ICS 621: Analysis of Algorithms Fall 2021

Lecture 6

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

1 Overview (BST)

Previously, we covered Dynamic Programming (DP), often performed in a step by step process starting with thinking about the brute force solution, defining a recursive solution to identify the optimal subproblems, and finally by adding the solved problems to a memo table to lookup, thus reducing wasted effort resolving problems over and over again. We looked at DP in the context of specific examples including Subset Sum, Knapsack, and Longest Increasing Subsequence. In addition, we introduced two approaches in DP: top-down and bottom-up and elaborated how they differ from each other in terms of implementation while their time complexity is usually of the same order.

In this lecture, we will be introducing 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 will introduce 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 will be discussing how rotations are made in pairs, dependent on the structure of the access path. Lastly, we will analyze the Amortized Cost of each Splay Operation using a potential function.

2 Optimal Binary Search Tree

The problem of a Optimal Binary Search Tree can be rephrased as:

Given a list of n keys ($A[1, ..., n]$) and their frequencies of access ($F[1, ..., n]$), construct a optimal binary search tree in which the cost of search is minimum.

We will start with a list of keys in a tree and their frequencies. Now to find the best binary tree with minimum cost to search in the tree, we can create all possible binary trees out of our list of keys and find the cost of searching items in those binary trees and pick one of the best binary tree as our optimal binary search tree.

Further we can use divide and conquer strategy to solve our problem. Note that if we have an optimal binary tree then at any node of that tree, it should have an optimal binary search tree on its left child tree and also an optimal binary search tree on its right child tree. Thus we can break our original problem into two sub problems and use same method to solve for those sub problems.

Our objective for this problem is to find the tree with minimum cost for searching. So we can use the cost of searching keys in the tree as our objective function for our recursive function. And before we move on to our recursive solution, let's look at the cost of searching a binary search tree in detail.

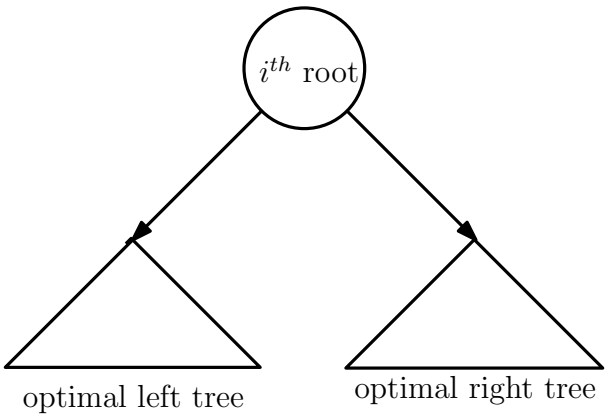

Figure 1: Dividing Optimal Binary Search Tree problem to sub problems

2.1 OptBST Cost Analysis

Given any binary search tree T, the total cost of searching can be calculated by calculating sum of product of frequencies of the nodes ($f[v]$) and depth of the nodes in the tree ($d[v]$).

$$cost_{total} = \sum_{v \in T} f[v] \cdot d[v] \tag{1}$$

Thus, the problem of finding the cost of searching the binary search tree can be solved recursively as follows (see Figure 1) for reference).

For the left optimal tree (T_{left}), the cost of searching in the left optimal subtree is given by:

$$cost_{left} = \sum_{v \in T_{left}} f[v] \cdot d[v] \tag{2}$$

Similarly for right optimal subtree (T_{right}), the cost of searching in the right optimal subtree is given by:

$$cost_{right} = \sum_{v \in T_{right}} f[v] \cdot d[v] \tag{3}$$

Now, if we combine left and right optimal trees with the root node just like in Figure 1, the depth for nodes in left and right subtree are increased by 1 and the updated cost for left and right subtree is given by:

$$
\begin{aligned}
cost_{left\_new} &= \sum_{v \in T_{left}} f[v] \cdot (d[v] + 1) \\
&= \sum_{v \in T_{left}} f[v] \cdot d[v] + \sum_{v \in T_{left}} f[v] \\
&= cost_{left} + \sum_{v \in T_{left}} f[v]
\end{aligned}
$$

$$
\begin{aligned}
cost_{right\_new} &= \sum_{v \in T_{right}} f[v] \cdot (d[v] + 1) \\
&= cost_{right} + \sum_{v \in T_{right}} f[v]
\end{aligned}
$$

And, the final cost is sum of the cost of search on right, cost of search on left and the number of times we search for the root node (equal to the frequency of the key in the root node).

$$
\begin{aligned}
cost_{total} &= cost_{left\_new} + cost_{right\_new} + f[root] \\
&= \left(cost_{left} + \sum_{v \in T_{left}} f[v] \right) + \left(cost_{right} + \sum_{v \in T_{right}} f[v] \right) + f[root] \\
&= cost_{left} + cost_{right} + \left(\sum_{v \in T_{left}} f[v] + \sum_{v \in T_{right}} f[v] + f[root] \right)
\end{aligned}
$$

For our implementation with array of keys and frequencies, if we sort the keys and frequencies based on keys then we can say that if we pick any of the array element as a root node for our tree, the sub-array in the left of the key can be used as nodes for left binary sub-tree and the sub-array in the right of the key can be used as nodes for right sub-tree. For example if we want to create tree out of array that is bound by indices L and R with root node at index i, the cost of searching in the tree can be obtained using:

$$
\begin{aligned}
cost_{total} &= \sum_{v=L}^{i-1} f[v] \cdot d[v] + \sum_{v=i+1}^{R} f[v] \cdot d[v] + \left(\sum_{v=L}^{i-1} f[v] + \sum_{v=i+1}^{R} f[v] + f[i] \right) \\
&= \sum_{v=L}^{i-1} f[v] \cdot d[v] + \sum_{v=i+1}^{R} f[v] \cdot d[v] + \sum_{v=L}^{R} f[v] \\
&= cost_{left} + cost_{right} + \sum_{v=L}^{R} f[v]
\end{aligned}
$$

So using the cost function we can implement recursive solution for the problem as:

Algorithm 1 MAIN

```
1: Sort A[1..n] and F[1..n] by the keys
2: return findOptimalCost(A, F, 1, n)
```

Algorithm 2 findOptimalCost(A, F, $left$, $right$)

```
1: best = +∞
2: for i = left to right                                   ▷ try each as the root
3:     cost_left = findOptimalCost(A, F, left, i − 1)
4:     cost_right = findOptimalCost(A, F, i + 1, right)
5:     cost_total = cost_left + cost_right + Σ_{v=left}^{right} F[v]
6:     if cost_total < cost_best
7:         best = cost_total
8: return best
```

2.2 OptBST Runtime Analysis: Dynamic Programming

In interest of saving time, we will summarize the process of finding a dynamic programming solution and leave the actual solution up to the students as an exercise.

To convert our current Optimal Binary Search Tree into a dynamic programming solution, we use the memoization table. First find what values the table relies on for every entry l and r. For a bottom-up solution, notice our memoization table relies on $(l, l-1)$ and $(l+1, r)$, (l, l) and $(l+2, r)$ ..., etc. The size our table will be $(N \cdot N)$ and for every entry that we fill in, we need a loop from l to r. Which in the worst case will be order of n. Therefore we will have a cubic time solution if we used dynamic programming.

$$= (\sum + \text{OPTBST}(l, l) + \text{OPTBST}(l + l, r), (\sum + \text{OPTBST}(l, l + 1) + \text{OPTBST}(l + 2, r), ...$$
$$= O(n^3)$$

Exercise: Find a complete dynamic Programming Solution for OptimalBST.

Refer to the class textbook for more information [1].

3 Splay Trees

SPLAY TREES, detailed in [9] are self-adjusting BSTs that:

- implement dictionary APIs,
- are simply structured with limited overhead,

- are c-competitive * with best offline BST, i.e., when all queries are known in advance,
- are conjectured to be c-competitive with the best online BST, i.e., when future queries are not know in advance

3.1 Splay Tree Operations

Simply, SPLAY TREES are BSTs that move search targets to the root of the tree. This is accomplished by 3 simple rotate rules that are applied repeatedly until the search target becomes the root.

Algorithm 3

```
1: function SEARCH(k)
2:     x = BST-FIND(k)
3:     while x ≠ root
4:         SPLAY(x)
```

The 3 rotate rules are ZIG, ZIG-ZAG, and ZIG-ZIG, and are detailed below:

3.1.1 Zig

The ZIG operation is applied if the element being moved, x, is the child of the root, as seen in Figure 2. When this is the case, the element, x, is simply rotated with the root node, making it the new root.

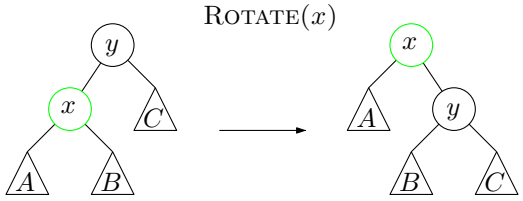

Figure 2: A case where the ZIG operation is used to move target node x to the root. This is only used when the target node is the direct child of the root node.

3.1.2 Zig-zag

The ZIG-ZAG operation is applied when the target element being moved, x, needs to be moved up in alternate directions (clockwise and anti-clockwise), as illustrated in Figure 3. When a node being moved up the tree falls on the opposite direction of its parent as its parent falls from its grandparent, we employ the ZIG-ZAG operation. The ZIG-ZAG operation involves rotating x with its parent (y), and then rotating again with its new parent (z).

*An algorithm $\mathbb{A}$ is c-competitive for a problem $\mathcal{P}$ if there exists a constant $\alpha \geq 0$ such that $time(\mathbb{A}) \leq c \cdot time(\mathbb{O}) + \alpha$ where $\mathbb{O}$ is the optimal solution for $\mathcal{P}$.

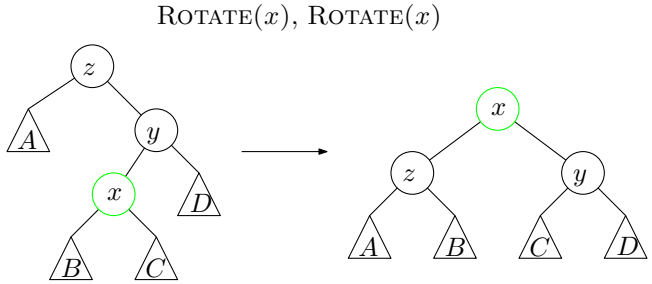

Figure 3: A case where the ZIG-ZAG operation is employed to move target node x up the tree. The ZIG-ZAG operation would also be used in the symmetric case (i.e., where x is the right child of y and y is the left child of z).

3.1.3 Zig-zig

The final operation, ZIG-ZIG, is used when both the node being moved up falls on the *same* spine of its parent as its grandparent. When this occurs, we first perform *Rotate* on the targets parent node, followed by *Rotate* on the target node itself. Figure 4 illustrates the result of the ZIG-ZIG operation on node x.

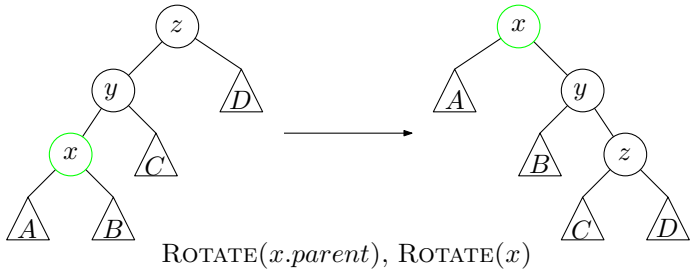

Figure 4: An example of how the ZIG-ZIG operation is performed by a SPLAY TREE. First, y is rotated upwards, then x is rotated up to the root. This is only employed when both the target (x) and its parent y fall on the same spine of their respective parent nodes.

3.2 Splay Tree Example

We consider the worst-case initial BST and look at how the Splay tree and its operations would perform. Clearly, our worst-case BST would be a tree where each node has only 1 child, and the other is NULL. In this case, searching for the leaf node would result in an $O(n)$ time search. However, using a Splay tree, each search would result in the SPLAY operation moving the searched (leaf) node to the root, thereby moving lots of other elements. Figure 5 illustrates how a Splay tree would perform when repeatedly searching such a target.

We see in Figure 5 that, after just 3 searches on a worst-case tree, we have a tree that is mostly balanced. All subsequent searches will be $O(\log n)$, and would further adjust the tree as needed. This example shows how SPLAY TREES quickly self-balance and achieve optimal or near-optimal performance quickly. Amortized over a large number of queries, the cost of searching (and SPLAYING a node to the root) is $O(\log n)$.

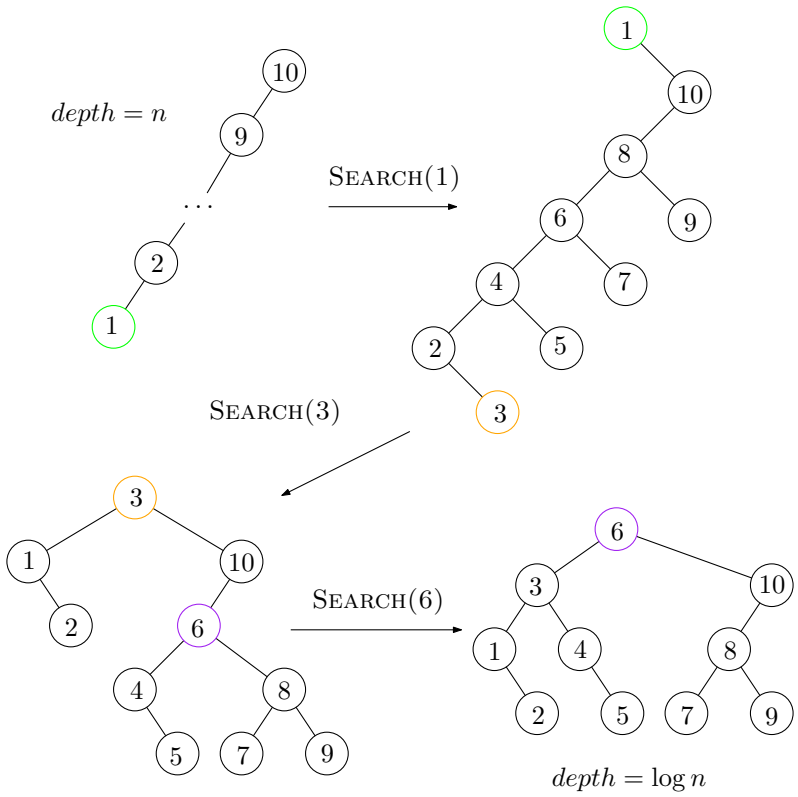

Figure 5: An example of a worst-case tree and result of repeatedly searching (and using the SPLAY operation) it. We see that, after searching just 3 times, we have a tree of optimal height for the input size.

3.3 Amortized Analysis

We will use the potential method of amortized analysis but first we define several variables:

- $w(x)$ be an arbitrary weight associated with each node x. In the Splay tree context we'll define $w(x) \leq 1$ for all x.
- $s(x)$ be the sum of all weights located in subtree $T(x)$ rooted at x (including itself), i.e., $s(x) = \sum_{v \in T(x)} w(v)$.
- $r(x) = \log s(x)$ called the rank of a node x.

Note that if we set $w(x) = 1$ for all x, then $s(x)$ is just the number of nodes in the subtree and $r(x)$ is the number of levels of (a balanced) subtree rooted at x.

We define the potential function of a Splay tree as follows:

$$\Phi(T_i) = \sum_{x \in T_i} r(x),$$

where T_i is the entire Splay tree structure after the i-th operation. To perform amortized analysis, we analyse the three tree operations (zig, zig-zag, and zig-zig) defined in Section 3.1. Recall that the amortized cost using the potential method is defined as:

$$\hat{c_i} = c_i + \Delta\Phi_i$$

3.3.1 Zig-zig analysis

Consider the ZIG-ZIG operation defined in Section 3.1 and the 2 rotations it involves. The cost of performing a rotation is 1, and since we have two rotations we have:

$$\hat{c}_i = 2 + \Delta\Phi_i = 2 + (\Phi(T_i) - \Phi(T_{i-1}))$$

Where $\Phi(T_{i-1})$ and $\Phi(T_i)$ are the potential function values before and after performing the ZIG-ZIG operation. We see in Figure 4 that the potential of subtrees A,B,C, and D don't change, so they cancel out in $\Delta\Phi_i$ and we only need to consider the 3 nodes involved in the rotations: the target node x, its parent y, and its grandparent z. Therefore, we expand the above formula to the changes in potential for each of the nodes x, y, and z:

$$\hat{c}_i(x) = 2 + r_i(x) + r_i(y) + r_i(z) - r_{i-1}(x) - r_{i-1}(y) - r_{i-1}(z)$$

Observe the following inequalities from Figure 4:

$$\begin{aligned}
r_{i-1}(z) &= r_i(x) && \text{(because of the symmetry)} \\
r_i(y) &\le r_i(x) && (T_i(y) \text{ is a subtree of } T_i(x)) \\
r_{i-1}(y) &\ge r_{i-1}(x) && (T_{i-1}(x) \text{ is a subtree of } T_{i-1}(y)) \\
s_i(z) + s_{i-1}(x) &\le s_i(x) && \big(\underbrace{(s(C) + s(D) + w(z))}_{s_i(z)} + \underbrace{s(A) + s(B) + w(x)}_{s_{i-1}(x)} + w(y) = s_i(x)\big)
\end{aligned}$$

Therefore, the amortized cost is then:

$$\hat{c}_i(x) \le 2 + r_i(x) + r_i(z) - r_{i-1}(x) - r_{i-1}(x).$$

Since the log function is monotone[†] and convex[‡] we have $\frac{\log a + \log b}{2} \le \log \frac{a+b}{2}$:

$$\begin{aligned}
r_i(z) + r_{i-1}(x) &= \log s_i(z) + \log s_{i-1}(x) \\
&\le 2 \log \frac{s_i(z) + s_{i-1}(x)}{2} \\
&\le 2(\log s_i(x) - \log 2) && (s_i(z) + s_{i-1}(x) \le s_i(x)) \\
&= 2(r_i(x) - 1).
\end{aligned}$$

Hence, $r_i(z) \le 2r_i(x) - 2 - r_{i-1}(x)$. Using this simplification, the amortized cost of a zig-zig operation becomes:

$$\begin{aligned}
\hat{c}_i(x) &\le 2 + r_i(x) + \quad r_i(z) \quad - r_{i-1}(x) - r_{i-1}(x) \\
&\le 2 + r_i(x) + (2r_i(x) - 2 - r_{i-1}(x)) - r_{i-1}(x) - r_{i-1}(x) \\
&= 3(r_i(x) - r_{i-1}(x)).
\end{aligned}$$

3.3.2 ZigZag analysis

We can use a similar analysis to show that the ZIG-ZAG operation has amortized cost $\hat{c}_i(x) \le 3(r_i(x) - r_{i-1}(x))$ by replacing terms of y and z into inequalities for x.

[†] $f\colon \mathbb{R} \to \mathbb{R}$ is monotone if $a \le b \implies f(a) \le f(b)$.

[‡] $f\colon \mathbb{R} \to \mathbb{R}$ is convex if $a < b \implies \frac{f(a)+f(b)}{2} < f(\frac{a+b}{2})$.

3.3.3 Zig analysis

The Zig operation has only a single rotation, and hence the amortized cost of the ZIG operation is given by:

$$\begin{aligned}\hat{c}_i(x) &= 1 + \Delta\Phi_i \\ &= 1 + r_i(x) + r_i(y) - r_{i-1}(x) - r_{i-1}(y)\end{aligned}$$

where x is our target node being moved up and y is its parent (and the root node of the tree). We note from Figure 2 that:

$$\begin{aligned} r_i(y) &\leq r_i(x) && (T_i(y) \text{ is a subtree of } T_i(x)) \\ r_{i-1}(y) &= r_i(x) && (\text{by symmetry}) \end{aligned}$$

Therefore, we can simplify the amortized cost to:

$$\begin{aligned}\hat{c}_i(x) &\leq 1 + r_i(x) + r_i(x) - r_{i-1}(x) - r_i(x) \\ &= 1 + r_i(x) - r_{i-1}(x)\end{aligned}$$

And because $r_i(x) \geq r_{i-1}(x)$, it follows that $\hat{c}_i(x) \leq 1 + 3(r_i(x) - r_{i-1}(x))$

3.3.4 Analysis of moving a node to the root

With the bound on the amortized cost for the three operations above, we can bound the total cost to move a node x from its position to the root through, let's say, k splay operations as:

$$\begin{aligned}\hat{c}(x) &= \sum_{i=1}^{k} \hat{c}_i(x) && (k \leq \text{height}(x)/2 + 1) \\ &\leq \underbrace{\hat{c_k}(x)}_{\text{possible zig}} + \underbrace{\sum_{i=1}^{k-1} \hat{c}_i(x)}_{\text{possible zig-zig or zig-zag}} .\end{aligned}$$

Since we perform either ZIG-ZAG or ZIG-ZIG operations for all steps from $i = 1$ to $k-1$ and at most one final ZIG operation to move x to the root, the amortized is upper bounded by

$$\begin{aligned}\hat{c}(x) &\leq 1 + 3(r_k(x) - r_{k-1}(x)) + \sum_{i=1}^{k-1} 3(r_i(x) - r_{i-1}(x)) \\ &\leq 1 + \sum_{i=1}^{k} 3(r_i(x) - r_{i-1}(x)) \\ &\leq 1 + 3(r_k(x) - r_0(x)) && (\text{telescoping series}) \\ &\leq 1 + 3(\log s_{root} - \log s_0(x)) \\ &= 1 + 3\frac{\log s_{root}}{s_0(x)} \\ &= O\left(\log \frac{s_{root}}{s_0(x)}\right).\end{aligned}$$

Theorem 1. *The amortized cost of searching a* SPLAY *tree is* $O(\log n)$.

Proof. Setting the weight of each node x to be $w(x) = 1$, we get $s_{root} = n$, and $s_0(x) \geq 1$ (s_0 contains at least the node x itself). Hence, the amortized cost of a Splay search operation is then bounded by:

$$\hat{c}(x) \leq O(\log(n/1)) = O(\log n). \qquad \square$$

Theorem 2. *(Static optimality) Given a sequence Q of queries with known access frequencies $f(x_i)$, Splay trees are $O(1)$-competitive with the best offline* BST *for Q.*

Proof. Let $w(x) = p(x) = \frac{f(x)}{m}$, therefore the amortized cost is:

$$\hat{c}_Q = \sum_{x \in Q} \hat{c}(x) = \sum_{x \in Q} O\left(\log \frac{s_{root}}{s(x)}\right)$$

Where Q is the sequence of queries being performed. Note that, since s_{root} is the sum of weights on all nodes:

$$s_{root} = \sum_{x \in X} w(x) = \sum_{x \in X} p(x) = \sum_{x \in X} \frac{f(x)}{m} = 1.$$

Furthermore, since $s(x)$ includes the weight of x, $s(x) = \sum_{v \in T(x)} w(v) \geq p(x)$. Using these, the amortized cost can be written as:

$$\begin{aligned}
\hat{c}_Q &\leq \sum_{x \in Q} O\left(\log \frac{1}{s(x)}\right) \leq \sum_{x \in Q} O\left(\log \frac{1}{p(x)}\right) \\
&= O\left(\sum_{i=1}^{n} f(x) \log \frac{1}{p(x_i)}\right) \\
&= O\left(m \cdot \sum p(x) \cdot \log \frac{1}{p(x)}\right) \\
&= O(m \cdot H_n) \\
&\leq c' \cdot m \cdot H_n) && \text{for some constant } c'
\end{aligned}$$

where H_n is the entropy of the sequence of queries Q on n distinct keys. Recall from Section 1 that the cost of m queries on the optimal static BST is $C_m = \Theta(m \cdot H_n)$, i.e., $C_m = \Omega(m \cdot H_n)$ or, equivalently, $C_m \geq \bar{c} \cdot m \cdot H_n$ for some constant $\bar{c}$. Then

$$\hat{c}_Q \leq c' \cdot m \cdot H_n \leq \frac{c'}{\bar{c}} \cdot C_m = O(1) \cdot C_m$$

$\square$

I.e., the time to execute a sequence Q of queries on splay trees is c-competitive with executing this sequence of queries on the optimal BST, for some constant $c = c'/\bar{c}$.

Algorithm 4

```
1: function INSERT(k)
2:     v = BST-INSERT(k)
3:     while v ≠ root
4:         SPLAY(v)
```

Algorithm 5

```
1: function DELETE(k)
2:     v = BST-DELETE(k)
```

3.4 Other Splay tree operations

Insertion (Algorithm 4) and deletions call their respective BST operations followed by splaying.

The amortized cost of insertion to a Splay tree is just the sum of amortized cost of an insertion into a BST plus the cost of sequence of Splay operations up to the root. Using Theorem 1 we get $O(\log n)$ amortized for insertion. Deletion is just a normal BST deletion which takes $O(\log n)$ amortized over a sequence of insertions and queries.

References

[1] T. Cormen, C. Lieserson, R. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, third edition, 2009.

[2] Thomas M. Cover and Joy A. Thomas. *Elements of information theory (2. ed.).* Wiley, 2006.

[3] Michael Fredman, Robert Sedgewick, Daniel Sleator, and Robert Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1:111–129, 11 1986.

[4] D. E. Knuth. Optimum binary search trees. *Acta Inf.*, 1(1):14–25, March 1971.

[5] Donald Ervin Knuth. *The art of computer programming, , Volume III, 2nd Edition.* Addison-Wesley, 1998.

[6] Kurt Mehlhorn. Nearly optimal binary search trees. *Acta Inf.*, 5:287–295, 1975.

[7] Kurt Mehlhorn. Nearly optimal binary search trees. *Acta Inf.*, 5:287–295, 1975.

[8] D.D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[9] D.D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.