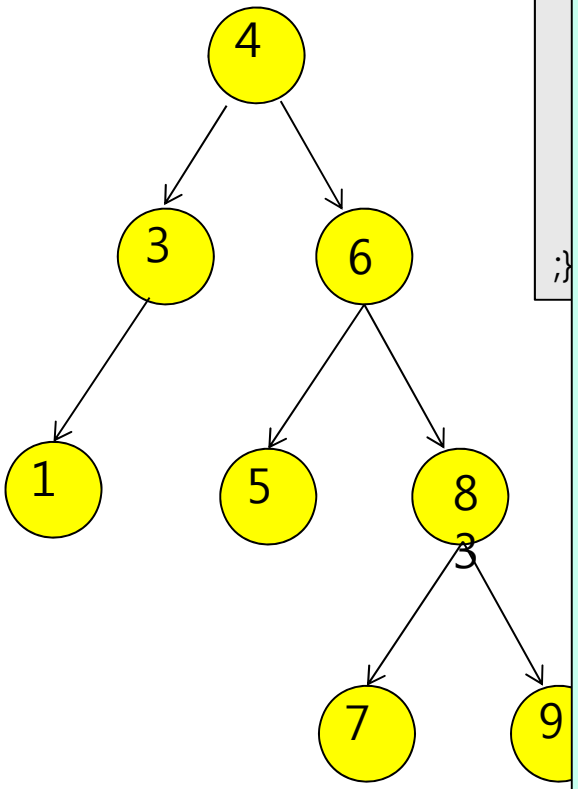
remove( ):




```
_____________________(treeNode * & curr_node,
                                    const K & k) {
    if (curr_node != NULL) {
        if (k == curr_node->key)
            doRemoval(curr_node);
        else if (k < curr_node->key)
            remove(curr_node->left, k);
        else
            remove(curr_node->right, k);
    }
```

```
void BST<K,V>::doRemoval(treeNode * & curr_node) {
    if ((curr_node->left == NULL) &&
        (curr_node->right == NULL))
            _______ChildRemove(curr_node);
    else if ((curr_node->left != NULL) &&
            (curr_node->right != NULL))
            _______ChildRemove(curr_node);
    else
            ________ChildRemove(curr_node)
;}
```

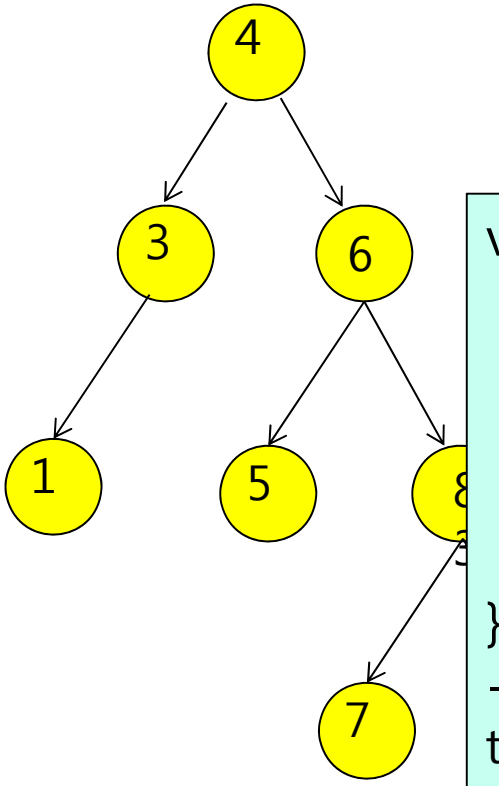
remove( ):




```
void BST<K,V>::doRemoval(treeNode * & curr_node) {
   if ((curr_node->left == NULL) &&
      (curr_node->right == NULL))
;}
```

```
void BST<K,V>::noChildRemove(treeNode * &
                                        curr_node) {

   treeNode * temp = curr_node;
   curr_node = NULL;
   delete temp;
}
---------------------------------

void BST<K, V> oneChildremove(treeNode * &
                                        curr_node) {




}
```

remove( ):




```
void BST<K,V>::doRemoval(treeNode * & curr_node) {
   if ((curr_node->left == NULL) &&
      (curr_node->right == NULL))
            _____ChildRemove(curr_node);
   else if ((curr_node->left != NULL) &&
            (curr_node->right != NULL))
            _____ChildRemove(curr_node);
   else
```

```
void BST<K,V>::twoChildRemove(treeNode * &
                                        curr_node) {

   treeNode * iop = rightMostChild(curr_node>left);
   curr_node->key = iop->key; // trick!!
   doRemoval(iop);
}
-----------------------------
trreeNode * & BST<K>::rightMostChild(treeNode * & curr_n
ode) {
```