Binary Search Trees

A **binary search tree** is a binary tree with a special property called the **BST-property**, which is given as follows:

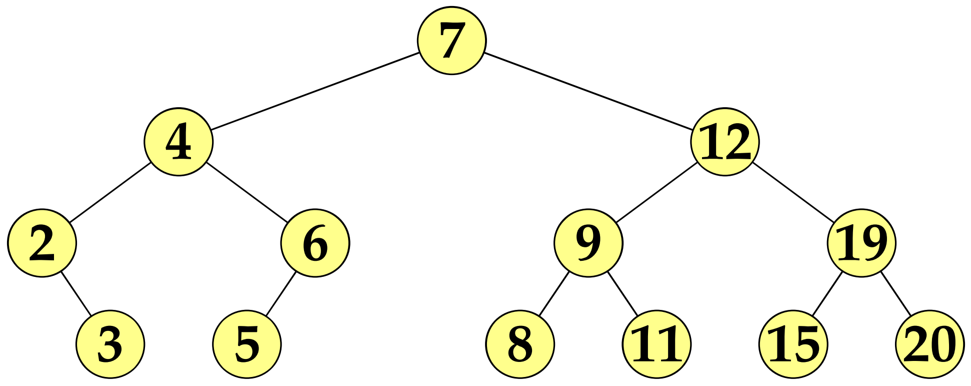
- ⋆ *For all nodes x and y, if y belongs to the left subtree of x, then the key at y is less than the key at x, and if y belongs to the right subtree of x, then the key at y is greater than the key at x.*

We will assume that the keys of a BST are pairwise distinct.

Each node has the following attributes:

- p, left, and right, which are pointers to the parent, the left child, and the right child, respectively, and
- key, which is key stored at the node.

An example



Traversal of the Nodes in a BST

By “traversal” we mean visiting all the nodes in a graph. Traversal strategies can be specified by the ordering of the three objects to visit: the current node, the left subtree, and the right subtree. We assume the the left subtree always comes before the right subtree. Then there are three strategies.

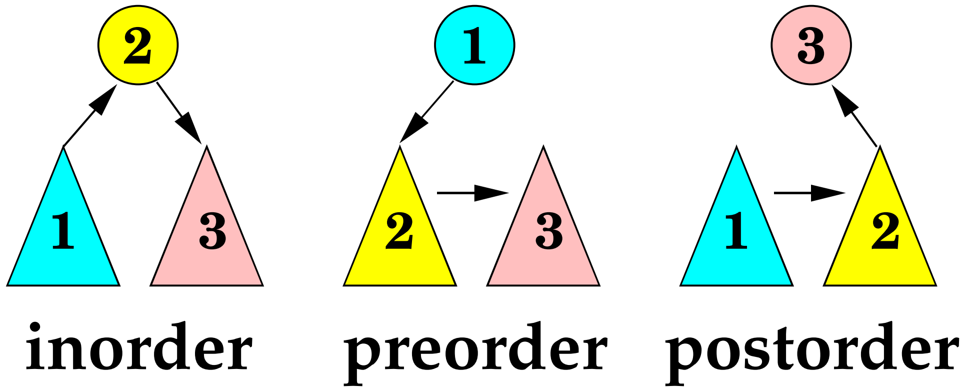
1. Inorder. The ordering is: the left subtree, the current node, the right subtree.
2. Preorder. The ordering is: the current node, the left subtree, the right subtree.
3. Postorder. The ordering is: the left subtree, the right subtree, the current node.

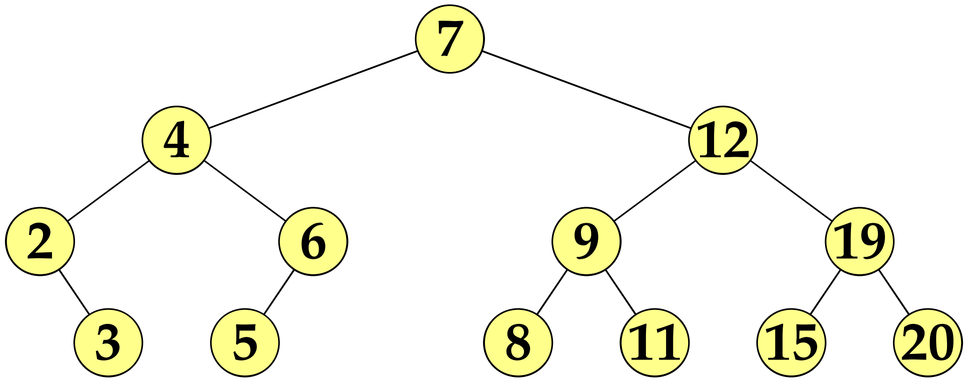
Inorder Traversal Pseudocode

This recursive algorithm takes as the input a pointer to a tree and executed inorder traversal on the tree. While doing traversal it prints out the key of each node that is visited.

```
**Inorder-Walk**($x$)
1: if $x = \mathbf{nil}$ then return
2: Inorder-Walk($left[x]$)
3: Print $key[x]$
4: Inorder-Walk($right[x]$)
```

We can write a similar pseudocode for preorder and postorder.





What is the outcome of inorder traversal on this BST? How about postorder traversal and preorder traversal?

Inorder traversal gives: 2, 3, 4, 5, 6, 7, 8 , 9, 11, 12, 15, 19, 20.

Preorder traversal gives: 7, 4, 2, 3, 6, 5, 12, 9, 8, 11, 19, 15, 20.

Postorder traversal gives: 3, 2, 5, 6, 4, 8, 11, 9, 15, 20, 19, 12, 7.

So, inorder travel on a BST finds the keys in nondecreasing order!

Operations on BST

1. Searching for a key

We assume that a key and the subtree in which the key is searched for are given as an input. We'll take the full advantage of the BST-property.

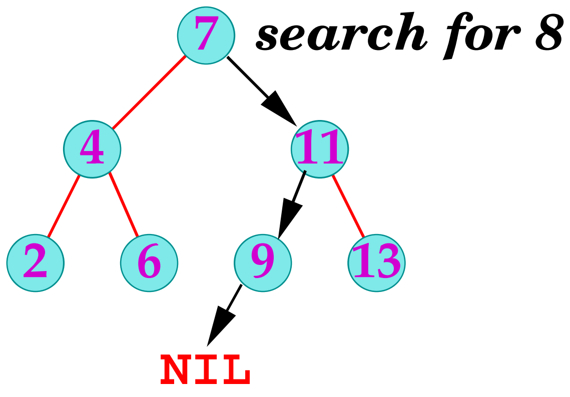
Suppose we are at a node. If the node has the key that is being searched for, then the search is over. Otherwise, the key at the current node is either strictly smaller than the key that is searched for or strictly greater than the key that is searched for. If the former is the case, then by the BST property, all the keys in th left subtree are strictly less than the key that is searched for. That means that we do not need to search in the left subtree. Thus, we will examine only the right subtree. If the latter is the case, by symmetry we will examine only the right subtree.

Algorithm

Here k is the key that is searched for and x is the start node.

```
**BST-Search**$(x, k)$
1: $y \leftarrow x$
2: while $y \neq$ nil do
3:     if $key[y] = k$ then return $y$
4:     else if $key[y] < k$ then $y \leftarrow right[y]$
5:     else $y \leftarrow left[y]$
6: return ("NOT FOUND")
```

An Example



What is the running time of search?

2. The Maximum and the Minimum

To find the minimum identify the leftmost node, i.e. the farthest node you can reach by following only left branches.

To find the maximum identify the rightmost node, i.e. the farthest node you can reach by following only right branches.

```
BST-Minimum($x$)
  1: if $x = \mathbf{nil}$ then return ("Empty Tree")
  2: $y \leftarrow x$
  3: while $\mathit{left}[y] \neq \mathbf{nil}$ do $y \leftarrow \mathit{left}[y]$
  4: return $(\mathit{key}[y])$
```

```
BST-Maximum($x$)
  1: if $x = \mathbf{nil}$ then return ("Empty Tree")
  2: $y \leftarrow x$
  3: while $\mathit{right}[y] \neq \mathbf{nil}$ do $y \leftarrow \mathit{right}[y]$
  4: return $(\mathit{key}[y])$
```

3. Insertion

Suppose that we need to insert a node z such that $k = key[z]$. Using binary search we find a **nil** such that replacing it by z does not break the BST-property.

```
**BST-Insert**($x$, $z$, $k$)
 1: if $x = \mathrm{nil}$ then return "Error"
 2: $y \leftarrow x$
 3: while $\mathbf{true}$ do $\{$
 4:       if $key[y] < k$
 5:       then $z \leftarrow left[y]$
 6:       else $z \leftarrow right[y]$
 7:       if $z = \mathrm{nil}$ break
 8: $\}$
 9: if $key[y] > k$ then $left[y] \leftarrow z$
10: else $right[p[y]] \leftarrow z$
```

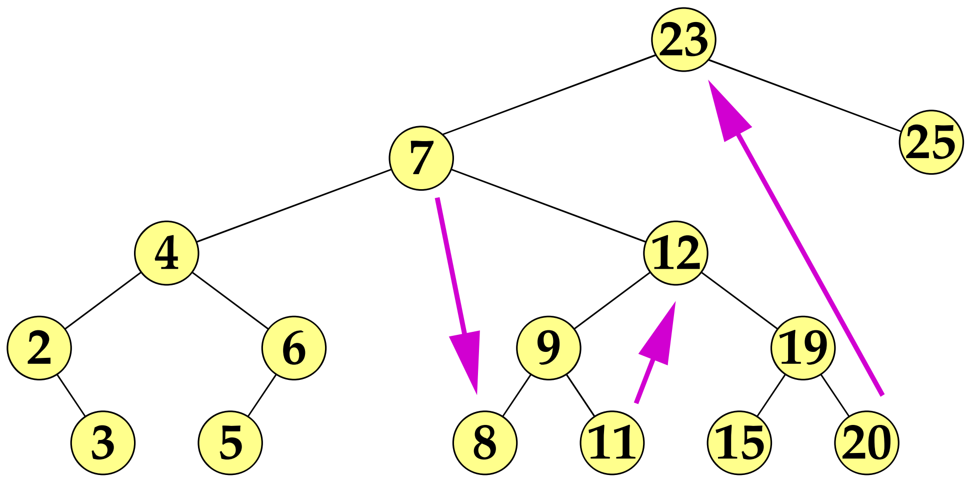
4. The Successor and The Predecessor

The successor (respectively, the predecessor) of a key k in a search tree is the smallest (respectively, the largest) key that belongs to the tree and that is strictly greater than (respectively, less than) k.

The idea for finding the successor of a given node x.

- If x has the right child, then the successor is the minimum in the right subtree of x.
- Otherwise, the successor is the parent of the farthest node that can be reached from x by following only right branches backward.

An Example



Algorithm

```
BST-Successor($x$)
1: if $right[x] \neq$ nil then
2: {   $y \leftarrow right[x]$
3:     while $left[y] \neq$ nil do $y \leftarrow left[y]$
4:     return ($y$) }
5: else
6: {   $y \leftarrow x$
7:     while $right[p[x]] = x$ do $y \leftarrow p[x]$
8:     if $p[x] \neq$ nil then return ($p[x]$)
9:     else return (“NO SUCCESSOR”) }
```

The predecessor can be found similarly with the roles of left and right exchanged and with the roles of maximum and minimum exchanged.

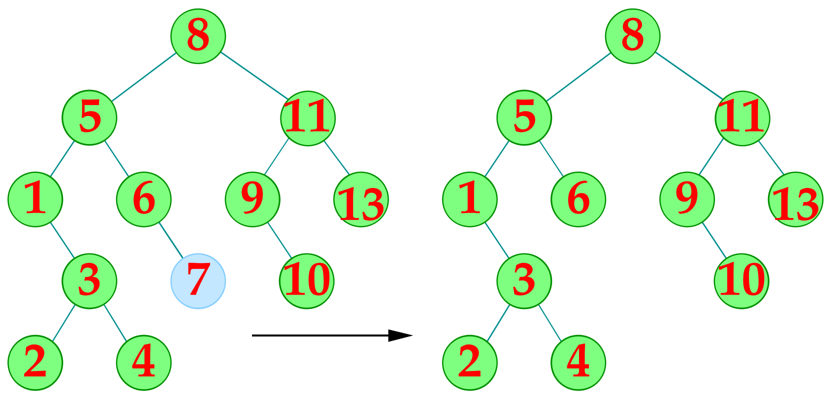
For which node is the successor undefined?

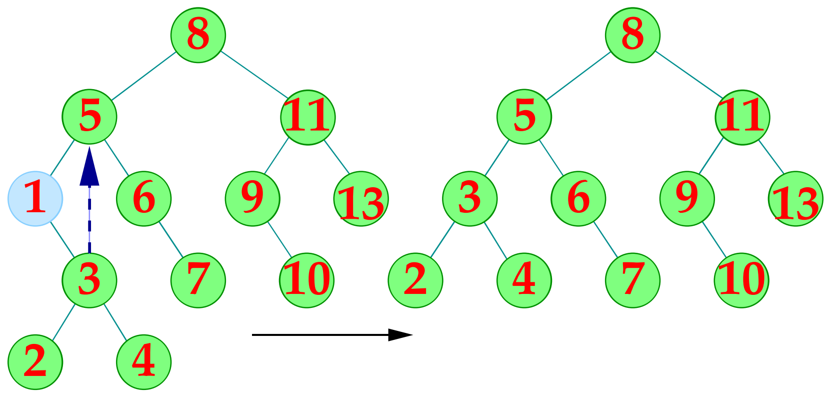
What is the running time of the successor algorithm?

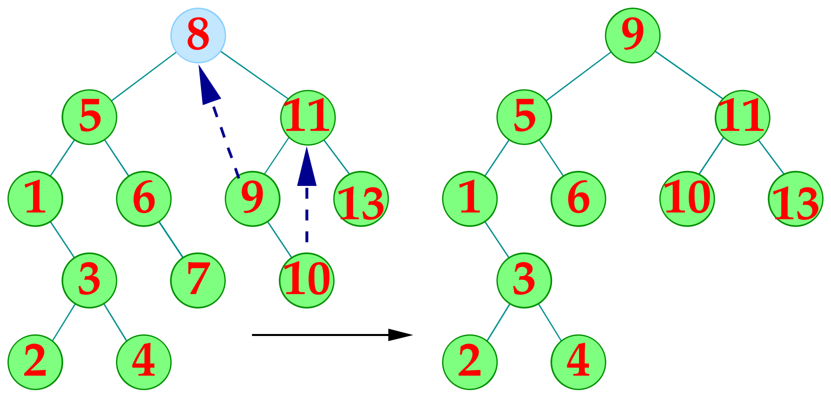
5. Deletion

Suppose we want to delete a node z.

1. If z has no children, then we will just replace z by $\mathbf{nil}$.
2. If z has only one child, then we will promote the unique child to z's place.
3. If z has two children, then we will identify z's successor. Call it y. The successor y either is a leaf or has only the right child. Promote y to z's place. Treat the loss of y using one of the above two solutions.


8
5
11
1
6
9
13
3
7
10
2
4
8
5
11
1
6
9
13
3
10
2
4


8
5
11
1
6
9
13
3
7
10
2
4
8
5
11
3
6
9
13
2
4
7
10


8
5
11
1
6
9
13
3
7
10
2
4
9
5
11
1
6
10
13
3
2
4

Algorithm

This algorithm deletes z from BST T.

```
BST-Delete($T$, $z$)
 1: if $left[z] = \textbf{nil}$ or $right[z] = \textbf{nil}$
 2: then $y \leftarrow z$
 3: else $y \leftarrow$ BST-Successor($z$)
 4: ▷ $y$ is the node that's actually removed.
 5: ▷ Here $y$ does not have two children.
 6: if $left[y] \neq \textbf{nil}$
 7: then $x \leftarrow left[y]$
 8: else $x \leftarrow right[y]$
 9: ▷ $x$ is the node that's moving to $y$'s position.
10: if $x \neq \textbf{nil}$ then $p[x] \leftarrow p[y]$
11: ▷ $p[x]$ is reset If $x$ isn't NIL.
12: ▷ Resetting is unnecessary if $x$ is NIL.
```

Algorithm (cont'd)

```
13: if $p[y] = \mathbf{nil}$ then $root[T] \leftarrow x$
14: ▷ If $y$ is the root, then $x$ becomes the root.
15: ▷ Otherwise, do the following.
16: else if $y = left[p[y]]$
17:     then $left[p[y]] \leftarrow x$
18: ▷ If $y$ is the left child of its parent, then
19: ▷ Set the parent's left child to $x$.
20:     else $right[p[y]] \leftarrow x$
21: ▷ If $y$ is the right child of its parent, then
22: ▷ Set the parent's right child to $x$.
23: if $y \neq z$ then
24: {   $key[z] \leftarrow key[y]$
25:     Move other data from $y$ to $z$ }
27: return $(y)$
```

BINARY SEARCH TREE PERFORMANCE

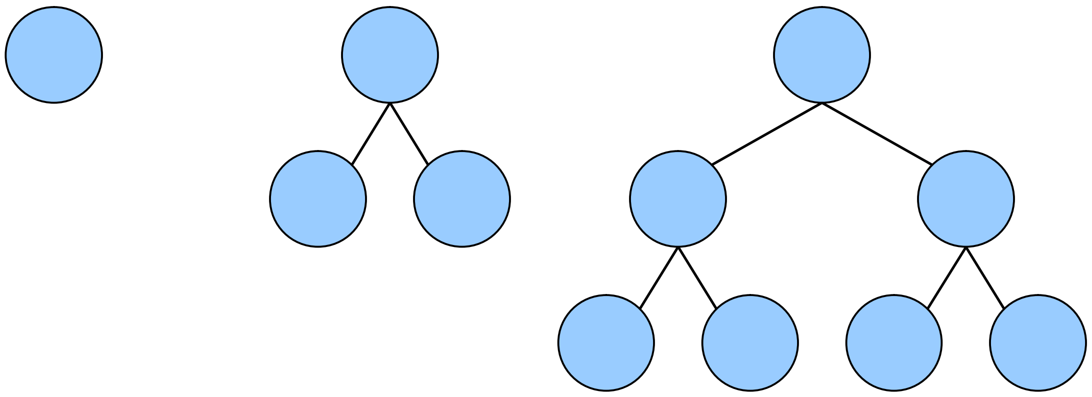
Operation	Best Time	Average Time	Worst Time
		(on a tree of n nodes)	
Find		??	
Insert	O(lg n)	O(lg n)	O(n)
Delete		??	

Fastest Running Time

The find, insert and delete algorithms start at the tree root and a follow path down to, at worst case, the leaf at the very lowest level. The total number of steps of these algorithms is, therefore, the largest level of the tree, which is called the ***depth*** of the tree.

The best (fastest) running time occurs when the binary search tree is ***full*** – in which case the search processes used by the algorithms perform like a binary search.

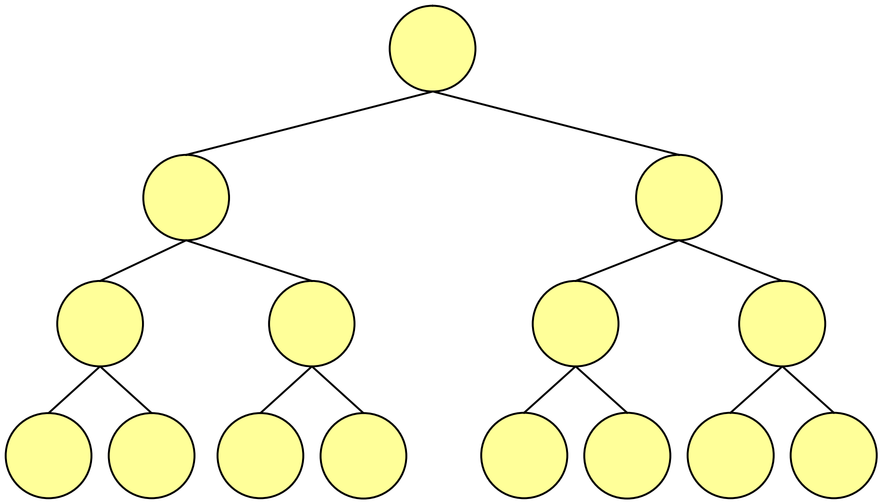
Let's verify this. A ***full binary tree*** is one in which nodes completely fill every level. For example, here are the unique full binary trees of which have tree depths of 0, 1 and 2:



As you can verify by looking at the examples, a full tree has 1 node at level 0, 2 nodes at level 1, 4 nodes at level 2 and so on. Thus, it will have 2^d nodes at level d. Adding these quantities, the total number of nodes n for a full binary tree with depth d is:

$$n = 2^0 + 2^1 + 2^2 + \ldots + 2^d = 2^{d+1} - 1$$

For example, the full binary tree of depth 2 above has $2^3 - 1 = 7$ nodes. The binary tree below is a full tree of depth 3 and has $2^4 - 1 = 15$ nodes.



Now, considering the formula above for the number of nodes in a full binary search tree:

$$n = 2^{d+1} - 1$$

Solving for *d*, we get:

$$\begin{aligned}
&n+1 = 2^{d+1} \\
&\lg(n+1) = \lg(2^{d+1}) \\
&\lg(n+1) = (d+1)\lg(2) \\
&\lg(n+1) = d+1 \\
&d = \lg(n+1) - 1 \\
&d = \lfloor \lg(n) \rfloor
\end{aligned}$$

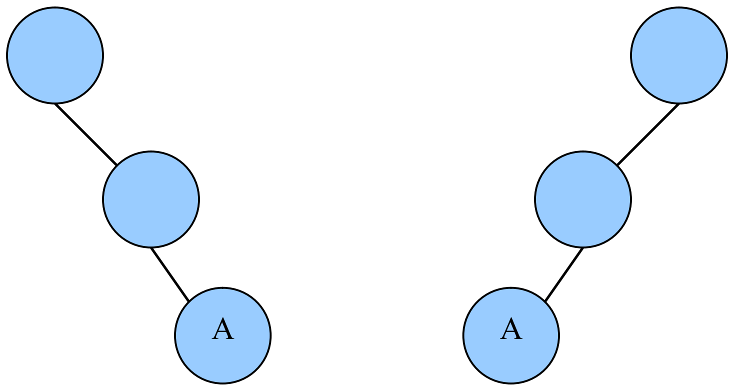
For example, the depth of a full binary search tree with 15 nodes is 3.

In other words, the depth of a binary search tree with *n* nodes can be no less than lg(*n*) and so the running time of the find, insert and delete algorithms can be no less than lg(*n*).

A full binary search tree is said to be ***balanced*** because every node's proper descendants are divided evenly between its left and right subtrees. Thus, the search algorithm employed by the find, insert and delete operations perform like a binary search.

Slowest Running Time

As a binary search tree becomes more and more unbalanced, the performance of the find, insert and delete algorithms degrades until reaching the worst case of O(*n*), where *n* is the number of nodes in the tree. For example, he number of comparisons needed to find the node marked *A* in the binary search trees below is 3, which is equal to the number of nodes.



Binary search trees, such as those above, in which the nodes are in order so that all links are to right children (or all are to left children), are called ***skewed trees***.

Average Running Time

The average running time of the binary search tree operations is difficult to establish because it is not clear that all binary search trees are equally likely.[1] It has been proven that when a binary search tree is constructed through a random sequence of insertions then the average depth of any node is O(lg *n*). For example, Figure 4.26[2] shows a 500-node randomly generated tree whose average node depth (calculated over all nodes) is 9.98.

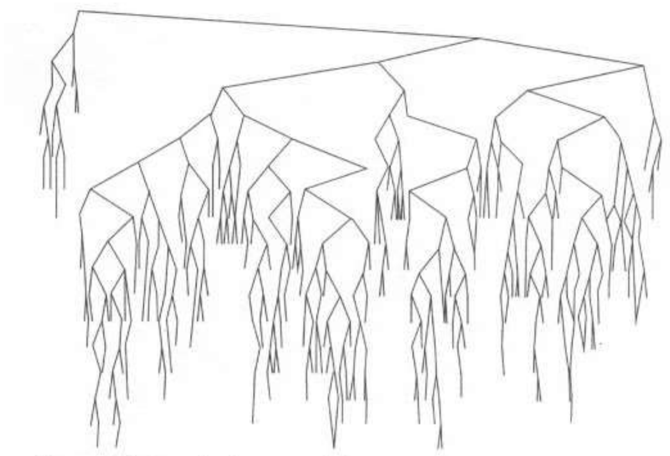


Figure 4.26 A randomly generated binary search tree

The problem arises when considering the effect of deletions since the deletion algorithm replaces a deleted node with one from its right subtree, thus favoring left imbalance. For example, Figure 4.27[3] shows the result of applying 250,000 random insert/remove operations to the tree of Figure 4.26. The tree still has 500 nodes but now has an average node depth of 12.51.

[1] Mark Allen Weiss, *Data Structures and Algorithm Analysis in Java*, Addison-Wesley, 1999, page 117.
[2] Ibid, page 117.
[3] Ibid, page 117.

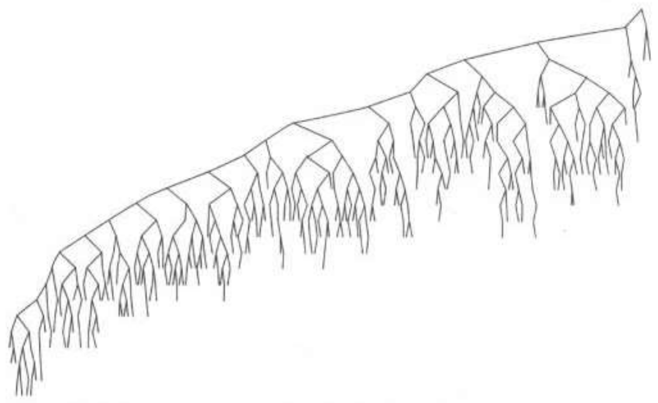


Figure 4.27 Binary search tree after $\Theta(N^2)$ `insert/remove` pairs

Extreme left imbalance after deletion, however, does not always happen nor does it happen with all trees, so its effect on the average execution times is not yet understood. In short, the average execution time of the binary search tree operations appears (but cannot be proven) to be O(lg *n*).[4]

[4] For a list of references on this topic, see Ibid, page 152.

Balanced versus Unbalanced Trees

The time complexities of operations find, insert and delete on a binary search tree is:

- At best O(lg *n*), which occur when the tree is full
- At worst O(*n*) which occur when the tree is skewed
- Thought to be on average O(lg *n*)

A ***balanced*** binary search tree is close to being full, although not necessarily completely full. It has, for each node, about the same number of nodes in its left subtree as in its right subtree. Thus, the find, insert and delete operations on a balanced tree give close to O(lg *n*) performance. The more unbalanced the tree becomes, the longer the search time grows until, at worst, it is O(*n*), occurring when the tree is skewed.

If you can live with running times that average O(lg *n*) but may degenerate to O(*n*) in the worst case, then the binary search tree is sufficient for your purposes. If, however, you must have O(lg *n*) as your worst-case running time, then you must use a balanced binary search tree.

[5] Mark Allen Weiss, *Data Structures and Algorithm Analysis in Java*, Addison-Wesley, 1999, pages 118-138.
[6] Ibid, pages pages 118-138.