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** (see Figure 1):

> 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 z is a node in the right subtree of x, then $key[x] \leq key[z]$.

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.

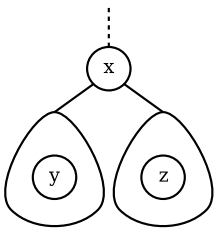

Figure 1: Binary Search Tree Property: $key[y] < key[x] \leq key[z]$

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2     then INORDER-TREE-WALK(left[x])
3          print key[x]  /* or some other operation on 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

2.1 Search

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-SEACH(left[x], k)
5     else  return TREE-SEACH(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
```

2.2 Minimum and Maximum

The propertities 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] ≠ NIL
2        do x ← left[x]
3  return x
```

```
TREE-MAXIMUM(x)
1  while right[x] ≠ NIL
2        do x ← right[x]
3  return x
```

2.3 Successor

The successor of a node x is the node with the next biggest key. Essentially, this is the minimum of all the nodes with equal or greater keys. If node x has a right sub-tree R, then its successor is the minimum of R. Otherwise, its successor must be the lowest ancestor of x whose left child is either an ancestor of x or x itself.

```
TREE-SUCCESSOR(x)
1  if right[x] ≠ NIL
2     then return TREE-MINIMUM(right[x])
3  y ← p[x]
4  while y ≠ NIL and x = right[y]
5        do x ← y
6           y ← p[y]
7  return y
```

3 Data Manipulation

3.1 Insert

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 of the current node is NIL. Node z is then inserted into the tree as the appropriate child of the current node.

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

3.2 Delete

TREE-DELETE deletes a node z from tree T. There are three possible states for node z prior to deletion. It may have no children, one child, or two children. The TREE-DELETE algorithm works by considering each of these possibilities and behaving accordingly. If z has no children, then it is simply removed from the tree. For the case where z has one child, $p[z]$'s left/right child pointer that

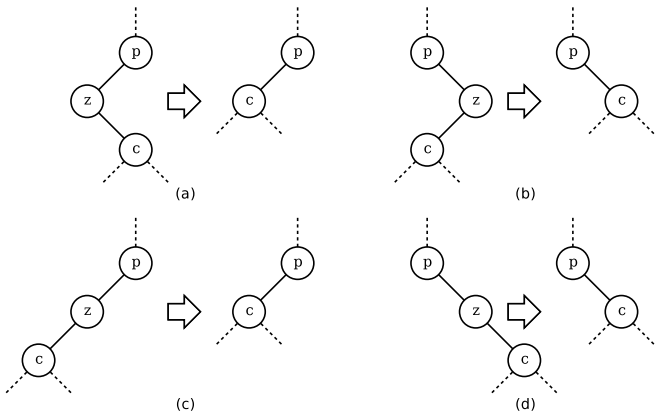

Figure 2: TREE-DELETE. If z has one child, it is simply spliced out of the tree. Since z can be the left or right child of its parent p, and it may have a left or right child c itself, there are four possibilities (a), (b), (c), and (d).

is currently directed at z is redirected to z's child (see Figure 2). Lastly, if z has two children, it's successor is removed from the tree and is used to replace z. The algorithm follows:

```
TREE-DELETE(T, z)
1   if left[z] = NIL or right[z] = NIL
2      then y ← z
3      else y ← TREE-SUCCESSOR(z)
4   if left[y] ≠ NIL
5      then x ← left[y]
6      else x ← right[y]
7   if x ≠ NIL
8      then p[x] ← p[y]
9   if p[y] = NIL
10     then root[T] ← x
11     else if y = left[p[y]]
12             then left[p[y]] ← x
13             else right[p[y]] ← x
14  if y ≠ z
15     then key[z] ← key[y]
16          copy y's satellite data into z
17  return y
```

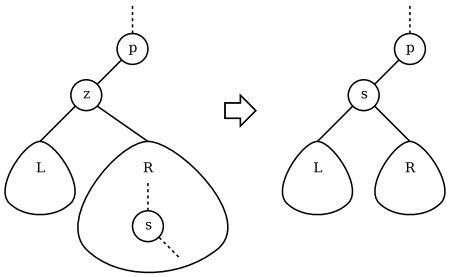

Figure 3: TREE-DELETE. If z has two children, we remove its successor s. Node s is guaranteed to have *at most* one child because it is the minimum of sub-tree R. It's removal is thus either of the other two TREE-DELETE cases. When this is done, s is used to replace z.