1. Design two methods that can add a value to a BST. First, design the recursive one that takes the value to add and pointer to the tree to add it to as parameters. Now, think about how you as a user of a BST class would want to call add. I bet you would like to call something like:

```
myBST.add( value );
```

   Note that the public add doesn't take any extra parameters. How can you design around this?

```
template <typename ItemType, typename KeyType>
BinaryNode<ItemType>* BST<ItemType, KeyType>::RecAdd(BinaryNode<ItemType>*
subTreePtr,  BinaryNode<ItemType>* newNodePtr)
{
    if(subTreePtr == nullptr)
    {
        return newNodePtr;
    }
    else
    {
        BinaryNode<ItemType>* leftPtr = subTreePtr->getleftChildPtr();
        BinaryNode<ItemType>* rightPtr = subTreePtr->getrightChildPtr();
        if((subTreePtr->getItem().getAN())>(newNodePtr->getItem().getAN()))
        {
            leftPtr=RecAdd (leftPtr,newNodePtr);
            subTreePtr->setleftPtr(leftPtr);
        }
        else
        {
            rightPtr=RecAdd(rightPtr,newNodePtr);
            subTreePtr->setrightPtr(rightPtr);
        }
        return subTreePtr;
    }
}

template <typename ItemType, typename KeyType>
void BST<ItemType, KeyType>::Add(const ItemType& newEntry) throw (NotFoundException)
{
   BinaryNode<ItemType>* newPtr = new BinaryNode<ItemType> (newEntry);
   m_root = RecAdd(m_root, newPtr);
}
```

2. Now list out all the methods your want in your BST class. Remember, public methods like add, search, and remove won't be able to pass in pointers to nodes in the tree.

```
//constructor
BST();
//copyTree constructor
BST(const BST<ItemType, KeyType>& tree);
//destructor
~BST();


//check if the item has no children
bool isLeaf(BinaryNode<ItemType>* subtree);

//delete a value from tree
bool remove(const KeyType& target);
```

```
        //get fouctions
        ItemType getEntry(const KeyType& aKey) const throw(NotFoundException);

        //find a keyword in tree
        bool search(const KeyType& aKey) const;
        void Add(const ItemType& newEntry) throw(NotFoundException);

        // inserts it in a leaf at that point
        BinaryNode<ItemType>*
    RecAdd(BinaryNode<ItemType>*subTreePtr,  BinaryNode<ItemType>* newNodePt)

        //deep copy the whole tree
        BinaryNode<ItemType>* copy(const BinaryNode<ItemType>* treePtr) const;

         // Recursive traversal methods
         void inorderTraverse(void visit(ItemType&)) const;
         void preorderTraverse(void visit(ItemType&)) const;
         void postorderTraverse(void visit(ItemType&)) const;


        private:

        // Recursively deletes all nodes from the tree
        void destroyTree(BinaryNode<ItemType>* subTreePtr);

        //remove the target Value from a subtree
        BinaryNode<ItemType>* removeValue(BinaryNode<ItemType>* subTreePtr, const
    KeyType& target);

//remove the node which have target value
      BinaryNode<ItemType>* removeNode( BinaryNode<ItemType>* nodePtr);

      //switch node.
      BinaryNode<ItemType>* removeLeftmostNode(BinaryNode<ItemType>*
    nodePtr,ItemType& inorderSuccessor);

      // Returns a pointer to the node containing the given value,
      // or nullptr if not found.
      BinaryNode<ItemType>* findNode(BinaryNode<ItemType>* subTreePtr, constKeyType&
    target) const

      //visit methods
      void inorder( void visit(ItemType&), BinaryNode<ItemType>* treePtr) const;
      void preorder( void visit(ItemType&), BinaryNode<ItemType>* treePtr) const;
      void postorder( void visit(ItemType&), BinaryNode<ItemType>* treePtr) const;
```

3. Why are there two templated types? What advantage do we get by being able to provide a string as a search term, but receive an object as a return value.

   These two templated types are different, the Itemtype is the templated type of classes, and the Keytype is the type of the values in the class. By provide a string, we can get any value when we need in the right class.

4. If we only had one templated type, how would that limit our BST?

We cannot use search method anymore, it also influenced removed function. Since we only have one templated type, it will be the type of the class or the type of a value. Thus, when the class has more than one value with different types, but the templated type only get a single type, and we won't find a Pokemon's ID and Japanese by searching its US name.

5. Describe your remove algorithm in words. What nodes do you pick as a replacement candidate and why?

First, checking whether the item to be removed exist or not.Second, checking whether the item in the node is a leaf or not. If it is a leaf, delete the node (pointer points to nullptr).If it has only one child, creating a temporary connecting node (cutting from the tree) and connect its child and its parent.

If it has two children, find a minimum value in the right/left subtree, then replace value of the node to be removed with found minimum, in the end delete the node.

For replacement candidate, we usually choose the minimum leaf node from the left/right tree, because it allows replace only one value without subtrees.