ICS 621: Analysis of Algorithms Fall 2021

Lecture 7: Randomized search trees

Prof. Nodari Sitchinava *Scribe: In Woo Park, Michal Rogers, Darlene Agbayani*

1 Overview

Previously, we covered Optimal Binary Search Trees (BST), by first constructing a BST then determining the recursive formula for the cost, and analyzing the runtime. In addition, we introduced Splay Trees, a self-balancing data structure as well as the three (3) reconstructing heuristics also known as Splay Operations. For each operation: Zig, Zig-Zag, and Zig-Zig, we discussed how rotations are made in pairs, dependent on the structure of the access path. Lastly, we analyzed the Amortized Cost of each Splay Operation using a potential function.

In this lecture, we will be introducing Treaps, defining their basic structure. As well as Treap Operations, including: Treap-Insert(T, v), Treap-Delete(v), Treap-Increase-Priority$(v, new_priority)$, Treap-Split(T, key) and Treap-Merge$(T_<, T_>)$. After we define the Treap Operations, we will then analyze their runtimes. Lastly, we will introduce Skip Lists and define their structure.

2 Treaps

2.1 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 tree maintains the BST property on the keys, i.e., 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 tree also maintains the heap property on the priorities, i.e., the priority of each node is smaller than the priorities of its children. Equivalent to a treap is a BST constructed by inserting nodes in increasing order of priorities, using standard BST insertion.

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).

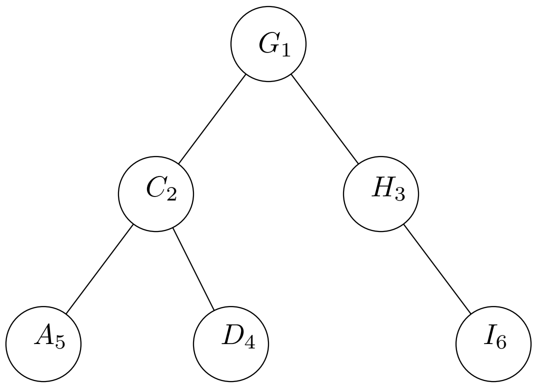

Figure 1: A treap with each node K_P having key $= K$ and priority $= P$

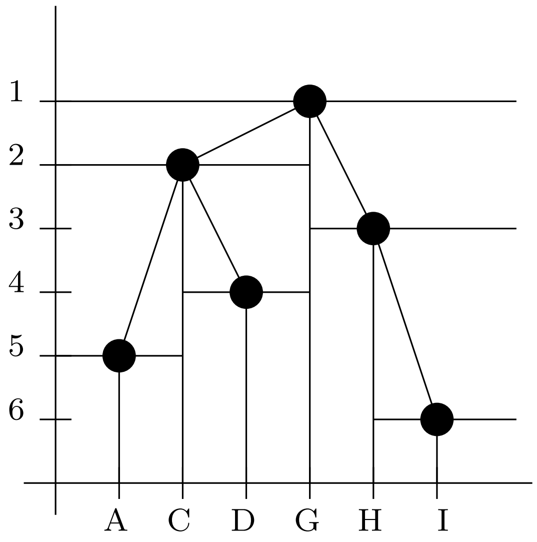

Figure 2: A Cartesian tree representation of the treap

2.2 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.

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 1

```
TREAP-INSERT($T, v$)
    BST-INSERT($T, v$)
    while $v.priority < v.parent.priority$
        ROTATE($v$)
```

Algorithm 2

```
function TREAP-DELETE($v$)
    TREAP-INCREASE-PRIORITY($v, \infty$)
    DELETE($v$)
```

```
function TREAP-INCREASE-PRIORITY($v, new\_priority$)
    $v.priority = new\_priority$
    if $v.left \neq nil$ & ($v.right \neq nil$ & $v.right.priority > v.left.priority$)
        $min = v.left$
    else
        $min = v.right$
    while $min \neq nil$
        ROTATE($min$)
        if $v.left \neq nil$ & ($v.right \neq nil$ & $v.right.priority > v.left.priority$)
            $min = v.left$
        else
            $min = v.right$
```

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

Algorithm 3 Splits the treap T around key k

```
function TREAP-SPLIT($T, k$)
    $v \leftarrow$ new Node($k, -\infty$)       ▷ Create a new node with key $k$ and priority $-\infty$
    TREAP-INSERT($T, v$)                ▷ $v$ will be the root, because $v.priority = -\infty$
    $T_< \leftarrow v.left$
    $T_> \leftarrow v.right$
    DELETE($v$)
    return ($T_<, T_>$)
```

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

Algorithm 4

```
1: function TREAP-MERGE($T_<, T_>$)
2:     $v \leftarrow$ new Node$\left(\frac{\max(T_<)+\min(T_>)}{2}, -\infty\right)$    ▷ New key is between all keys in $T_<$ and in $T_>$
3:     $v.left \leftarrow T_<.root$
4:     $v.right \leftarrow T_>.root$
5:     $T \leftarrow v$                                                 ▷ $v$ is the new root of $T$
6:     TREAP-DELETE($v$)
7:     return $T$
```

2.3 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 in the worse case, we can use randomization to ensure balance. In fact, the definition of a treap is a BST, where the priorities of the nodes are radomly chosen.* When priorities are randomly chosen from a uniform distribution, the result is a tree with the expected height of $O(\log n)$, as we will show next.

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 random variable is equal to the probability of the variable being equal to 1:

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

Now let us define an ordered 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.

*The y-coordinate of the points in the Cartesian tree are not random, but are given as part of the input.

Lemma 1. *$Y_{ij} = 1$ (x_i is a proper ancestor of x_j) 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 $\{x_i, \dots, x_k\}$ 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 $O(\log n)$ expected time.

3 Skip lists

Introduced by William Pugh[†] 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, just like treaps, we will show that the performance of skip lists is as good as balanced BSTs in expectation, but may exhibit worse performance in the worst cases (but as we show this is extremely unlikely).

3.1 Intuition

A skip list, can be views intuitively, as multiple train lines, where an express train line skips some stops on the way, compared to a normal train line that stops at each location. To reach a destination, one takes the express train to the last reachable stop before a destination on the express train line, and then switches to a normal train for the rest of the trip if needed.

Given a sorted set $X_1 = \{x_0, \ldots, x_{n-1}\}$, consider its subset of every k^{th} element:

$$X_2 = \{x_0, x_k, x_{2k}, \ldots, x_{\ell k}\} \subseteq X_1, \text{ where } \ell = \lfloor (n-1)/k \rfloor$$

We store the two sets as linked lists. Moreover, for every $x_i \in X_2$, we create a pointer from every $x_i \in X_2$, to the corresponding $x_{ik} \in X_1$. Note that between every pair of neighboring elements x_i and x_{i+1} in X_2, there are $k-1$ elements in X_1. An example of the two lists for $k = 2$ are shown in Figure 3.

To search for an element with key x, we traverse the second linked list X_2 until we reach a node x_i whose neighbor x_{i+1} is larger than x. Next, by following the (down) pointer from x_i to $x_{ik} \in X_1$, we continue the search for x in X_1.

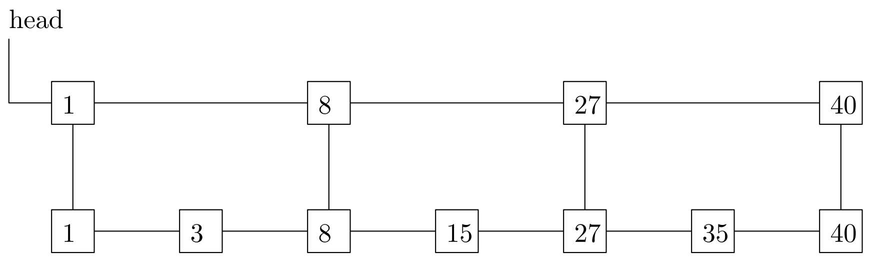

Figure 3: Skip list with two levels and $k = 2$.

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

3.2 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 $2\sqrt{n}$.

3.3 Increasing the number of lists

In similar way, we can define $X_3 = \{x_0, x_{k'}, x_{2k'}, \ldots, x_{\ell' k'}\}$ where $\ell' = \lfloor \frac{\ell+1}{k'} \rfloor$ on top of X_2, and then the cost function $f(k, k')$ is given by $f(k, k') = \ell' + k' + k \leq \frac{n}{kk'} + k' + k$. To calculate the minima, we take the derivatives with respect to both k and k' to get the following:

$$\begin{aligned} \frac{df}{dk} &= -\frac{n}{k'k^2} + 1 = 0 && \Longrightarrow k'k^2 = n \\ \frac{df}{dk'} &= -\frac{n}{kk'^2} + 1 = 0 && \Longrightarrow kk'^2 = n \end{aligned}$$

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 $3\sqrt[3]{n}$. We can generalize this approach to t linked lists. By doing a similar analysis for 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 search time of $2 \log n$.

Algorithm 5

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

3.4 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. Instead of maintaining a strict distance between elements of neighboring linked lists, we use randomization. To decide whether to include an element in the list above, we flip a fair coin. If the coin comes up HEADS, we promote a copy of the element to the next level, stopping at the first occurence of TAILS.

We still need to provide an analysis of why this produces an efficient version of the skip list, and for that, we calculate the expected maximum level τ that any element $x \in X_1$ will reach, i.e., that $x \in X_\tau$.

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

$$\Pr[H \le (c+1)\log n] \ge 1 - \frac{1}{n^c}$$

for any constant $c > 0$.

Proof. Let $H(x)$ denote the highest level that element x reaches. For $H(x)$ to be greater than τ, every one of the first τ tosses of the coin must be HEADS. The probability of first τ tosses to be HEADS is $1/2^\tau$. Hence, $\Pr[H(x) > \tau] = 1/2^\tau$.

The height H of the skip list is the height of the maximum $H(x)$ among all $x \in X_1$:

$$\begin{aligned}
\Pr[H > \tau] &= \Pr[\max_{x \in X_1}\{H(x)\} > \tau] \\
&= \Pr[H(x_1) > \tau \text{ or } H(x_2) > \tau \cdots \text{ or } H(x_n) > \tau] \\
&\le \sum_{x \in X_1} \Pr[H(x) > \tau] && \text{(by the Union Bound)} \\
&= \sum_{x \in X_1} \frac{1}{2^\tau} \\
&= \sum_{i=1}^{n} \frac{1}{2^\tau} && \text{(there are n keys in X_1)} \\
&= \frac{n}{2^\tau}
\end{aligned}$$

Choosing $\tau = (c+1)\log n$ gives us:

$$\Pr[H > (c+1)\log n] \le \frac{n}{2^{(c+1)\log n}} = \frac{n}{n^{c+1}} = \frac{1}{n^c}$$

The event $H \le (c+1)\log n$ is the opposite of $H > (c+1)\log n$. Therefore:

$$\Pr[H \le (c+1)\log n] = 1 - \Pr[H > (c+1)\log n] \ge 1 - \frac{1}{n^c}.$$ □

Algorithm 6

```
1: function INSERT($k, S$)
2:     $v \leftarrow$ SEARCH($k, S$)                      ▷ the predecessor node at the bottom level
3:     if $v.key \neq k$
4:         LISTINSERT($v$, new $Node(k)$)      ▷ Insert a new node with key $k$ into the list after $v$
5:         $bit \leftarrow$ COINFLIP()
6:         while $bit ==$ HEADS
7:             promote a copy of $v$ to the level above
8:             $bit \leftarrow$ COINFLIP()
```

References

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