template <class Object>
BinaryNode<Object> *Tree<Object> :: find(const Object &x, BinaryNode<Object> *t) const {
    BinaryNode<Object> *ptr;

    if (t == NULL)
        return NULL;
    else if (x == t->element)
        return t;
    else if (ptr = find(x, t->left))
        return ptr;
    else
        return(ptr = find(x, t->left));
}
Find : Static K-ary

template <class Object>
KaryNode<Object> *KaryTree<Object> :: find(const Object &x, KaryNode<Object> *t) const {
    KaryNode<Object> *ptr;
    if (t == NULL)
        return NULL;
    else if (x == t->element)
        return t;
    else {
        i = 0;
        while ((i < MAX_CHILDREN) &&
            !(ptr = find(x, t->children[i])) i++;
        return ptr;
    }
}

Find : Sibling/Child

template <class Object>
KTreeNode<Object> *Tree<Object> :: find(const Object &x, KTreeNode<Object> *t) const {
    KTreeNode<Object> *ptr;
    if (t == NULL)
        return NULL;
    else if (x == t->element)
        return t;
    else if (ptr = find(x, t->firstChild))
        return ptr;
    else
        return(ptr = find(x, t->nextSibling));
}
Insert : Static Binary

Insert : Static K-ary
Insert : Sibling/Child

Remove : Static Binary
Remove : Static K-ary

Remove : Sibling/Parent
Binary Search Tree

Def: A *Binary Search Tree* is a Binary Tree in which, at every node \( v \), the value stored in the left child node is less than the value at \( v \) and the value stored in the right child is greater.

The elements in the BST must be comparable. Duplicates are not allowed.

---

BST Implementation

The SearchTree ADT

- A *search tree* is a binary search tree in which are stored homogeneous elements with no duplicates.
- It is dynamic.
- The elements are ordered in the following ways
  - inorder -- as dictated by operator\(<\`
  - preorder, postorder, levelorder -- as dictated by the structure of the tree
- Each BST maintains a simple object, known as `Item_Not_Found`, that is guaranteed to not be an element of the tree. `Item_Not_Found` is provided to the constructor.
BinarySearchTree class

template <class Comparable>
class BinarySearchTree {
    public:
        BinarySearchTree(const Comparable &notFnd);
        BinarySearchTree (const BinarySearchTree &rhs);
        ~BinarySearchTree();
        const Comparable &findMin() const;
        const Comparable &findMax() const;
        const Comparable &find(const Comparable &x) const;
        bool isEmpty() const;
        void printTree() const;
        void makeEmpty();
        void insert (const Comparable &x);
        void remove (const Comparable &x);
        const BinarySearchTree &operator=(const BinarySearchTree &rhs);

    private:
        BinaryNode<Comparable> *root;
        const Comparable ITEM_NOT_FOUND;
        const Comparable &elementAt(BinaryNode<Comparable> *t) const;
        void insert (const Comparable &x, BinaryNode<Comparable> * &t) const;
        void remove (const Comparable &x, BinaryNode<Comparable> * &t) const;
        BinaryNode<Comparable> *findMin(BinaryNode<Comparable> * t) const;
        BinaryNode<Comparable> *findMax(BinaryNode<Comparable> * t) const;
        BinaryNode<Comparable> *find(const Comparable &x, BinaryNode<Comparable> * t) const;
        void makeEmpty(BinaryNode<Comparable> * &t) const;
        void printTree(BinaryNode<Comparable> * t) const;
        BinaryNode<Comparable> *clone(BinaryNode<Comparable> * t) const;
    };
BinarySearchTree Implementation

```cpp
template <class Comparable>
const Comparable &BinarySearchTree<Comparable> ::
    find(const Comparable &x) const {
    return elementAt(find(x, root)); }

template <class Comparable>
const Comparable &BinarySearchTree<Comparable> ::
    elementAt(BinaryNode<Comparable> *t) const {
    return t == NULL ? ITEM_NOT_FOUND : t->element;
}

template <class Comparable>
BinaryNode<Comparable> *BinarySearchTree<Comparable> ::
    find(const Comparable &x, BinaryNode<Comparable> *t) const {
    if (t == NULL) return NULL;
    else if (x < t->element) return find(x, t->left);
    else if (t->element < x) return find(x, t->right);
    else return t; // Match
}
```

Performance of find

Search in randomly built BST is $O(\log n)$ on average
– but generally, a BST is not randomly built

Asymptotic performance is $O(h)$ in all cases
Predecessor and Successor in BST

Predecessor of a node v in a BST is the node that holds the data value that immediately precedes the data at v in order.

Finding predecessor
- v has a left subtree
  - then predecessor must be the largest value in the left subtree (the rightmost node in the left subtree)
- v does not have a left subtree
  - predecessor is the first node on path back to root that does not have v in its left subtree

Successor immediately follows that at v.
- v has right subtree: succ is smallest in right subtree
- v does not have right subtree: succ is first node on path back to root that does not have v in its right subtree

The remove Operation

```plaintext
template <class Comparable>
void BinarySearchTree<Comparable>::remove(const Comparable &x, 
BinaryNode<Comparable> *&t) const {
    if (t == NULL)
        return; // item not found, do nothing
    if (x < t->element)
        remove(x, t->left);
    else if (t->element < x)
        remove(x, t->right);
    else if ((t->left != NULL) && (t->right != NULL)) {
        t->element = findMin(t->right)->element;
        remove (t->element, t->right);
    }
    else {
        BinaryNode<Comparable> *oldNode = t;
        t = (t->left != NULL) ? T->left : t->right;
        delete oldNode;
    }
} 
```
The insert Operation

template <class Comparable>
void BinarySearchTree<Comparable>::insert(const Comparable &x) {
    insert (x, root);
}

template <class Comparable>
void BinarySearchTree<Comparable>::insert(const Comparable &x, 
    BinaryNode<Comparable> *&t) const {
    if (t == NULL)
        t = new BinaryNode<Comparable>(x, NULL, NULL);
    else if (x < t->element)
        insert (x, t->left);
    else if(t->element < x)
        insert (x, t->right);
    else
        ; // Duplicate; do nothing
}

Implementation of makeEmpty

template <class Comparable>
void BinarySearchTree<Comparable>::makeEmpty() { 
    makeEmpty(root);
}

template <class Comparable> 
void BinarySearchTree<Comparable>::makeEmpty(BinaryNode<Comparable> *&t) const {
    if (t != NULL) {
        makeEmpty (t->left);
        makeEmpty (t->right);
        delete t;
    }
    t = NULL;
}
Tree Iterators

Could provide separate iterators for each desired order
- `Iterator<T> *GetInorderIterator();`
- `Iterator<T> *GetPreorderIterator();`
- etc

Approach 1: Store traversal in list. Return list iterator for list.

```cpp
Iterator<T> BinaryTree::GetInorderIterator()
{
    List<T> *lst = new ArrayList<T>
    FullListInorder(list, getRoot());
    return list->GetIterator();
}
```

```cpp
void FillListInorder(ArrayList<T> *lst, Bnode<T> *node)
{
    if (node == NULL) return;
    FillListInorder(list, node->left);
    lst->Append(node->data);
    FillListInorder(lst, node->right);
}
```

Tree Iterators (cont)

Approach 2: store traversal in stack to mimic recursive traversal

```cpp
template <class T>
class InOrderIterator : public Iterator {
private:
    Stack<T> _stack;
    BinaryTree<T> *_tree;
public:
    InOrderIterator(BinaryTree<T> *t);
    bool hasNext() {return (!_stack.isEmpty());}
    T Next();
};
```
Tree Iterators (cont)

InOrderIterator(BinaryTree<T> *t) {
    _tree t;
    Bnode<T> *v = getRoot();
    while (v != NULL) { // push root and all left descendents
        _stack.Push(v);
        v = v->left;
    }
}

T Next() {
    Bnode<T> *top = _stack.Top();
    _stack.Pop();
    Bnode<T> *v = top->right;
    while (v != NULL) { // push right child and all left descendents
        _stack.Push(v);
        v = v->left;
    }
}