CS120: Intro. to Algorithms and their Limitations Prof. Salil Vadhan

Lecture 5: Binary Search Trees

Harvard SEAS - Fall 2021 *Sept. 16, 2021*

1 Announcements

Recommended Reading:

- Roughgarden II Sections 11.2–11.3.7
- CLRS Sections 12.0–12.3
- Salil OH after class in 3.327
- PS2 posted
- Active Learning next Tuesday

2 Binary Search Trees

We were looking for a dynamic data structure to solve the following problem:

Updates: Insert an item-key pair (I, K) with $K \in \mathbb{R}$
Queries : Given a threshold $q \in \mathbb{R}$, return (I_i, K_i) such that $K_i = \max\{K_j : K_j \leq q, j \in [n]\}$, where $(I_0, K_0), (I_1, K_1), \ldots, (I_{n-1}, K_{n-1})$ is the sequence of all prior insertions, or $\perp$ if $q < \min\{K_j : j \in [n]\}$

Data-Structure Problem Dynamic Predecessors

Note that we've dropped the initial input array x from the formulation, since any array x of n item-key pairs can be constructed by a sequence of n insertions. We want both updates (insertions) and (predecessor) queries to be very efficient. This led us to:

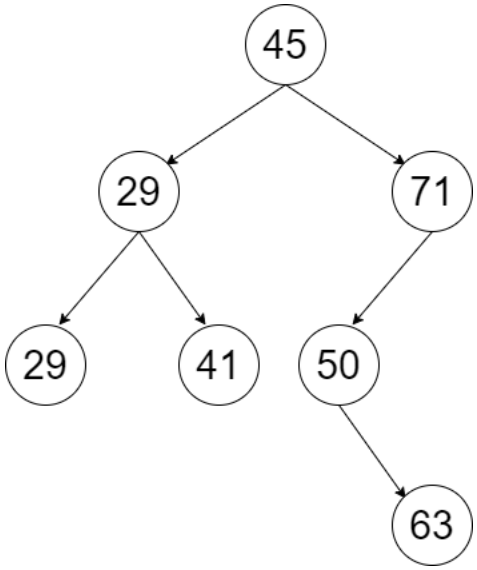
Definition 2.1. A *binary search tree (BST)* is a recursive data structure where every node v has:

- an item I_v
- a key K_v
- a pointer to a left child v_L (or `None`)
- a pointer to a right child v_R (or `None`)

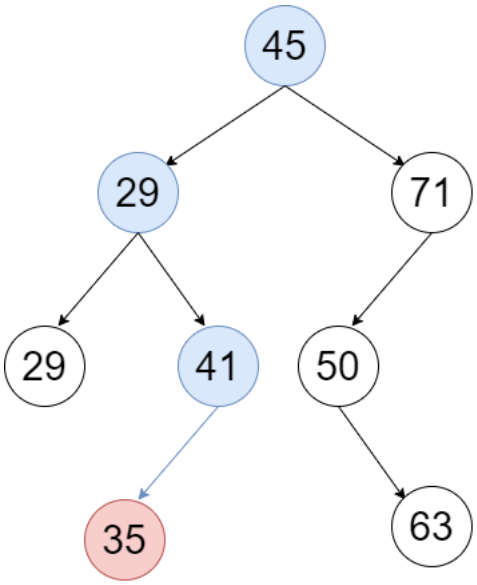
Crucially, we also require that the keys satisfy the *BST Property*:

> If v has a left-child v_L, then the keys of v_L and all its descendants are no larger than K_v, and similarly, if v has a right-child, then the keys of v_R and all of its descendants are no smaller than K_v.

Example:



If we need to insert 35, then we color the nodes that we compare 35 to:



We remark that the description of binary search trees in Roughgarden and CLRS differs slightly from ours. First, instead considering the data associated with a key to be an item stored at the same node, they treat the node itself as the data associated with a key. So, for example, to replace one data element with another, in our formulation, we could just rewrite the item and key attributes of the corresponding node, whereas the texts would require removing the node from the tree and rewiring pointers to the new node. Relatedly, the texts also include a parent pointer at every node. It's not necessary for anything we are going to do, but it can be useful in implementations (so feel free to include it if you prefer).

It is also worth noting that in practice, trees with fanout greater than 2 are often used for greater efficiency in terms of memory accesses. Specifically, in large data systems, the fanout is often chosen so that the size of each node is a multiple of the size of a "page" in memory or on disk.

Theorem 2.2. *Given a binary search tree of height (equivalently, depth) h, all of the following operations (queries and updates) can be performed in time $O(h)$:*

1. *Search (or membership) queries: given K, return a matching pair (I, K) in the dataset specified by the updates (if one exists)*
2. *Insertion updates: given an item-key pair (I, K), add (I, K) to the dataset.*
3. *Minimum queries: return the pair (I, K) with the smallest value of K in the dataset specified by the updates*
4. *Maximum queries: return the pair (I, K) with the largest value of K in the dataset specified by the updates*
5. *Predecessor queries: given a key K', return the item-key pair in the dataset with the maximum key (I, K) such that $K \leq K'$, or $\perp$ if no such key K exists*
6. *Successor queries: analogous to predecessor queries*
7. *Deletion updates: given a key K, delete a matching pair (I, K) from the dataset specified by the updates (if one exists)*

Proof.

- **Search:**

```
1 Search(T,K)
2 Let v be the root of T.
3 if K_v = K then
4 |  return (I_v, K_v)
5 if K > K_v and T_R ≠ ∅ then
6 |  return Search(T_R,K)
7 if K ≤ K_v and T_L ≠ ∅ then
8 |  return Search(T_L,K)
9 return ⊥
```

  Our proof of correctness uses induction on the height of the tree. Specifically, we prove the following statement by induction on h:

  > (*) for every BST T of height at most h, if T contains a node with key K, then `Search`(T, K) returns a pair (I_w, K_w) such that $K_w = K$, and otherwise `Search`(T, K) returns $\perp$.

  Our base case is height 0, i.e. a tree whose only node is the root v. In this case, T contains K if and only if $K = K_v$, which is exactly the condition under which `Search`(T, K) returns

(I_v, K_v). Otherwise, `Search`(T, K) returns $\perp$, since $T_R = T_L = \emptyset$. So `Search`(T, K) is correct.

For the induction step, assume that (*) is true for trees T of height at most h, and we'll prove it for trees of height at most $h+1$. Given an arbitrary tree T of height $h+1$ and a search key K, we split into casework. Letting v be the root of T, if $K_v = K$ we are successful because we return (I_v, K_v). If $K_v < K$ then by the BST property K is in T if and only if K is in the right subtree T_R (which is of height at most h), so the recursive call `Search`(T_R, K) succeeds by induction, and the left call is similar.

For the runtime, note that we do a constant amount of work at each level and recurse at most h times on a tree of height h. To formally analyze this, let $T(h)$ be the maximum amount of work done by a search call on a tree of height at most h. Then for $h > 0$, we have

$$T(h) = T(h-1) + c,$$

for a constant c. Unrolling, we get

$$T(h) = T(h-1) + c = (T(h-2) + c) + c = \cdots = T(0) + h \cdot c = O(h).$$

- **Insertions:**

```
1  Insert(T,(I, K))
2  Let v be the root of T.
3  if K ≤ K_v then
4  |  if T_L ≠ ∅ then
5  |  |  Insert(T_L,(I, K))
6  |  |  return
7  |  else
8  |  |  Let T_L = (I, K).
9  |  |  return
10 else
11 |  if T_R ≠ ∅ then
12 |  |  Insert(T_R,(I, K))
13 |  |  return
14 |  else
15 |  |  Let T_R = (I, K).
16 |  |  return
```

- **Minimum (and Maximum):**
  For Minimum, we always go left until there is no left child.

- **Predecessor (and Successor):**
  Given a query q, we again use a search type strategy. If $K_v = q$, we have found the predecessor and can return (I_v, K_v). If $K_v > q$, return Predecessor on the left subtree T_L (if it exists - if not, return $\perp$). Finally where $K_v < q$, call Predecessor on the right subtree T_R (if it exists). If we receive $\perp$, return (I_v, K_v) and otherwise return the result of the call on T_R.

- **Deletions:** next class active learning exercise

□

In-class Exercise: Which of the following are valid binary search trees?

In-class Exercise : which of the following two sequences of operations on an initially empty BST `T` will take asymptotically longest as a function of $U - L$, where $U \geq L$ are natural numbers? Below, `T.insert(i)` refers to inserting an empty item with a key of `i`.

```
1 InsertDescending(L,U) foreach i=U-1 down to L do
2 |   T.insert(i)
```

```
1 InsertRecursively(L,U)
2 if U > L then
3 |   M = floor((U+L)/2)
4 |   T.insert(M)
5 |   InsertRecursively (L,M)
6 |   InsertRecursively (M+1,U)
7 |   return
```

Let $n = U - L$. After j iterations of the loop in `InsertDescending(L,U)`, the height of the tree is j an the insertion takes time $O(j+)$. Thus the runtime is

$$O\left(\sum_{j=0}^{n-1} j\right) = O(n^2).$$

Whereas `InsertRecursively(L,UI)` produces a tree of height $\lceil \log n \rceil$ (here and throughout the course all log's are base 2 unless otherwise specified), and so all operations take time $O(\log n)$ and

thus total runtime is

$$O(n \log n).$$