ICS 621: Analysis of Algorithms Fall 2019

Lecture 7: Randomized search trees

Prof. Nodari Sitchinava *Scribe: Jeremy Ong and Muzamil Yahia*

1 Treaps

1.1 Basic Structure

A treap is a search tree data structure based on binary search trees (BST) and heaps. The basic structure of a treap is that of a binary tree, with each node containing a key and a priority. The keys are ordered to fit the BST property, where each node has a key greater than the keys of all nodes in its left subtree, and smaller than the keys of all nodes in its right subtree. The priorities of each node are ordered to maintain the heap property, where the priority of each node is smaller than the priorities of its children. Equivalent to a treap is a BST constructed by inserting nodes based on their priority, using standard BST insertion.

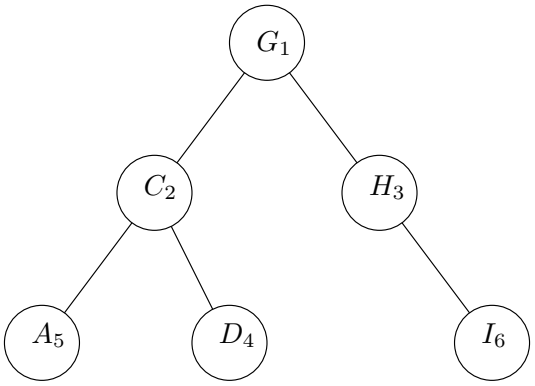

Figure 1: A treap with each node K_P having key $= K$ and priority $= P$

Treaps can also be described geometrically. For a given set of nodes, let each node be a point on an xy plane, with the key of each node as its x-coordinate, and the priority of each node as its y-coordinate. Additionally, let the y-axis of the graph be reversed, so that the lowest priority point is the highest on the graph. By sectioning off the graph by the y-intercepts and the lower portion of the x-intercepts, and then connecting each node to the right and left nodes (if any) that are within the boundaries of each section, we will recreate the same treap structure as defined above.

This structure is known as a Cartesian tree, one application of which is a 3-sided query. An example of this would be a database search for all people whose income is greater than a threshold (1 side) and who are within a certain age range (2 sides).

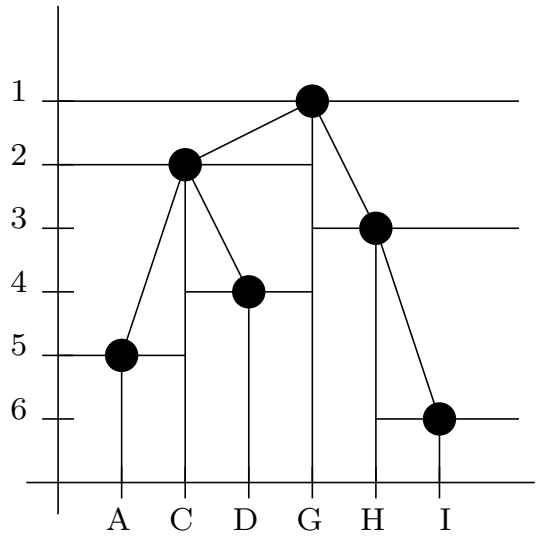

Figure 2: A Cartesian tree representation of the treap

1.2 Treap Operations

Insertion into a treap can be accomplished by performing a normal BST insertion, then performing rotations on the inserted node until the heap property is restored.

Algorithm 1

```
1: procedure TREAP-INSERT(T, v)
2:     BST-INSERT(T, v)
3:     while v.priority < v.parent.priority do
4:         ROTATE(v)
```

The initial BST insertion ensures that the BST order is maintained. Rotations do not affect the BST order, and they are performed until the priority of v is greater than its parent's, ensuring that the heap property is maintained at the end of the treap insertion.

Deleting a node v is accomplished by increasing the priority of v to infinity, then repeatedly rotating on the child of v, which has lower priority of the two children, until v becomes a leaf, at which point we can remove it.

Algorithm 2

```
1: function TREAP-DELETE(v)
2:     TREAP-INCREASE-PRIORITY(v, ∞)
3:     DELETE(v)
```

```
1: function TREAP-INCREASE-PRIORITY(v, new_priority)
2:     v.priority ← new_priority
3:     while v is not a leaf do
4:         if v.right = nil or (v.left ≠ nil and v.left.priority < v.right.priority) then
5:             w ← v.left
6:         else
7:             w ← v.right
8:         ROTATE(w)
```

A treap can also be split into two treaps along a specified key value. The result is one treap containing all nodes with keys that are smaller than the specified value and another treap containing all nodes with keys that are greater.

Algorithm 3

```
1: function TREAP-SPLIT(T, key)
2:     v ← new node(key, −∞)
3:     TREAP-INSERT(T, v)
4:     T_< ← v.left
5:     T_> ← v.right
6:     DELETE(v)
7:     return (T_<, T_>)
```

Likewise, two treaps can be merged into a single treap.

Algorithm 4

```
1: function TREAP-MERGE(T_<, T_>)
2:     v ← new node(nil, nil)
3:     v.left ← T_<
4:     v.right ← T_>
5:     T ← v
6:     TREAP-DELETE(v)
7:     return T
```

1.3 Treap Operation Analysis

So what are the runtimes of these operations? In each case, we need to traverse across the height of the treap. This gives us a runtime of $O(height) = O(\log n)$ in a balanced treap, since a balanced binary tree has a height of $O(\log n)$. However, nothing about the treap structure guarantees balance; a treap height could easily be linear. So instead of guaranteeing balance, we can use randomized priorities for the nodes. In fact, the definition of a treap is a BST containing nodes of randomized priorities organized to maintain the heap property (Cartesian trees do not have random priorities). When priorities are randomly chosen from a uniform distribution, the result is a tree with expected height of $O(\log n)$.

To prove the expected height of the treap, we start by defining some variables: let x_k be the node with the k^{th} smallest key in the treap. Then we want to prove that for any node x_k, $E[depth(x_k)] = O(\log n)$.

Let Y_{ij} be an indicator random variable defined as $I\{x_i \text{ is a proper ancestor of } x_j\}$. So, $Y_{ij} = 0$ when x_i is not a proper ancestor of x_j and 1 when it is. Note that $Y_{ii} = 0$.

The depth of any node is equal to the total number of all nodes that are its proper ancestors. Which can be written as:

$$depth(x_k) = \sum_{i=1}^{n} Y_{ik}$$

The expected depth of x_k is then equal to the expected number of proper ancestors of x_k. Using linearity of expectations we get:

$$E[depth(x_k)] = E\left[\sum_{i=1}^{n} Y_{ik}\right] = \sum_{i=1}^{n} E[Y_{ik}]$$

And since the expected value of an indicator variable is equal to the probability of the indicator variable being equal to 1:

$$E[depth(x_k)] = \sum_{i=1}^{n} Pr[Y_{ik} = 1]$$

Now let us define a set $x(i, j) = \{x_i, x_{i+1}, ...x_{j-1}, x_j\}$ such that all keys are in ascending sorted order from x_i to x_j.

Lemma 1. *$Y_{ij} = 1$ if and only if x_i has the smallest priority among all keys in $x(i, j)$.*

Proof. To prove the above lemma, let us consider four cases:

Case 1: x_i is the root of T.

If x_i is the root of the entire treap, then by the heap property, it must have the lowest priority of all nodes. By definition of a binary tree, x_i is the ancestor of all other nodes, and is therefore a proper ancestor of x_j.

Case 2: x_j is the root of T.

If x_j is the root of the entire treap, then it must have the lowest priority among all nodes. Therefore, x_i cannot be the lowest priority node in $x(i, j)$. Additionally, since x_j is the ancestor of all nodes, x_i cannot be a proper ancestor of x_j.

Case 3: x_i and x_j are in different subtrees.

Since x_i and x_j are in different subtrees, there must exist some value k such that $i < k < j$ and the priority of x_k is less than the priority of x_i. Since x_i and x_j are in different subtrees, x_i cannot be the ancestor of x_j.

Case 4: x_i and x_j are in the same subtree.

If x_i and x_j are in the same subtree, then the lemma must be true by induction, since the subtree is a smaller case of our original treap, T. □

Now that we have proven that a node x_i is only a proper ancestor of x_j iff x_i has the lowest priority in $x(i,j)$, to get the expected number of ancestors of any node x_k, we can simply sum the expected values of Y_{ik} for all nodes x_i.

$$E[depth(x_k)] = \sum_{i=1}^{n} Pr[x_i \text{ has the smallest priority in } x(i,k) \ \wedge i \neq k]$$

Since the priorities of $x(i,k)$ are uniformly distributed, i.e., each of $\{x_i, \ldots, x_k\}$ is equally likely to be the smallest,

$$\begin{aligned}
E[depth(x_k)] &= \sum_{i=1}^{n} \frac{1}{|i-k|+1} - 1 \\
&= \sum_{i=1}^{k-1} \frac{1}{k-i+1} + \sum_{i=k+1}^{n} \frac{1}{i-k+1} \\
&= \sum_{j=2}^{k} \frac{1}{j} + \sum_{j=2}^{n-k+1} \frac{1}{j} \\
&\leq 2\sum_{j=2}^{n} \frac{1}{j} < 2\int_{1}^{n} \frac{1}{x} dx = 2\ln n \qquad \text{(geometrically justified)} \\
&= O(\log n).
\end{aligned}$$

As the expected depth of any node is $O(\log n)$, the above operations all run in $O(\log n)$ time.

2 Skip lists

Introduced by William Pugh[1] in [Pug90], skips lists are *probabilistically* balanced data structures that offer the same (expected) bounds for insert, search, and deletion operations as balanced BSTs. They are built to offer faster and simpler implementations than balanced BSTs. Since they are probabilistic in nature, skip lists are as good as balanced BSTs on average, but may exhibit lower performance in unlikely negligible extreme cases.

2.1 Intuition

A skip list, more or less, is like an express train that skips some stops on the way compared to a normal train that stops at each location. To reach a destination, one takes the express train to the last reachable stop before a destination on express train, and then switches to a normal train for the rest of the trip if needed. Similarly, given a sorted set $X_1 = \{x_1, \ldots, x_n\}$ stored in a linked list, another linked list $X_2 = \{x_k, x_{2k}, \ldots, x_{\ell k}\} \subseteq X_1$, where $n = \ell k$, can be constructed such that

[1] Pugh introduced the problem in Algorithms and Data Structures, Workshop WADS in 1989 before publishing it in 1990.

$x_{ik} \in X_2$ points to $x_{ik} \in X_1$. Note that $\ell(k-2)$ elements are skipped from X_1 in X_2. To search for an element x, one traverses the second list X_2, till reaching a node v that points to a node with bigger value or equal to x. Using the pointer in v to X_1, the search can be continued for x in X_1. See Figure 3.

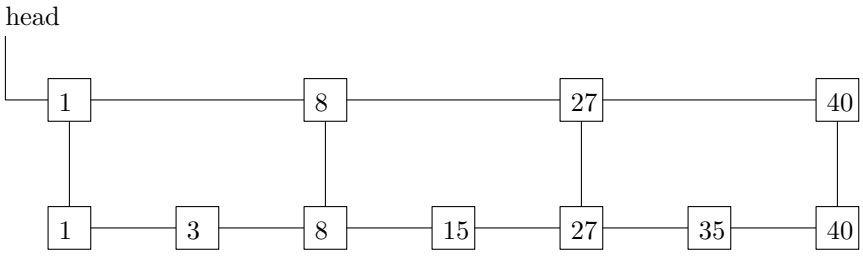

Figure 3: Skip list with two levels

2.2 Optimal skip list parameter k

Given a list X_1 and a skip list X_2, the cost for a search for x in the worst case is $\frac{n}{k} + k$ where $\frac{n}{k}$ are spent on searching X_2, and k searches in X_1. Putting $f(k) = n/k + k$ where $1 \leq k \leq n$, then the first and second derivatives are $\frac{df(k)}{dk} = -n/k^2 + 1$, and $\frac{d^2 f(k)}{dk^2} = 2n/k^3$. Hence the minimum value for f is at $k = \sqrt{n}$ and the total cost is then $2\sqrt{n}$.

In similar way, we can define $X_3 = \{x_{k'}, x_{2k'}, \ldots, x_{\ell' k'}\}$ where $\frac{n}{k} = \ell = \ell' k'$ on top of X_2, and then the cost function $f(k, k')$ is given by $\ell' + k' + k = \frac{n}{kk'} + k' + k$. To calculate the minima, we take the derivatives of both k and k' to get the following:

$$\frac{df}{dk} = -\frac{n}{k'k^2} + 1 = 0 \qquad \Longrightarrow\ k'k^2 = n$$
$$\frac{df}{dk'} = -\frac{n}{kk'^2} + 1 = 0 \qquad \Longrightarrow\ kk'^2 = n$$

which gives $kk'^2 = k'^2 k$ or $k = k'$. Replacing k' with k in either equation gives $k = k' = \sqrt[3]{n}$ and the optimal worst cost $3\sqrt[3]{n}$. By doing a similar analysis for at level t, i.e., on X_t, we obtain optimal size $k = n^{1/t}$ with optimal worst case $t \cdot n^{1/t}$. Choosing $t = \log n$, gives $k = n^{1/\log n} = 2$, with worst case $2 \log n$.

Algorithm 5

```
1: function SEARCH(k, S)          ▷ Returns the node with the largest key smaller or equal to k
2:     v ← S.start
3:     while (v.next ≠ nil and v.next.val ≤ k) or (v.down ≠ nil) do
4:         if v.next ≠ nil and v.next.val ≤ k then
5:             v ← v.next
6:         else
7:             v ← v.down
8:     return v
```

2.3 Insertion

To insert a new key x into a skip list, we traverse the skip lists down to the bottom list X_1 and then insert a new node containing x into X_1. This takes $2\log n$ in the worst case, same cost as SEARCH. The question remains is *how efficiently can we promote elements into the upper lists X_i with $1 < i \leq \log n$?* In skip lists, the answer is surprisingly simple, use randomization by flipping a coin! As long as the coin keeps coming up HEADS, keep promoting the item to the next level.

Not so fast, we need to provide an analysis of why this works, and for that, we calculate the expected maximum level τ that an element $x \in X_1$ will reach i.e. $x \in X_\tau$.

Theorem 2. *The height $L(x)$ of a skip list is at most $(c+1)\log n$ with high probability i.e.*

$$prob[L(x) \geq c\log n] \geq 1 - \frac{1}{n^c}$$

for any constant $c > 0$.

Proof. An element x will be at level at least τ if at least first τ tosses of the coin were HEADS. The probability of first τ tosses to be HEADS is $1/2^\tau$. Hence, $Pr[L(x) \geq \tau] = 1/2^\tau$. Since we are looking for maximum $L(x)$ among all $x \in X_1$, therefore we calculate the following probability

$$\begin{aligned} Pr[\max_{x\in X_1}\{L(x)\} \geq \tau] &= Pr[L(x_1) \geq \tau \text{ or } L(x_2) \geq \tau \cdots \text{ or } L(x_n) \geq \tau] \\ &\leq \sum_{x\in X_1} Pr[L(x) \geq \tau] && \text{(by the Union Bound)} \\ &= \sum_{x\in X_1} \frac{1}{2^\tau} \\ &= n\frac{1}{2^\tau}. \end{aligned}$$

By choosing $\tau = (c+1)\log n$ then gives $Pr[\max_{x\in X_1}\{L(x)\} \geq (c+1)\log n] \leq n \cdot \frac{1}{2^{(c+1)\log n}} = \frac{1}{n^c}$. Hence, we have

$$Pr[\max_{x\in X_1}\{L(x)\} \leq (c+1)\log n] \geq 1 - \frac{1}{n^c}.$$

□

Algorithm 6

```
1: function INSERT(k, S)
2:     v ← SEARCH(k, S)                          ▷ the predecessor node at the bottom level
3:     if v.key ≠ k then
4:         listINSERT(v, new Node(k))            ▷ Insert a new node with key k into the list after v
5:         bit ← coinFLIP()
6:         while bit == HEADS do
7:             promote a copy of v to the level above
8:             bit ← coinFLIP()
```

References

[Pug90] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.