Chapter 12: Binary Search Trees

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

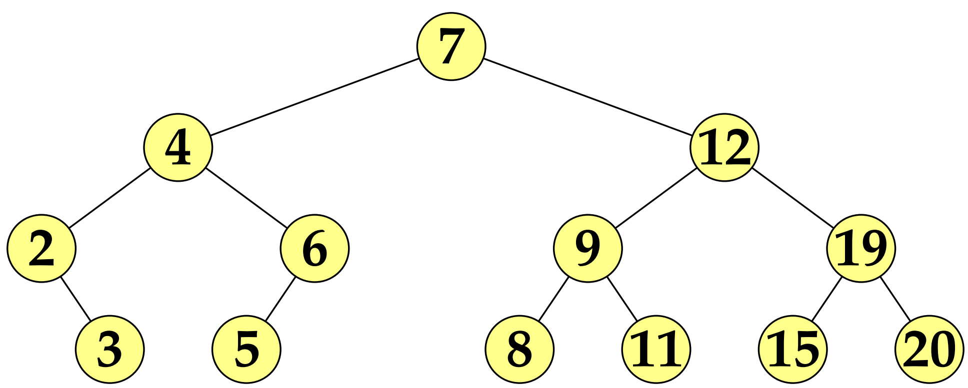
⋆ *For all nodes x and y, if y belongs to the left subtree of x, then the key at y is less than the key at x, and if y belongs to the right subtree of x, then the key at y is greater than the key at x.*

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.

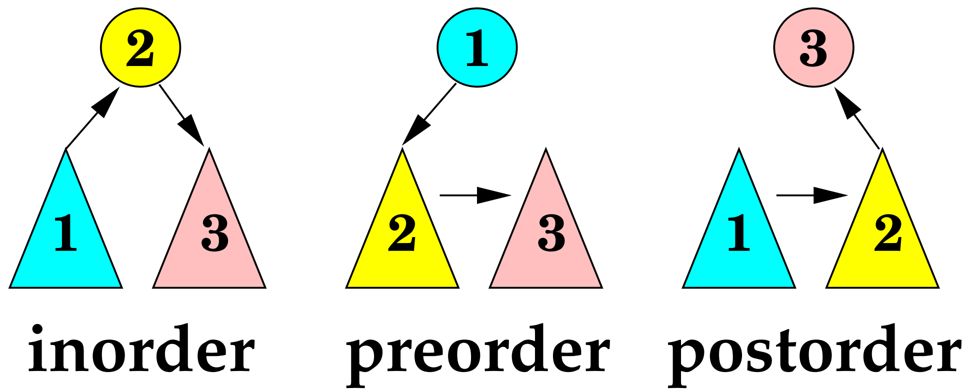
Inorder Traversal Pseudocode

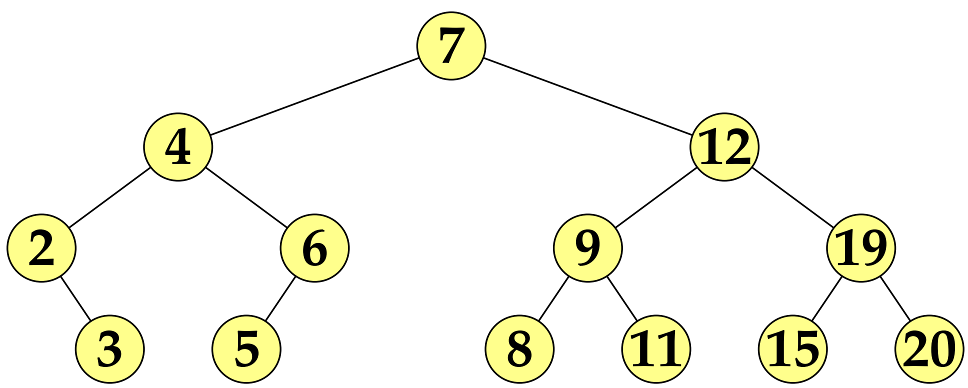
This recursive algorithm takes as the input a pointer to a tree and executed inorder traversal on the tree. While doing traversal it prints out the key of each node that is visited.

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.

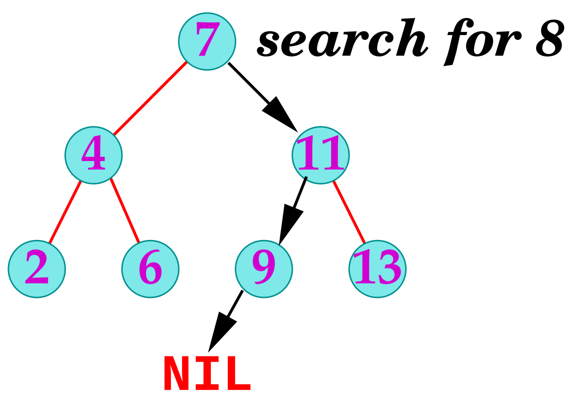
Algorithm

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

BST-Search(x, k)

```
y ← x
while y ≠ nil do
    if key[y] = k then return y
    else if key[y] < k then y ← right[y]
    else y ← left[y]
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)

```
if x = nil then return "Error"
y ← x
while true do {
    if key[y] < k
    then z ← left[y]
    else z ← right[y]
    if z = nil break
}
if key[y] > k then left[y] ← z
else right[p[y]] ← z
```

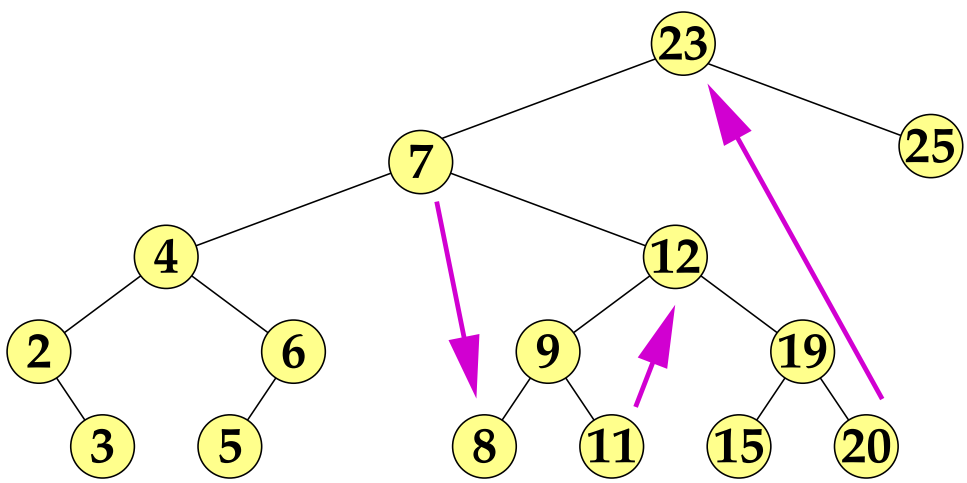
4. The Successor and The Predecessor

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)

```
if right[x] ≠ nil then
{   y ← right[x]
    while left[y] ≠ nil do y ← left[y]
    return (y) }
else
{   y ← x
    while right[p[x]] = x do y ← p[x]
    if p[x] ≠ nil then return (p[x])
    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.

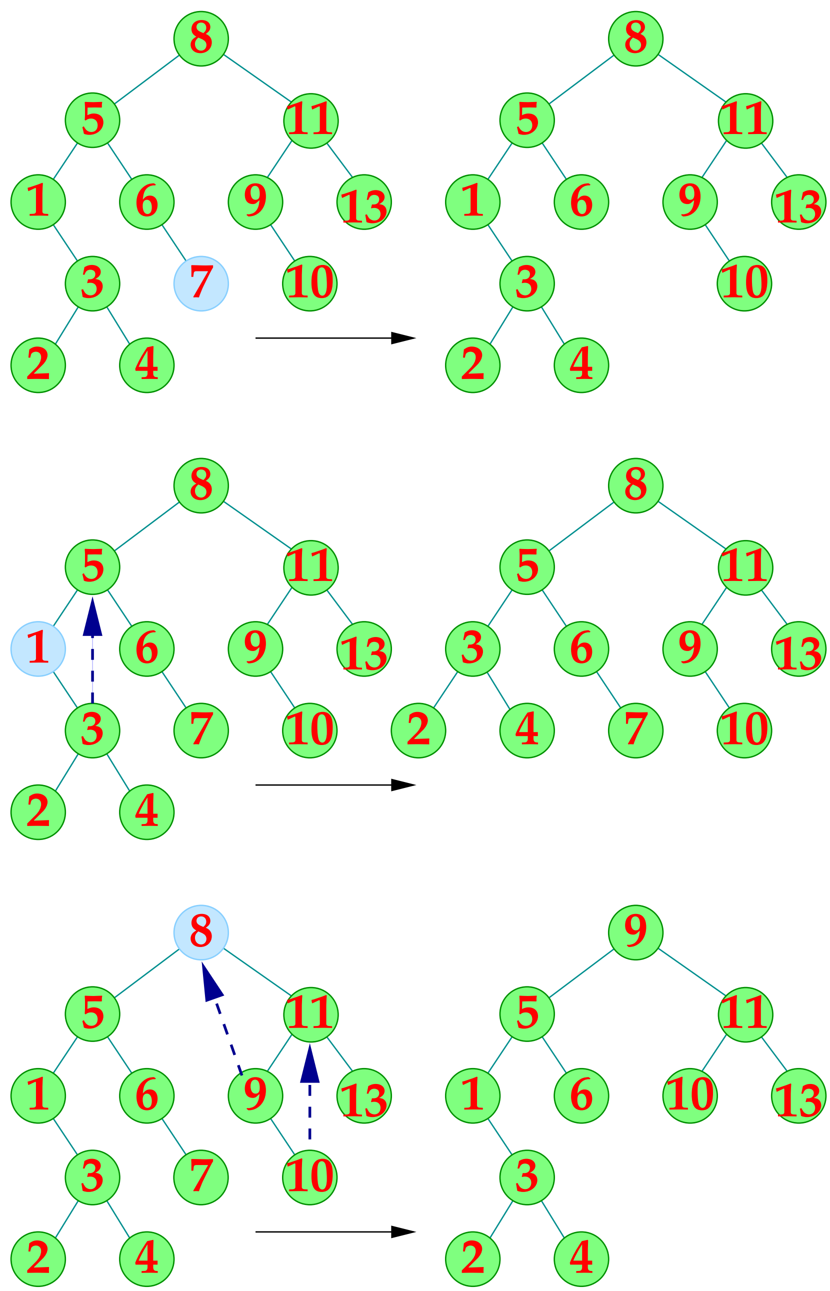
For which node is the successor undefined?

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)

if $\mathit{left}[z] = \textbf{nil}$ **or** $\mathit{right}[z] = \textbf{nil}$
then $y \leftarrow z$
else $y \leftarrow$ **BST-Successor**(z)
$\triangleright$ y is the node that's actually removed.
$\triangleright$ Here y does not have two children.
if $\mathit{left}[y] \neq \textbf{nil}$
then $x \leftarrow \mathit{left}[y]$
else $x \leftarrow \mathit{right}[y]$
$\triangleright$ x is the node that's moving to y's position.
if $x \neq \textbf{nil}$ **then** $p[x] \leftarrow p[y]$
$\triangleright$ $p[x]$ is reset If x isn't NIL.
$\triangleright$ Resetting is unnecessary if x is NIL.

Algorithm (cont'd)

if $p[y] = \mathbf{nil}$ **then** $root[T] \leftarrow x$
$\triangleright$ If y is the root, then x becomes the root.
$\triangleright$ Otherwise, do the following.
else if $y = left[p[y]]$
    then $left[p[y]] \leftarrow x$
$\triangleright$ If y is the left child of its parent, then
$\triangleright$ Set the parent's left child to x.
    else $right[p[y]] \leftarrow x$
$\triangleright$ If y is the right child of its parent, then
$\triangleright$ Set the parent's right child to x.
if $y \neq z$ **then**
$\{$ $key[z] \leftarrow key[y]$
    Move other data from y to z $\}$
return (y)

Summary of Efficiency Analysis

Theorem A On a binary search tree of height h, **Search**, **Minimum**, **Maximum**, **Successor**, **Predecessor**, **Insert**, and **Delete** can be made to run in $O(h)$ time.

Randomly built BST

Suppose that we insert n distinct keys into an initially empty tree. Assuming that the n! permutations are equally likely to occur, what is the average height of the tree?

To study this question we consider the process of constructing a tree T by **inserting in order randomly selected n distinct keys** to an initially empty tree. Here the actually values of the keys do not matter. What matters is the position of the inserted key in the n keys.

The Process of Construction

So, we will view the process as follows:

A key x from the keys is selected uniformly at random and is inserted to the tree. Then all the other keys are inserted. Here all the keys greater than x go into the right subtree of x and all the keys smaller than x go into the left subtree. Thus, the height of the tree thus constructed is one plus the larger of the height of the left subtree and the height of the right subtree.

Random Variables

$n =$ number of keys

$X_n =$ height of the tree of n keys

$Y_n = 2^{X_n}$.

We want an upper bound on $E[Y_n]$.

For $n \geq 2$, we have

$$E[Y_n] = \frac{1}{n}\left(\sum_{i=1}^{n} 2E[\max\{Y_{i-1}, Y_{n-i}\}]\right).$$

$$\begin{aligned} E[\max\{Y_{i-1}, Y_{n-i}\}] &\leq E[Y_{i-1} + Y_{n-i}] \\ &\leq E[Y_{i-1}] + E[Y_{n-i}] \end{aligned}$$

Collecting terms:

$$E[Y_n] \leq \frac{4}{n}\sum_{i=1}^{n-1} E[Y_i].$$

Analysis

We claim that for all $n \geq 1$ $E[Y_n] \leq \frac{1}{4}\binom{n+3}{3}$.
We prove this by induction on n.

Base case: $E[Y_1] = 2^0 = 1$.

Induction step: We have

$$E[Y_n] \leq \frac{4}{n} \sum_{i=1}^{n-1} E[Y_i]$$

Using the fact that

$$\sum_{i=0}^{n-1} \binom{i+3}{3} = \binom{n+3}{4}$$

$$E[Y_n] \leq \frac{4}{n} \cdot \frac{1}{4} \cdot \binom{n+3}{4}$$

$$E[Y_n] \leq \frac{1}{4} \cdot \binom{n+3}{3}$$

Jensen's inequality

A function f is **convex** if for all x and y, $x < y$, and for all λ, $0 \leq \lambda \leq 1$,

$$f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda)f(y)$$

Jensen's inequality states that for all random variables X and for all convex function f

$$f(E[X]) \leq E[f(X)].$$

Let this X be X_n and $f(x) = 2^x$. Then $E[f(X)] = E[Y_n]$. So, we have

$$2^{E[X_n]} \leq \frac{1}{4}\binom{n+3}{3}.$$

The right-hand side is at most $(n+3)^3$. By taking the log of both sides, we have

$$E[X_n] = O(\log n).$$

Thus the average height of a randomly build BST is $O(\log n)$.