Binary Search Trees and Randomization

Want a data structure (dictionary) with following operations:

1. Find \( (k, S) \): Is \( k \in S \)?
2. Insert \( (k, S) \)
3. Delete \( (k, S) \)

If only want three, then use hashing (hash table).
But also want

4. Range \( (k_1, k_2, S) = \# \text{ of keys in range } k_1, \ldots, k_2 \)
   \[ = \# \{ x \in S \mid k_1 \leq x < k_2 \} \]

In this case, **binary search trees are good**.

**BST:** T is a BST for set S if

1. T is a binary tree.
2. Every node stores a key \( \in S \).
3. All keys are in **order**

i.e., if node stores \( k \in S \), then left child of node contains only keys from \( \{ x \mid x \in S, x < k \} \)
and right child has only larger keys than \( k \).

E.g., \( 1, 2, 3, 4 \) could give

- \( 3 \)
- \( 1 \rightarrow \rightarrow 4 \) or \( 1 \rightarrow 2 \rightarrow 4 \) or \( 1 \rightarrow 4 \)
- \( 3 \)
- \( 1 \rightarrow 3 \) or \( 1 \rightarrow 2 \)
Easy to search for key $k$.

Compare and walk down tree.

$\text{search}(k, T) = \begin{cases} 
\text{if } T.\text{root} = k \text{ then return YES} \\
\text{else see if } (k < T.\text{root}) \text{ search } (k, T.\text{left}) \\
\text{else if } (k > T.\text{root}) \text{ search } (k, T.\text{right}) \\
\text{of course if } T = \emptyset \text{ say NO.}
\end{cases}$

Note: time to search depends on depth.

Balanced BST: depth $= O(\log n)$

Since binary tree, depth $= O(\log n)$.

Examples: worst case balanced: AVL trees, Red-Black trees, B-trees
randomized: skip lists
amortized: splay trees

Important Operation: Rotate

Note: local operation, only $O(1)$ pointers changed. $x, p, q$ unchanged.

Crucial: inorder property preserved!!
All these BSTs use rotation to balance trees. We look at trees today.

- Search is same as above — no change.

- Insert: Let’s first do a “vanilla” insert

  Insert \((k, T)\)  
  Suppose \(k\) is not in tree,  
  then we reach leaf.  
  Add \(k\) to that leaf.

  e.g. \(T = \)  

  \[
  \begin{array}{c}
  4 \\
  3 \\
  \end{array}
  \]

  Insert \((1, T)\) —  

  Insert \((35, T)\) —  

  etc.

Problem: Insert 1, 2, 3, ..., \(n\) in that order.

get very unbalanced tree

How to avoid? Randomization!!

Idea: Every key \(k \in S\) will be given a priority \(p(k)\).

Ensure that we maintain BST that is  
(a) BST with respect to keys  
(b) heap with respect to priorities
E.g.: \( a \) \( b \) \( c \) \( d \)

\[
\begin{align*}
p(a) &= 7\quad & p(b) &= 3\quad & p(c) &= 9\quad & p(d) &= 1
\end{align*}
\]

then get

Claim: There is a unique heap-order BST given keys' priorities.

Pr: do vanilla insert in priority order

\[ \]

How to assign priorities? Randomly!!

Intuition: if random priorities for n elements then

\[ a_1 < a_2 < \ldots < a_n \]

\[ \text{min priority is for item approximately in the middle } \approx \text{a}_m\]

\[ \Rightarrow \text{splits evenly, get} \]

\[ \]

Of course this is just intuition

Let's see if this works.

And how to do it efficiently (even with inserts/deletes).
**Insert(k):**

1. Do vanilla insert(k) into T.
   So k is at a leaf.
2. Pick a random priority \( \beta(k) \).
3. Rotate k up until in heap order.

**Example:**

Insert f into

```
  m
  /  \
 t   f
 /    /\  \
 a    g
```

```
  m
  /  \
 t   f
 /    /\  \
 a    g
```

```
  m
  /  \
 t   f
 /    /\  \
 a    g
```

**Smirably:**

**delete(k):**

1. Rotate k down to a leaf each time picking a child with higher priority.
2. Delete k.

**Note:** If we delete f, the above movie will be run backwards.

To delete c instead, T_final rotate:
Now, want that insert, delete, find all take $O(\log n)$ expected time.

We will first give upper bound on the # of comparisons for a failed search (where element is not in the BST).

And then show that costs of all other ops are $\leq$ this bound (upto constants).

For concreteness: suppose keys in current BST are $1,2,\ldots, n$.

Of course they have random priorities.

And we search for $a.5$ where $a \neq \text{integer}$ (so $a.5$ not in BST)

Let search for $7.5$.

What's $E[\# \text{ comparisons to search for } a.5]$? (**) 

$E$ expectation over the random priorities assigned to keys.

This does not depend on data. Just internal implementation.

OK: define random variable (indicator)

$$X_i = \begin{cases} 1 & \text{if } a.5 \text{ and } i \text{ are compared during find(a.5)} \\ 0 & \text{otherwise} \end{cases}$$

By linearity of expectation:

$$E[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} E[X_i]$$

$$= \sum_{i=1}^{n} \Pr[i \text{ and } a.5 \text{ compared}].$$

Claim: this probability $\leq \frac{1}{|i-a|}$

Formally: $\Pr(i \text{ and } a.5 \text{ compared}) = \begin{cases} \frac{1}{i-a} & \text{if } i > a \\ \frac{1}{a-i+1} & \text{if } a < i \leq a.5 \end{cases}$
Example: \( S = \{1, 2, 3\} \), search for 2.5

Expectation over all possible orderings of priorities

Priority: \( P_1 < P_2 < P_3 \) \( P_1 < P_3 < P_2 \) \( P_2 < P_1 < P_3 \) \( P_3 < P_1 < P_2 \) \( P_3 < P_2 < P_1 \)

Comparison for find(2.5)

\[
\text{Expectation} = \frac{1}{3!} (3 + 3 + 2 + 2 + 2) = \frac{15}{6} = 2.5
\]

Note: different than taking random BST, since some BSTs appear more than once.

Exp if we take random BST = \( \frac{3 + 3 + 2 + 2 + 2}{5} = \frac{13}{5} = 2.6 \)
So the probabilities fall like this:

\[
\begin{array}{cccc}
1 & 2 & i & a & n \\
\frac{1}{a} & \ldots & \frac{1}{3} & \frac{1}{2} & \frac{1}{2} & \frac{1}{3} & \ldots & \frac{1}{n-a} \\
\end{array}
\]

\[\Rightarrow \text{the sum is } \leq 2 \left(1 + \frac{1}{2} + \ldots + \frac{1}{n}\right) = 2H_n \leq 2(\ln n + O(1)).\]

\[E[\text{#comparisons for generic find (a,b)]} \leq O(\log n).\]

Let's prove the Claim1: (about \(Pr(i \text{ and } a \text{ are compared})\))

- Let's build the heap (and choose the random priorities) in our minds.

- Say \(a \leq a\) instead of choosing random priorities for keys, let's choose instead the key with lowest priority value.

this will be root of heap.

- If this falls in \([a, a]\) we will not compare to \(i\).

- If this equals \(i\) we will certainly compare to \(i\) (since \(i \in \text{root}\)).

In all other cases both \(i\) and \(a\) fall on same side of root and we defer this decision to next round and have same argument.

- But eventually \(a \leq a\) and \(a \leq i\) so the chance that we compare to \(i\) is \(\frac{1}{a-1}\).
In pictures:

```
 i  a
```

- If choose i, then compare
- If choose anyone else, don't compare

\[
\frac{1}{a-i+1} = \frac{1}{\text{length of interval in interesting case}}
\]

Same as: Suppose 9 said, roll a die. (6 x sided).

- If even, roll again.
- If odd, you win if get 5.

What is the chance of winning? \( \frac{1}{6} \).

Because keep rolling until get to interesting case
and then conditioned on being in this case

1 outcome wins
2 outcomes lose \( \Rightarrow \frac{1}{3} \).

All three outcomes equally likely.
Saw that $E[\text{cost of find}(a) \text{ when } a \text{ not in BST}] \leq O(\log n)$.

How about other cases:

**Find:**

\[
\begin{align*}
E[\text{cost of find}(a) \text{ if } a \text{ is BST}] & \leq E[\text{cost of find}(a) \text{ if } a \text{ not in BST} \\
& \quad \text{and no elements between } a \text{ and } a] \\
& \leq O(\log n).
\end{align*}
\]

**Insert:**

\[
\begin{align*}
E[\text{cost of insert}(a)] & \leq E[\text{cost of find}(a)] \\
& \quad + E[\text{cost of rotation}] \quad \text{# A rotation} \\
& \quad \leq \text{depth of } a \\
& \leq O(E(\text{cost of find } a)).
\end{align*}
\]

**Delete:**

\[
\begin{align*}
E[\text{cost of delete}(a) \text{ when } a \text{ is BST}] & \leq E[\text{cost of find}(a)] \\
& \quad + E[\text{length of leftmost path of right subtree}] \\
& \quad + E[\text{length of rightmost path of left subtree}] \\
& \leq 3O(\log n) \quad \text{for each form}
\end{align*}
\]

Why?

When we rotate a down either leftmost path length loses 1
or rightmost path length loses 1.

Note: we assume all operations are in a fixed sequence, not being chosen adaptively

\( \Rightarrow \) called "oblivious" adversary.
OK: to wrap up -

saw treaps = trees + heaps.

**Binary Search Tree Data Structure**
- find, insert, delete.
- also range \((k_1, k_2)\) — exercise!

† Randomness is useful (very)
- Expected time for operations = \(O(\log n)\).
- Simple data structure
- Simple analysis.

† Linearity of expectations is powerful hammer
- reduces complicated objects to simpler ones.

† Same data structure can be implemented in different ways.
  - same API, different internals
  - think about what is best for your application.
  - (used heap property internally, even though BST does not require it.)

\(\text{Added: You can view this as a representation of left-child-right (LCR).}\)
\(\text{(depends on context; if your insert is get in random order.)}\)