

Binary Search Trees

Search trees are data structures that support many dynamic set operations including SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, and DELETE. Thus, a search tree can be used both as a dictionary and a priority queue.

Binary search trees are search trees in which the keys are stored in such a way as to satisfy the **binary search tree property**:

Let x be a node in a binary search tree. If y is a node in the left subtree of x , then $key[y] < key[x]$. If y is a node in the right subtree of x , then $key[x] \leq key[y]$.

1 Traversing

There are three ways to traverse binary search trees:

1. Inorder tree walk (visit the left subtree, the root, and right subtree)
2. Preorder tree walk (visit the root, the left subtree and right subtree)
3. Postorder tree walk(visit the left subtree, the right subtree, and the root)

INORDER-TREE-WALK(x)
1 if $x \neq \text{NIL}$

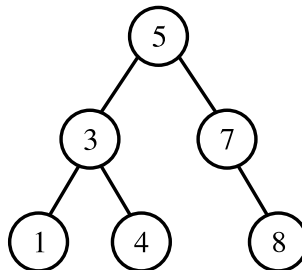


Figure 1: Binary Search Tree

```

2   then INORDER-TREE-WALK(left[x])
3       print key[x]
4       INORDER-TREE-WALK(right[x])

```

```

PREORDER-TREE-WALK(x)
1  if x ≠ NIL
2      then print key[x]
3          PREORDER-TREE-WALK(left[x])
4          PREORDER-TREE-WALK(right[x])

```

```

POSTORDER-TREE-WALK(x)
1  if x ≠ NIL
2      then POSTORDER-TREE-WALK(left[x])
3          POSTORDER-TREE-WALK(right[x])
4          print key[x]

```

2 Querying

The TREE-SEARCH algorithm comes in two flavours: the recursive and the iterative approaches. In the recursive approach, we examine the input node x and compare its *key* with the key we are looking for, k . If it matches, then we have found a node with the correct key; and can therefore return x . Otherwise, depending on the value of k and *key*[x], the node we are after is either in the left subtree or right subtree of node x .

```

TREE-SEARCH(x, k)
1  if x = NIL or k = key[x]
2      then return x
3  if k < key[x]
4      then return TREE-SEARCH(left[x], k)
5      else return TREE-SEARCH(right[x], k)

```

```

ITERATIVE-TREE-SEARCH(x, k)
1  while x ≠ NIL and k ≠ key[x]
2      do if k < key[x]
3          then x ← left[x]
4          else x ← right[x]
5  return x

```

The properties of the binary search tree makes the implementation of TREE-MINIMUM and TREE-MAXIMUM very simple. For minimum, simply start at the root and ask “is there a left child”. If so, visit it and repeat the question. The final left most node is the tree-minimum. Conversely, the right most node is the tree-maximum.

```

TREE-MINIMUM( $x$ )
1  while  $left[x] \neq \text{NIL}$ 
2      do  $x \leftarrow left[x]$ 
3  return  $x$ 

```

```

TREE-MAXIMUM( $x$ )
1  while  $right[x] \neq \text{NIL}$ 
2      do  $x \leftarrow right[x]$ 
3  return  $x$ 

```

The successor of a node x is the node with the next biggest key. The strategy for finding the successor of a node is two fold. Firstly, if the right child of node x is not empty, then the successor of x is simply the tree minimum of x 's right subtree. However, if the right child is empty, then the successor of x is the lowest ancestor of x whose left child is also an ancestor of x .

```

TREE-SUCCESSOR( $x$ )
1  if  $right[x] \neq \text{NIL}$ 
2      then return TREE-MINIMUM( $right[x]$ )
3   $y \leftarrow p[x]$ 
4  while  $y \neq \text{NIL}$  and  $x = right[y]$ 
5      do  $x \leftarrow y$ 
6       $y \leftarrow p[y]$ 
7  return  $y$ 

```

3 Insertion and Deletion

The operations of insertion and deletion cause the dynamic set represented by a binary search tree to change. The data structure must be modified to reflect this change, but in such a way that the binary search tree property continues to hold.

TREE-INSERT inserts a node z into tree T . The algorithm starts at the root of the tree and traverses down the tree following the rules of binary search trees. If the node to be inserted is smaller, then it has to be inserted into the left

subtree, else it should be inserted in the right subtree. The process continues until the appropriate subtree does not exist. That is to say, the left/right child is NIL.

```

TREE-INSERT( $T, z$ )
1   $y \leftarrow \text{NIL}$ 
2   $x \leftarrow \text{root}[T]$ 
3  while  $x \neq \text{NIL}$ 
4      do  $y \leftarrow x$ 
5          if  $\text{key}[z] < \text{key}[x]$ 
6              then  $x \leftarrow \text{left}[x]$ 
7              else  $x \leftarrow \text{right}[x]$ 
8   $p[z] \leftarrow y$ 
9  if  $y = \text{NIL}$ 
10     then  $\text{root}[T] \leftarrow z$  /* The tree was empty. */
11     else if  $\text{key}[z] < \text{key}[y]$ 
12         then  $\text{left}[y] \leftarrow z$ 
13         else  $\text{right}[y] \leftarrow z$ 

```

TREE-DELETE deletes a node z from tree T . The algorithm considers three cases: (1) the node z has no children, (2) z has one child, and (3) z has two children. In **case (1)**, z is simply deleted from T and $\text{left/right}[p[z]]$ updated to reflect the change. In **case (2)**, z is *spliced* out of the tree. This involves removing z and updating $\text{left/right}[p[z]]$ to point at z 's only child (see Figure 2). If z has two children, **case (3)**, z 's successor y is spliced out and z is replaced by y (see Figure 3).

```

TREE-DELETE( $T, z$ )
1  if  $\text{left}[z] = \text{NIL}$  or  $\text{right}[z] = \text{NIL}$ 
2      then  $y \leftarrow z$ 
3      else  $y \leftarrow \text{TREE-SUCCESSOR}(z)$ 
4  if  $\text{left}[y] \neq \text{NIL}$ 
5      then  $x \leftarrow \text{left}[y]$ 
6      else  $x \leftarrow \text{right}[y]$ 
7  if  $x \neq \text{NIL}$ 
8      then  $p[x] \leftarrow p[y]$ 
9  if  $p[y] = \text{NIL}$ 
10     then  $\text{root}[T] \leftarrow x$ 
11     else if  $y = \text{left}[p[y]]$ 
12         then  $\text{left}[p[y]] \leftarrow x$ 
13         else  $\text{right}[p[y]] \leftarrow x$ 
14  if  $y \neq z$ 
15     then  $\text{key}[z] \leftarrow \text{key}[y]$ 
16     copy  $y$ 's satellite data into  $z$ 

```

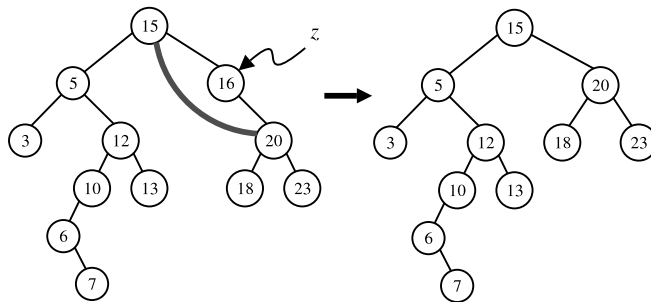


Figure 2: TREE-DELETE. If z has one child, we splice out z .

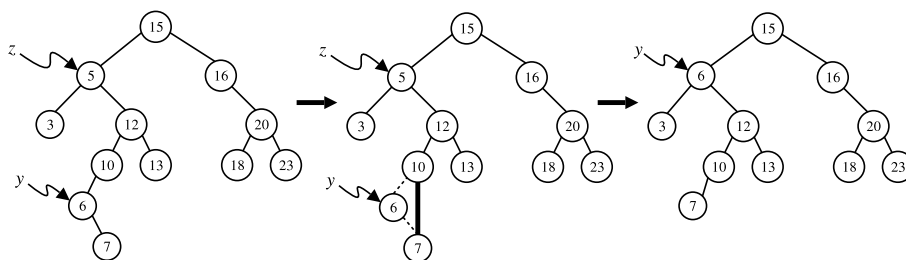


Figure 3: TREE-DELETE. If z has two children, we splice out its successor y , which has at most one child, and then replace z with y .

17 **return** y

An alternative approach to TREE-DELETE is to simply replace the deleted node with the maximum of its left subtree or the minimum of its right subtree. However, this may involve more steps than necessary.