15-750: Graduate Algorithms January 25, 2017

Lecture4: BST Introduction and Treaps

Lecturer: Gary Miller *Scribe: Mu-Chu Lee, C.J. Argue*

1 Binary Search Tree

1.1 Operations

We want to design a data structure, an ordered set S that supports the following operations:

1. $Search(k, S)$
2. $Insert(k, S)$
3. $Delete(k, S)$
4. $Range(k, k^{'}, S) \equiv |\{k^{''} \in S | k \le k^{''} \le k^{'}\}|$

If only (1), (2), (3) are required, we could use a hash table. However, if we also need (4), then we will need a BST.

Definition 1.1. T is a BST for keys S if:

1. T is an ordered binary tree with $|S|$ nodes.
2. Each node stores a unique key, i.e., there is a 1-1 correspondence between keys and nodes.
3. With in-order traversal, we would gain an ordered sequence, i.e., the keys are stored in in-order.

Definition 1.2. T is a *balanced* BST if $\textit{max-depth}(T) = O(\log n)$, and can be further categorized:

1. Always balanced: AVL, 2-3-4, Red-Black, B-Trees
2. Randomized balanced: Skip-lists, Treap
3. Amortized balanced: Splay tree

1.2 Rotations

All of the balanced tree mentioned above use similar techniques, which is rotations. Rotations are used so that we could change the root of some subtrees T', but the whole tree T and T' still have the BST properties.

The figure above shows the operation of one rotate, where capital A, B, C are subtrees whereas a, b are nodes. The rotation could be done efficiently since only three pointers need to be modified (the colored ones).
One should check that after rotation, the properties of BST still holds. Since the whole subtree A are smaller than b, therefore it is kept at the left child of b. b was the left child of a, therefore after rotation, a would be the right child of b. The subtree B is greater than b, but is in the left subtree of a, therefore, in the new tree it remains in a's left child or subtree.

2 Applications

2.1 Persistency

Let say we want an additional operation, *Persistence*, which is to undo the last operation. We have two ways to achieve this:

1. We could save a tree for each time. We would need $O(nt)$ space, where n is the number of nodes and t is the total number of operations (trees to be saved).

2. We could also store the "difference" for each operations. We would give an example of insertion in the next page.

Suppose we are going to insert the red node marked with an X into T. We could observe that the whole structure remains the same, except the pointer and node marked as red will differ after the insertion operation. Inspired by this reason, instead of copying a whole tree, we could just reuse parts of T that is black. Therefore, we could just keep the nodes along the path (in this case, it would be a, b, and c). Let say the copied nodes are a', b', c', we could just insert the new node onto c', and copy the pointer of a, b, c to a', b', c' correspondingly.
The total space that is needed would be $O(t \log n)$.

2.2 Vanilla-BST

We will see the connection between quicksort and a Vanilla-BST.

Definition 2.1. Define $Vanilla\text{-}Insert(k, T)$ as inserting k to be a child of some leaf node.

Definition 2.2. Quick Sort normally pivots on a random. Key here we will use the first element. Let's call this procedure NotQuickSort. Thus $NotQuickSort(A)$ sorts the array A with the following procedure:

1. Pick first a in A.
2. Split A into three sets, $S < a$, a, $a < L$.
3. Return $NotQuickSort(S)$, a, $NotQuickSort(L)$.

Definition 2.3. $Vanilla\text{-}BST(A, T)$ follows the procedure:

1. Extract first a from A
2. Return $Vanilla\text{-}BST(A, Vanilla\text{-}Insert(a, T))$

One should check that NotQuicksort and Vanilla-BST do exactly the same comparisons, but in different order. We can think of NotQuicksort as eager and Vanilla-BST as lazy.

3 Treap

Lets mention the structure and elements of a Treap first. A treap has:

1. Keys $\equiv \{1, \ldots, n\}$
2. Priorities p for each key, such that $p(k) \neq p(l) \quad \forall k \neq l$.

Let T is a tree with a key at each node.

Definition 3.1. T is in *heap order* if $\forall x \in T$, and x is not the root, $p(parent(x)) < p(x)$.

Claim 3.2. *A heap order BST exists and is unique.*

Proof. Suppose we have a set of keys A with priorities. Now sort the keys of A by their priorities, and construct the tree T = Vanilla-BST(A). Note that T is in heap order. Thus heap ordered BST's exist. By induction on the size of A we get the tree is unique since the highest priority key must be the root. The remaining keys must be uniquely in the left or right subtree of the root. we are done by induction.

□

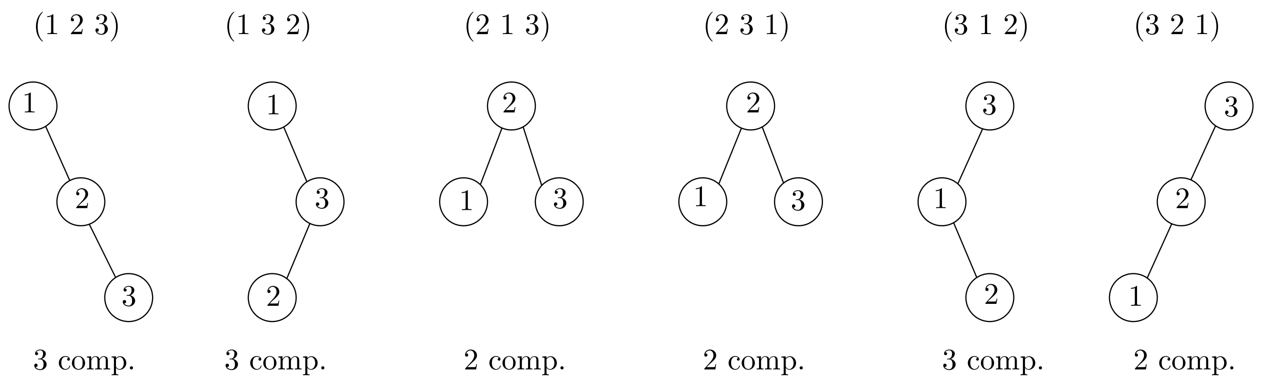
3.1 Random Treap

A Random Treap is built by assigning random keys. Accordingly, we have the following operations Insert:

3.2 Expected Cost

Suppose we build a random treap with keys $K = \{1, 2, \ldots, n\}$, and we are searching for key value $m+0.5$ for some $m \in \{1, 2, \ldots, n\}$. We want to compute the expected number of comparisons done in the search, where the expectation is taken over the random assignment of priorities. For a given tree, we will compare $m + 0.5$ with a node v, and then successively with the left or right child of v according to whether $m + 0.5$ is greater or less than v. When v has no left/right child as needed, the search terminates.

As an example, we compute the average number of comparisons made in searching for 2.5 in a random Treap with keys $\{1, 2, 3\}$. The figure below shows all possible trees, and the number of comparisons made in searching for 2.5 in each. For example, in the tree (1 2 3), we compare $2.5 > 1$, so we look at the right child of 1. Then we compare $2.5 > 2$ so we look at the right child of 2. Then we compare $2.5 < 3$ so we look at the left child of 3, which does not exist, so we stop having made 3 comparisons.



Definition 3.3. Define $C(i, m)$ to be the event that m is compared to i while performing Search(m).

For example, in the tree (2 1 3), $C(1, 2.5) = 0$, $C(2, 2.5) = 1$, $C(3, 2.5) = 1$.

We next determine the probability of the event $C(i, m)$.

Lemma 3.4. *Then*

$$\mathbb{P}[C(i, m)] = \begin{cases} \frac{1}{m-i+1} & : \quad i \le m \\ \frac{1}{i-m} & : \quad i > m \end{cases}$$

Proof. We will prove the case $i \le m$, and the other case is analogous. The following proof is slightly different than the one in lecture.

Let K be random variable which returns the element of $\{i, i+1, \ldots, m\}$ with the highest priority. We next show that the event $E(i, m)$ that $K = i$ is identical to $C(i, m)$.
Proof of claim:

For every node v in the path from the root to k, either $v < i$ or $v > m$, so every element of $\{i, i+1, \ldots, m\}$ is in the same subtree of v as k. Thus every element of $\{i, i+1, \ldots, m\}$ is in the subtree rooted at k. When searching for m, we will enter this subtree, and either stop at the root if $k = m$, or continue into the right subtree if $k < m$. If $k = i$ then we compare m to i, i.e. $C(i, m)$

occurs. Otherwise $i < k$ so i is in the left subtree of k, and we do not compare m to i, i.e. $C(i,m)$ does not occur. Thus $C(i,m)$ occurs precisely when i has the highest priority of $i, i+1, \ldots, m$.

Since the priority order is uniformly random among all permutations of $[n]$, the variable K is uniform over $\{i, i+1, \ldots, m\}$. Thus $\mathbb{P}[C(i,m)] = \mathbb{P}[E(i,m)] = \frac{1}{m-i+1}$. □

Theorem 3.5. *Let $S(x)$ be the number of comparisons made while performing Search(x). Then $\mathbb{E}[S(m+0.5)] = O(\lg n)$.*

Proof. Let $\chi(i,x)$ be the indicator random variable of the event $C(i,x)$. Then $S(m+0.5) = \sum_{i=1}^{n} \chi(i, m+0.5)$, so by linearity of expectation

$$\begin{aligned}
E[S(m+0.5)] &= \mathbb{E}\left[\sum_{i=1}^{n} \chi(i, m+0.5)\right] \\
&= \sum_{i=1}^{n} \mathbb{E}[\chi(i, m+0.5)] \\
&= \sum_{i=1}^{n} \mathbb{P}[C(i, m+0.5)] \\
&= \sum_{i=1}^{m} \frac{1}{m-i+1} + \sum_{i=m+1}^{n} \frac{1}{i-m} \\
&\leq 2\sum_{i=1}^{n} \frac{1}{i} \\
&= 2H_n \\
&= 2\lg n + O(1)
\end{aligned}$$

□

To delete a node, we rotate it to a leaf

Let m be a node. Then a node n is a right-most node of the left-subtree of m (RMLS) if n is the left-child of m or n is the right-child of an RMLS node. A left-most node of the right-subtree is defined similarly.

Claim 3.6. *In delete(m), m is pivoted with the right-most nodes in its left-subtree and the left-most nodes in its right-subtree, exactly once each.*

Proof. Let $v_1, \ldots, v_l$ be the number of left-most nodes of the right-subtree of m. Let $u_1, \ldots, u_r$ be the number of right-most nodes of the left-subtree of m. We proceed by induction on $l + r$.

Base: If $l = r = 0$, then m is already a leaf, so the claim holds.

Induction: Suppose that $l + r = k$ and we have proven the claim when $l + r < k$. WLOG $l > 1$ and WLOG we first rotate m with v_1. The left subtree of m remains unchanged, so its right-most nodes remain $u_1, \ldots, u_r$. The right subtree of m is now the tree rooted at v_2, so its left-most nodes are $v_2, \ldots, v_l$. By induction, from here forwards m is rotated precisely with $u_1, \ldots, u_r$ and $v_2, \ldots, v_l$, and so far it was only rotated with v_1, so the induction is complete. □

Define random variables

$$D_i = \begin{cases} 1 & : \quad i \text{ is a right-most node in the left subtree of } m \\ 0 & : \quad \text{otherwise} \end{cases}$$

$$D'_i = \begin{cases} 1 & : \quad i \text{ is a left-most node in the right subtree of } m \\ 0 & : \quad \text{otherwise} \end{cases}$$

We define D_i only for $i < m$ since only these i can be in the left-subtree of m, and similarly we define D'_i only for $i > m$.

Lemma 3.7. $\mathbb{P}[D_i = 1] = \left(\frac{1}{m-i+1}\right) \cdot \left(\frac{1}{m-i}\right)$ *and* $\mathbb{P}[D'_i = 1] = \left(\frac{1}{i-m+1}\right) \cdot \left(\frac{1}{i-m}\right)$.

Proof. We give the proof for D_i, and the other case is analogous. We claim that D_i occurs precisely when both of the following two events occur:

A: m has the highest priority of $\{i, i+1, \ldots, m\}$.

B: i has the highest priority of $\{i, i+1, \ldots, m-1\}$.

Given this claim, we have $\mathbb{P}(A) = \frac{1}{m-i+1}$ and $\mathbb{P}(B) = \frac{1}{m-i}$ as argued above. Since A, B are independent, we have $\mathbb{P}(D_i) = \mathbb{P}(A \cap B) = \mathbb{P}(A)\mathbb{P}(B) = (\frac{1}{m-i+1}) \cdot (\frac{1}{m-i})$. It remains to prove the claim.

As in lemma 3.4, let K_1 and K_2 be the random variables returning highest-priority and second highest-priority elements of $\{i, i+1, \ldots, m\}$. Consider the event that $F(m,i) : K_1 = m\mathrm{K}_2 = i$ and the event $G(m,i)$ that m is rotated against i on rotating m to root.

Claim: $F(m,i) = G(m.i)$

Proof:

All of $\{i, i+1, \ldots, m\}$ are in the subtree rooted at k. Either $m = k$ or m is in the right subtree of k and i cannot be in the right subtree of k, so if i is in the subtree rooted at m, we must have $m = k$. Let l be the second-highest priority element of $\{i, i+1, \ldots, m\}$. Then either $i = l$, or i is in the left-substree of l, so i is only a right-most node in the left subtree of m when $i = l$.

Thus $\mathbb{P}[G(m,i)] = \mathbb{P}[F(m,i)]$ which is equal to $\left(\frac{1}{m-i+1}\right) \cdot \left(\frac{1}{m-i}\right)$ for $i \leq m$. □

Now we let R_m be random variable returning the number of rotations performed when moving m to a leaf.

Theorem 3.8. $\mathbb{E}[R_m] < 2$.

Proof. First note that from Claim 3.6, $E[R_m] = \sum_{i<m} D_i + \sum_{i>m} D'_i$. Thus we have

$$\begin{aligned} E[R_m] &= \sum_{i<m} D_i + \sum_{i>m} D'_i \\ &= \sum_{i=1}^{m-1} \left(\frac{1}{m-i+1}\right) \cdot \left(\frac{1}{m-i}\right) + \sum_{i=m+1}^{n} \left(\frac{1}{i-m+1}\right) \cdot \left(\frac{1}{i-m}\right) \\ &= \sum_{j=1}^{m-1} \left(\frac{1}{j+1}\right) \cdot \left(\frac{1}{j}\right) + \sum_{j=1}^{n-m} \left(\frac{1}{j}\right) \cdot \left(\frac{1}{j+1}\right) \\ &= \sum_{j=1}^{m-1} \frac{1}{j} - \frac{1}{j+1} + \sum_{j=1}^{n-m} \frac{1}{j} - \frac{1}{j+1} \\ &= \sum_{j=1}^{m-1} \frac{1}{j} - \sum_{j=2}^{m} \frac{1}{j} + \sum_{j=1}^{n-m} \frac{1}{j} - \sum_{j=2}^{n-m+1} \frac{1}{j} \\ &= 2 - \frac{1}{m} - \frac{1}{n-m+1} \\ &< 2 \end{aligned}$$

□