Binary Search Tree (Part 1)

Yufei Tao

Department of Computer Science and Engineering
Chinese University of Hong Kong

This and the next lecture will be devoted to the most powerful data structure of this course: the binary search tree (BST). This is without a doubt one of the most important data structures in computer science.

In this lecture, we will focus on the static version of the BST (namely, without considering insertions and deletions), leaving the dynamic version to the next lecture.

We will discuss the BST on a specific problem:

Dynamic Predecessor Search

Let S be a set of integers. We want to store S in a data structure to support the following operations:

- A predecessor query: give an integer q, find its predecessor in S, which is the largest integer in S that does not exceed q;
- Insertion: adds a new integer to S;
- Deletion: removes an integer from S.

Example

Suppose that $S = \{3, 10, 15, 20, 30, 40, 60, 73, 80\}$.

- The predecessor of 23 is 20
- The predecessor of 15 is 15
- The predecessor of 2 does not exist.

Note that a predecessor query is more general (why?) than a "dictionary lookup". Recall that, given a value q, a dictionary lookup determines whether $q \in S$.

We will learn a version of the BST that guarantees:

- $O(n)$ space consumption.
- $O(\log n)$ time per predecessor query (hence, also per dictionary lookup).
- $O(\log n)$ time per insertion
- $O(\log n)$ time per deletion

where $n = |S|$. Note that all the above complexities hold in the worst case.

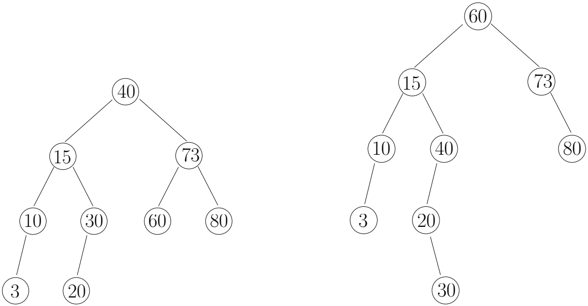
Binary Search Tree (BST)

A BST on a set S of n integers is a binary tree T satisfying all the following requirements:

- T has n nodes.
- Each node u in T stores a distinct integer in S, which is called the key of u.
- For every internal u, it holds that:
  - The key of u is larger than all the keys in the left subtree of u.
  - The key of u is smaller than all the keys in the right subtree of u.

Example

Two possible BSTs on $S = \{3, 10, 15, 20, 30, 40, 60, 73, 80\}$.



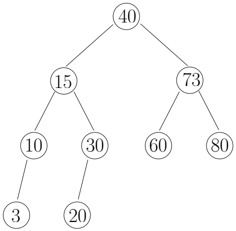
Balanced Binary Tree

A binary tree T is balanced if the following holds on every internal node u of T:

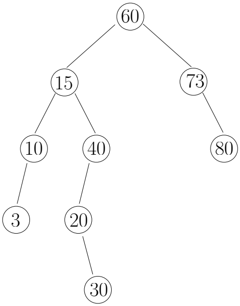
- The height of the left subtree of u differs from that of the right subtree of u by at most 1.

If u violates the above requirement, we say that u is imbalanced.

Example



Balanced



Not balanced (nodes 40 and 60 are imbalanced)

Height of a Balanced Binary Tree

Theorem: A balanced binary tree with n nodes has height $O(\log n)$.

Proof: Denote the height as h. We will show that a balanced binary tree with height h must have $\Omega(2^{h/2})$ nodes.

Once this is done, it will then follow that there is a constant $c > 0$ such that:

$$\begin{array}{rrcl} & n & \geq & c \cdot 2^{h/2} \\ \Rightarrow & 2^{h/2} & \leq & n/c \\ \Rightarrow & h/2 & \leq & \log_2(n/c) \\ \Rightarrow & h & = & O(\log n). \end{array}$$

Height of a Balanced Binary Tree

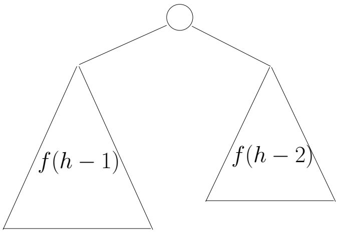
Let $f(h)$ be the minimum number of nodes in a balanced binary tree with height h. It is clear that:

$$f(1) = 1$$
$$f(2) = 2$$

Height of a Balanced Binary Tree

In general, for $h \geq 3$:

$$f(h) = 1 + f(h-1) + f(h-2)$$



Height of a Balanced Binary Tree

When h is an even number:

$$\begin{aligned} f(h) &= 1 + f(h-1) + f(h-2) \\ &> 2 \cdot f(h-2) \\ &> 2^2 \cdot f(h-4) \\ &\ldots \\ &> 2^{h/2-1} \cdot f(2) \\ &= 2^{h/2} \end{aligned}$$

Height of a Balanced Binary Tree

When h an odd number (i.e., $h \geq 3$):

$$\begin{aligned} f(h) &> f(h-1) \\ &> 2^{(h-1)/2} \\ &= 2^{h/2}/\sqrt{2} \\ &= \Omega(2^{h/2}) \end{aligned}$$

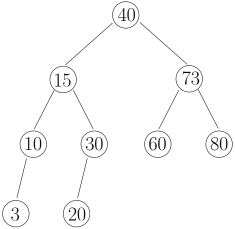
□

Predecessor Query

Suppose that we have created a balanced BST T on a set S of n integers. A predecessor query with search value q can be answered by descending a single root-to-leaf path:

1. Set $p \leftarrow -\infty$ (p will contain the final answer at the end)
2. Set $u \leftarrow$ the root of T
3. If $u = \texttt{nil}$, then return p
4. If key of $u = q$, then set p to q, and return p
5. If key of $u > q$, then set u to the left child (now $u = \texttt{nil}$ if there is no left child), and repeat from Line 3.
6. Otherwise, set p to the key of u, u to the right child, and repeat from Line 3.

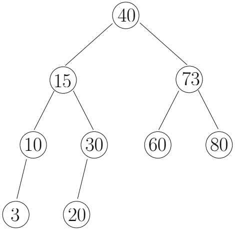
Example



Suppose that we want to find the predecessor of 35.

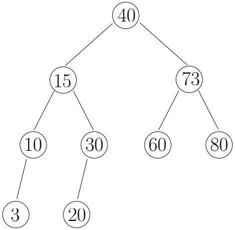
Start from u = root 40. Since $40 > 35$, the predecessor cannot be in the right subtree of 40. So we move to the left child of 40. Now u = node 15.

Example



Since 15 $<$ 35, the predecessor cannot be in the left subtree of 15. Update p to 15, because this is the predecessor of 35 so far, if we do not consider the right subtree of 15. Now, move u to the right child, namely, node 30.

Example



Since $30 < 35$, the predecessor cannot be in the left subtree of 30. Update p to 30. We need to move to the right child, but 30 does not have a right child. So the algorithm terminates here with $p = 30$ as the final answer.

Analysis of Predecessor Query Time

Obviously, we spend $O(1)$ time at each node visited. Since the BST is balanced, we know that its height is $O(\log n)$.

Therefore, the total query time is $O(\log n)$.

Successors

The opposite of predecessors are successors.

The successor of an integer q in S is the smallest integer in S that is no smaller than q.

Suppose that $S = \{3, 10, 15, 20, 30, 40, 60, 73, 80\}$.

- The successor of 23 is 30
- The successor of 15 is 15
- The successor of 81 does not exist.

Finding a Successor

Given an integer q, a successor query returns the successor of q in S.

By symmetry, we know from the earlier discussion (on predecessor queries) that a predecessor query can be answered using a balanced BST in $O(\log n)$ time, where $n = |S|$.