# Binary Search Trees

Manuel Serna-Aguilera

## Introduction

A **binary search tree** (or bst) is a more organized binary tree. In it each node contains a key value by which it can be searched, additional satellite data, and pointers *left*, *right*, and $p$ which point to the left child, right child, and parent node, respectively. The pointers may be populated, otherwise they will have the value NIL (coming form Latin, meaning *nothing*). The root is the top node from which all nodes are descended.

These keys are stored in such a way that they 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 $y.key \leq x.key$. If $y$ is a node in the right subtree of x, then $y.key \geq x.key$.

Three ways to walk, or traverse, binary search trees:

**Inorder tree walk**: traverse left subtree, visit current node, and then traverse right subtree.

This prints out all keys in sorted order, and traversing every node takes $\Theta(n)$ time when starting from the root.

INORDER-TREE-WALK($x$)
**if** $x \neq$ NIL
    INORDER-TREE-WALK($x.left$)
    print $x.key$
    INORDER-TREE-WALK($x.right$)

**Preorder tree walk**: visit current node, traverse left subtree, and traverse right subtree.

PREORDER-TREE-WALK($x$)
**if** $x \neq$ NIL
    print $x.key$
    PREORDER-TREE-WALK($x.left$)
    PREORDER-TREE-WALK($x.right$)

**Postorder tree walk**: traverse left subtree, traverse right subtree, and visit current node.

```
POSTORDER-TREE-WALK(x)
if x ≠ NIL
        POSTORDER-TREE-WALK(x.left)
        POSTORDER-TREE-WALK(x.right)
        print x.key
```

## Querying a binary search tree

Each of the following operations will run in $O(h)$ time for a bst of height $h$. The procedures discussed are, SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, and SUCCESSOR.

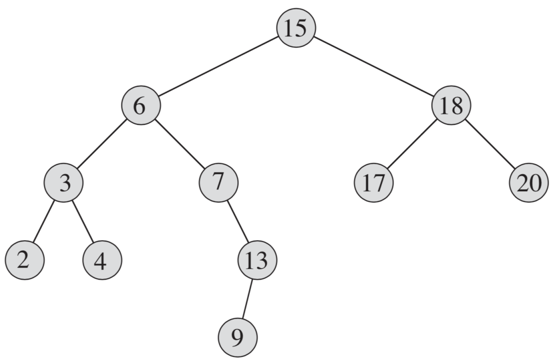


Figure 1: Referring to the above binary search tree, here are example cases of queries. **i.** To **search** for the lucky number 7, we always start searching at the root (15 at the top). Compare k=7 with the key of the root, $7 < 15$, so go down the *left* subtree. Now compare k=7 with 6, $7 > 6$, so go down the *right* subtree. Now compare k=7 with 7, $7 = 7$, duh, so in this simple procedure we return the value 7. **ii.** The **minimum** key of this bst is 2, you simply follow the *left* pointers. **iii.** The **maximum** key of this bst is 20, you simply follow the *right* pointers. **iv.** The **predecessor** of key 6 is 4, if the node has a left subtree, find the maximum of that left subtree. The predecessor of 9 is 7, if no left subtree exists, simply find the parent of the first node that is a right child (in this case 13 is 7's right child). **v.** The **successor** of 15 is 17, if the node has a right subtree, find the minimum of that subtree. The successor of 13 is 15, if no right subtree exists, simply find the parent of the first node that is a left child (in this case 6 is the right child of 15).

### Search

The binary search procedure is straightforward, given a key $k$, we must search the bst for a node with a key that matches $k$. We always start at the root, if $k$ is less than $x.key$, then we must go down the left subtree, if $k$ is larger than $x.key$, then we must go down the right subtree, if $k$ is equal to $x.key$, then we return a pointer to the node $x$. We repeat this process until we find the matching key, otherwise we return NIL.

```
TREE-SEARCH(x, k)
if x == NIL or k == x.key
    return x
if k < x.key
    return TREE-SEARCH(x.left, k)
else
    return TREE-SEARCH(x.right, k)
```

Here is the iterative version, which is more efficient on computers.

```
ITERATIVE-TREE-SEARCH(x, k)
while x ≠ NIL and k ≠ x.key
    if k < x.key
        x = x.left
    else
        x = x.right
return x
```

### Minimum and maximum

Finding the minimum simply means going left on the bst until a NIL value on the *left* field is encountered.

```
TREE-MINIMUM(x)
while x.left ≠ NIL
    x = x.left
return x
```

Symmetrically, the right operates in the same manner, but we go all right instead.

```
TREE-MAXIMUM(x)
while x.right ≠ NIL
    x = x.right
return x
```

### Predecessor and successor

Sometimes we want to find the node, based on the inorder tree walk, that comes before or after node $x$. The ***predecessor*** of a node $x$ is the node with the *largest* key smaller than $x.key$.

```
TREE-PREDECESSOR(x)
if x.left ≠ NIL
    return TREE-MAXIMUM(x.left)
y = x.p
while y ≠ NIL and x == y.left
    x = y
    y = y.p
return y
```

We have two cases to consider when searching for a predecessor.

1. If $x$ has a left subtree, find the maximum of that subtree.
2. If no left subtree exists, find the parent of the first node that is a right child.

The ***succcessor*** of a node $x$ is the node with the *smallest* key larger than $x.key$.

```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
    return TREE-MINIMUM(x.right)
y = x.p
while y ≠ NIL and x == y.right
    x = y
    y = y.p
return y
```

We have two cases to consider when searching for a successor, which is very much symmetrical to predecessor.

1. If $x$ has a right subtree, find the minimum of that subtree.
2. If no right subtree exists, find the parent of the first node that is a left child.

A good way to remember these procedures is to perform the reverse operations after you think you got the right node.

# Insertion

Inserting a new node into a binary search tree $T$ starts with a binary search in order to find where to insert node $z$, this can be to the left or right of some parent node $y$, or the existing tree may be empty! When we insert $z$, $z.key$ is some value $v$, while $z.left$ and $z.right$ will be NIL. The following procedure will take $O(h)$ time for a tree $T$ of height $h$.

```
TREE-INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else
        x = x.right
z.p = y
if y == NIL
    T.root = z // tree T was empty
elseif z.key < y.key
    y.left = z
else
    y.right = z
```

# Deletion

Deleting a node $z$ from a bst $T$ involves three cases, we will reference figure 12.4 from the book.

- Node $z$ has no children, remove it from the tree. In terms of pointers, modify the parent so that it refers not to $z$ but NIL instead (which is the value of $z$'s children).
- Node $z$ has one child, replace $z$ with its non-NIL child. Modify the pointers of the parent and non-NIL child of $z$ to point "around" it (part (a) and (b) where $l$ is $z$'s left child and $r$ is the right).
- Node $z$ has two children, this one is a tad tricky. We replace $z$ with either its predecessor or its successor, but we will go with the successor. There are two cases with this one.
  - Right child $r$ is $z$'s successor, in which case we just replace $z$ with $r$.
  - Right child $r$ is not $z$'s successor, in which case we must find the minimum of the right subtree.

To move subtrees around, the book defines a neat subroutine called TRANSPLANT. This TRANSPLANT procedure replaces the subtree rooted at $u$ with another subtree rooted at a different node $v$, and $u$'s parent becomes $v$'s parent. The pointers to the children of $v$ are not modified, so keep that in mind.

```
TRANSPLANT(T, u, v)
if u.p == NIL
    T.root = v
elseif u == u.p.left
    u.p.left = v
else
    u.p.right = v
if v ≠ NIL
    v.p = u.p
```

Having TRANSPLANT defined makes deleting much shorter.

```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else
    y = TREE-MINIMUM(z.right)
    if y.p ≠ z
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
```

The TREE-INSERT and TREE-DELETE procedure take $O(h)$ time for a bst of height $h$. For insertion, we must perform binary search, which takes $O(h)$ time while rearranging pointers takes constant time. For deletion, everything but TREE-MINIMUM takes constant time, finding the minimum takes $O(h)$ time again. Figure 12.4 from the book gives a good visualization for the delete cases.

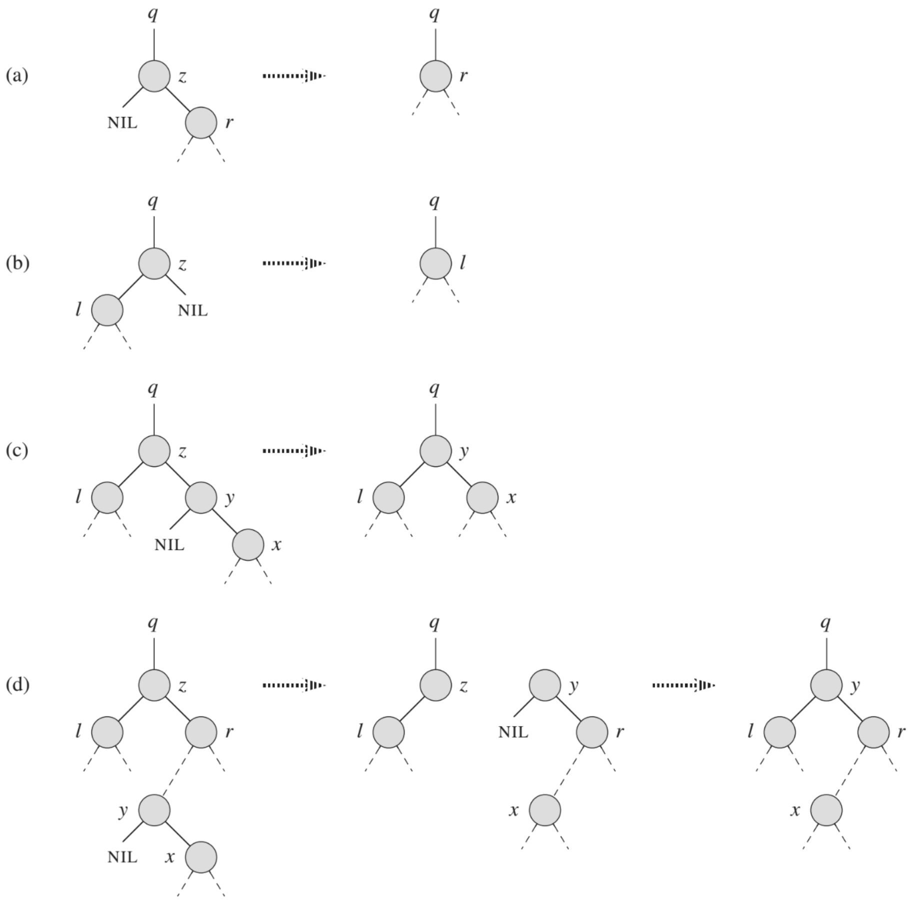


Figure 2: A visualization of the delete cases. **(a)** shows the single-child case for a right child $r$, and **(b)** for a left child $l$. **(c)** Two children case, $y$ is $z$'s successor, so replace $z$ with $y$. **(d)** Two children case where $z$'s successor is not its right child, so we must find the smallest key larger than $z.key$–the minimum of the right subtree.