

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 $x.\text{left}$ 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 .

Functionality 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.

```

SearchT, key current  $\leftarrow$  T.root while current  $\neq$  NIL do
    end
    current.key = key return current
current.key < key current  $\leftarrow$  current.right
else
    end
    current  $\leftarrow$  current.left return NIL
MinT current  $\leftarrow$  T.root while current.left  $\neq$  NIL do
    end
    current  $\leftarrow$  current.left return current
MaxT current  $\leftarrow$  T.root while current.right  $\neq$  NIL do
    end
    current  $\leftarrow$  current.right return current
Predecessorx if x.left  $\neq$  NIL then
    end

```

```

return Max $x.left$   $y \leftarrow x.parent$  while  $y \neq NIL$  and  $x = y.left$  do
end
 $x \leftarrow y$   $y \leftarrow y.parent$  return  $y$ 
Successor $x$  if  $x.right \neq NIL$  then
end
return Min $x.right$   $y \leftarrow x.parent$  while  $y \neq NIL$  and  $x = y.right$  do
end
 $x \leftarrow y$   $y \leftarrow y.parent$  return  $y$ 
Insert $T, key$   $newNode \leftarrow CreateNode$   $key$   $y \leftarrow NIL$   $x \leftarrow T.root$  while
 $x \neq NIL$  do
end
 $y \leftarrow x$  if  $key < x.key$  then
end
 $x \leftarrow x.left$  else
end
 $x \leftarrow x.right$   $newNode.parent \leftarrow y$  if  $y = NIL$  then
end
 $T.root \leftarrow newNode$   $newNode.key < y.key$   $y.left \leftarrow newNode$  else
end
 $y.right \leftarrow newNode$ 
Delete $T, x$  if  $x.left = NIL$  then
end
Transplant $T, x, x.right$   $x.right = NIL$  Transplant $T, x, x.left$  else
end
 $y \leftarrow Minx.right$  if  $y.parent \neq x$  then
end
Transplant $T, y, y.right$   $y.right \leftarrow x.right$   $y.right.parent \leftarrow y$  Transplant $T,$ 
 $x, y$   $y.left \leftarrow x.left$   $y.left.parent \leftarrow y$  Inorder $T$   $stack \leftarrow CreateStack$ 
 $current \leftarrow T.root$  while  $current \neq NIL$  or  $stack$  is not empty do
end
 $current \neq NIL$  Push $stack, current$   $current \leftarrow current.left$ 

```