Chapter 6

Binary Search Trees.

6.1 Binaries Trees and How to Encode Them.

We have already seen, in heaps, that organizing our data in a graph-like structure can offer a speed advantage. For future applications, and in particular for maintaining data in sorted order, we will have to encode our data using binary trees. These trees may not be almost complete and also have to support pointer manipulations, specifically placing a binary tree as a left or right subtree of a given node. To enable this, we will have to treat the **right**, **left**, and **parent** as variables, in contrast to heaps where they are determined completely by the node index. We begin this section by stating definitions.

Definition 6.1.1.

1. *Binary Tree: A tree in which any vertex has at most two children.*

2. *A descendant of vertex x is a vertex in the subtree whose root is x. A left descendant of vertex x is either a vertex in the subtree whose root is the left child of x, or x's left child.*

3. *An ancestor of x is a vertex to which x belongs as a descendant.*

4. *A leaf is a vertex without children.*

5. *Height of vertex x is the length of the longest simple path (without cycles) between x and one of the leaves.*

6. *Height of the tree is the height of its root, which is usually denoted by h.*

We encode a binary tree by associating a field to each vertex x, representing its right, left children, and parent. We use the notation x.left to refer to the left child of x, although the physical implementation may differ conceptually. For example, the way binary trees are implemented in Cormen is through 4 arrays. The first stores the value of x, while the others store pointers of specific types. For instance, the array LEFT, where LEFT.x stores the left child of x.

If nothing else has been mentioned, then we can assume that we can add additional fields to the vertices.

6.2 Binary Search Trees.

Binary search tree (BST) is a binary tree which any node x of it:

1. Contains a field key, storing a number x.key.
2. Any left descendant y of x satisfies y.key $\leq x$.key.
3. Any right descendant y of x satisfies y.key $\geq x$.key.

Question. Let T be a binary search tree, Where are the minimum and maximum values of T? (most left and right nodes).

Definition 6.2.1. *Let T be a binary search tree, and let x be a node belonging to it. The predecessor of x will be defined as a vertex y such that y.key $\leq x$.key and y.key is maximal among the nodes satisfying this condition. If we were to set the values of T in sorted order, then the predecessor of x would be located on its left. The successor of x will be defined as y, where x is the predecessor of y.*

Functionalitiy of BST.

1. Search(T, key): returns a pointer to the vertex whose key equals key.
2. Min(T): returns a pointer to the vertex with the minimum value in T.
3. Max(T): returns a pointer to the vertex with the maximum value in T.
4. Predecessor(x): returns a pointer to the predecessor of x.
5. Successor(x): returns a pointer to the successor of x.
6. Insert(T, key): inserts key into T (creates a new vertex).
7. Delete(T, x): removes x from T.
8. Inorder(T): outputs T's keys in sorted order.

```
Data: T - tree, key - key to search for
Result: pointer to vertex with key equal to key
1  Function Search(T, key)
2      if T is empty then
3          return null;
4      end
5      if T.key equals key then
6          return T;
7      end
8      if key is less than T.key then
9          return Search(T.left, key);
10     end
11     else
12         return Search(T.right, key);
13     end
14 end
```

```
Data: T - tree
Result: pointer to vertex with minimum value in T
1  Function Min(T)
2      if T is empty then
3          return null;
4      end
5      if T.left is empty then
6          return T;
7      end
8      return Min(T.left);
9  end
```

```
Data: T - tree
Result: pointer to vertex with maximum value in T
1  Function Max(T)
2      if T is empty then
3          return null;
4      end
5      if T.right is empty then
6          return T;
7      end
8      return Max(T.right);
9  end
```

```
   Data: x - vertex
   Result: pointer to predecessor of x
 1 Function Predecessor(x)
 2     if x.left is not empty then
 3         return Max(x.left);
 4     end
 5     y ← x.parent;
 6     while y is not empty and x is y.left do
 7         x ← y;
 8         y ← y.parent;
 9     end
10     return y;
11 end
```

```
   Data: x - vertex
   Result: pointer to successor of x
 1 Function Successor(x)
 2     if x.right is not empty then
 3         return Min(x.right);
 4     end
 5     y ← x.parent;
 6     while y is not empty and x is y.right do
 7         x ← y;
 8         y ← y.parent;
 9     end
10     return y;
11 end
```

```
   Data: T - tree, key - key to insert
   Result: inserts key into T
 1 Function Insert(T, key)
 2 |   newNode ← create new vertex with key key;
 3 |   y ← null;
 4 |   x ← T;
 5 |   while x is not empty do
 6 |   |   y ← x;
 7 |   |   if key is less than x.key then
 8 |   |   |   x ← x.left;
 9 |   |   end
10 |   |   else
11 |   |   |   x ← x.right;
12 |   |   end
13 |   end
14 |   newNode.parent ← y;
15 |   if y is null then
16 |   |   T ← newNode;
17 |   end
18 |   else if key is less than y.key then
19 |   |   y.left ← newNode;
20 |   end
21 |   else
22 |   |   y.right ← newNode;
23 |   end
24 end
```