A **binary search tree** is used to store ordered data to allow efficient queries and updates.

### 13.1 Binary Search Trees

A **binary search tree** is a binary tree with values at the nodes such that:

*left descendants are smaller, right descendants are bigger. (One can adapt this to allow repeated values.)*

This assumes the data comes from a domain in which there is a **total order**: you can compare every pair of elements (and there is no inconsistency such as \( a < b < c < a \)). In general, we could have a large object at each node, but the object are sorted with respect to a **key**.

Here is an example:

![Binary Search Tree Diagram]

An **inorder traversal** is when a node is visited after its left descendants and before its right descendants. The following recursive method is started by the call `inorder(root)`.

```c
void inorder(Node *v) {
    inorder(v->left);
    visit v;
    inorder(v->right);
}
```

*An inorder traversal of a binary search tree prints out the data in order.*
13.2 Insertion in BST

To find an element in a binary search tree, you compare it with the root. If larger, go right; if smaller, go left. And repeat. The following method returns `nullptr` if not found:

```c
Node *find(key x) {
    Node *t=root;
    while( t!=nullptr && x!=t->key )
        t = ( x<t->key ? t->left : t->right );
    return t;
}
```

Insertion is a similar process to searching, except you need a bit of look ahead. Here is a strange-looking recursive version:

```c
Node *insert(ItemType &elem, Node *t) {
    if( t==nullptr )
        return new Node( elem );
    else {
        if( elem.key<t->key )
            t->left = insert(elem,t->left);
        else if( elem.key>t->key )
            t->right = insert(elem,t->right);
        return t;
    }
}
```

13.3 Removal from BST

To remove a value from a binary search tree, one first finds the node that is to be removed. The algorithm for removing a node \( x \) is divided into three cases:

- **Node \( x \) is a leaf.** Then just delete.

- **Node \( x \) has only one child.** Then delete the node and do “adoption by grand-parent” (get old parent of \( x \) to point to old child of \( x \)).

- **Node \( x \) has two children.** Then find the node \( y \) with the next-lowest value: go left, and then go repeatedly right (why does this work?). This node \( y \) cannot have a right child. So swap the values of nodes \( x \) and \( y \), and delete the node \( y \) using one of the two previous cases.
The following picture shows a binary search tree and what happens if 11, 17, or 10 (assuming replace with next-lowest) is removed.

All modification operations take time proportional to depth. In best case, the depth is $O(\log n)$ (why?). But, the tree can become “lop-sided”—and so in worst case these operations are $O(n)$.

### 13.4 Finding the k’th Largest Element in a Collection

Using a binary search tree, one can offer the service of finding the $k$’th largest element in the collection. The idea is to keep track at each node of the size of its subtree (how many nodes counting it and its descendants). This tells one where to go.

For example, if we want the 4th smallest element, and the size of the left child of the root is 2, then the value is the minimum value in the right subtree. (Why?) (This should remind you of binary search in an array.)

### Sample Code

Here is code for a binary search tree.

<table>
<thead>
<tr>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSTNode.h</td>
</tr>
<tr>
<td>BinarySearchTree.h</td>
</tr>
<tr>
<td>BinarySearchTree.cpp</td>
</tr>
</tbody>
</table>