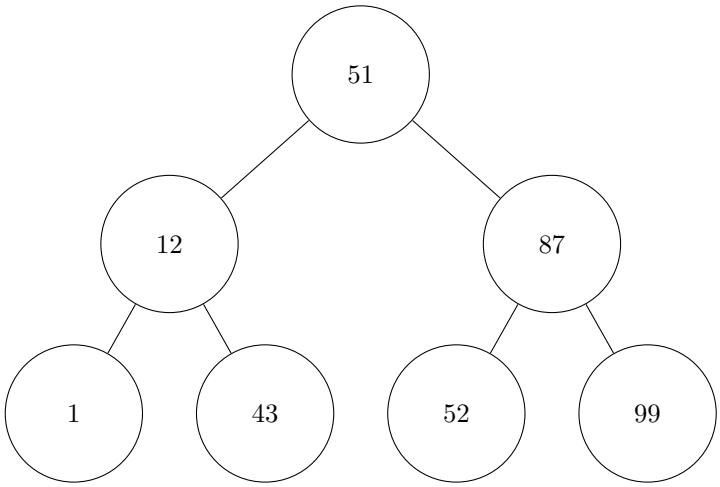
Splay Trees

Janosch Ruff

December 20, 2020

1 Binary Search Tree Model of Computation



- Walk to the left node (smaller than parent)
- Walk to the right node (greater than parent)
- Walk to the parent of the node
- Left/right rotation

↪ All operations in constant time $\mathcal{O}(1)$

2 Balanced Binary Search Tree

- Operations on BST: search(x), insert(x), delete(x) $\in \mathcal{O}(h)$.
- Rotations to minimize the height of the tree to get worst-case $\mathcal{O}(\log n)$.
- Approaches of balanced BST: Red-black tree, AVL tree and Treap i.a.
- General best case depth $\Omega(\log n)$ cannot be further improved.

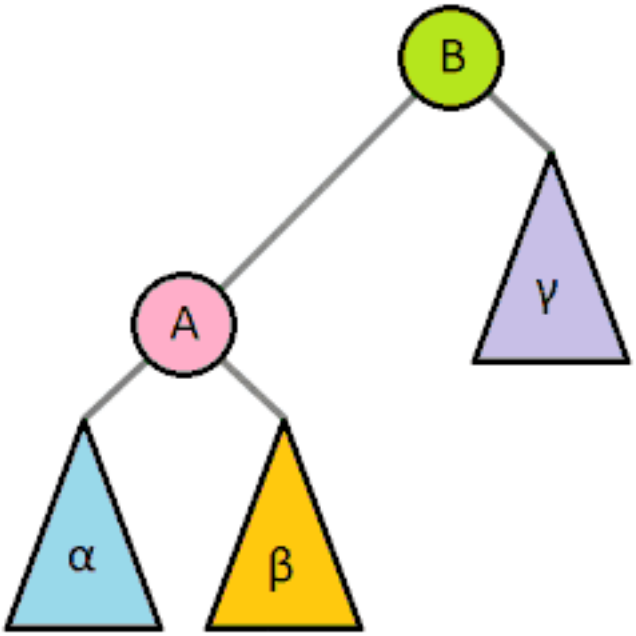


Figure 1: Left rotated binary search tree

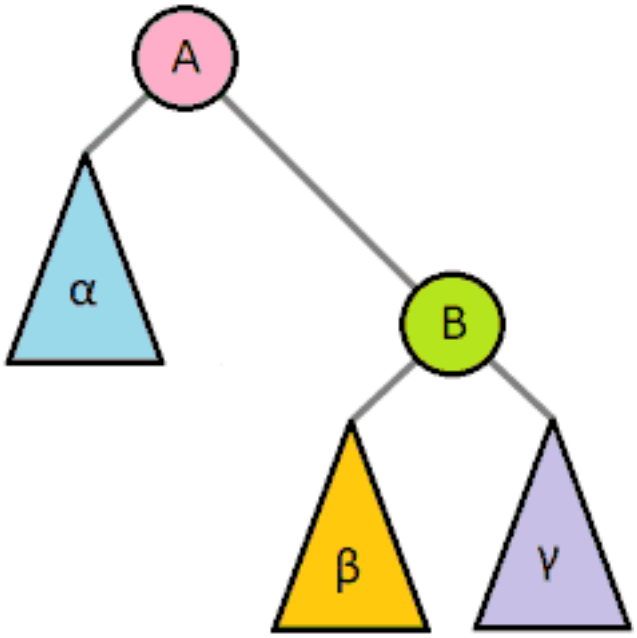


Figure 2: Right rotated binary search tree

3 Static Optimal Binary Search Trees

Problem:

- Universe of keys $k \in \{1, 2, .., u\}$ with weights $w \in \mathbb{R}^+$.
- Minimize the expected number of recursions to access n elements given by

$$\min_{bst \in BST} = \sum_{k=1}^{u} w_k \cdot (depth_k + 1) \tag{1}$$

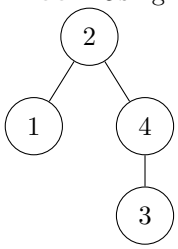
- Toy example: Keys {1,2,3,4} with weights $\{1, 10, 8, 9\}$ accordingly.
- First approach: Greedy algorithm inserting key with heighest weight into tree

Algorithm 1 GreedyBST(keys, weights)

```
BST ← empty Binary Search Tree
keys ← keys.sortedby(weights)
k ← |keys|
while k > 0 do
    MaximalWeight ← keys_k
    BST ← BST ∪ MaximalWeight
    weights ← weights ∖ MaximalWeight
    k ← k − 1
end while
```

- The greedy approach runs in $\mathcal{O}(u \log u + u)$ time using Quicksort hence, GreedyBST $\in \mathcal{O}(n \log n)$ where n is the number of keys.

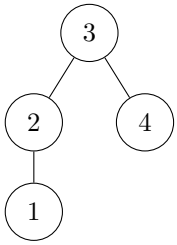
Lemma: The Greedy Algorithm does not always construct an optimal BST.
Proof: Using the greedy algorithm on our toy example we get the tree:



Calculating with our cost function 1 the for expected number of recursions we get

$$cost(GreedyBST) = 10 \cdot 1 + (1 + 9) \cdot 2 + 8 \cdot 3 = 54 \tag{2}$$

However, we can construct the following tree with a better expected number of recursions for such weights:



Again using our cost function 1 for the expected number of recursions we get

$$cost(OptimalBST) = 8 \cdot 1 + (10 + 9) \cdot 2 + 1 \cdot 3 = 49 \tag{3}$$

this shows there exists a more optimal binary search tree for the toy example.■

- Since it is an optimization problem minimizing a cost function with overlapping subproblems dynamic programming solves this problem optimal.
- Next: Create a binary search tree solving the problem without knowing the weights beforehand and those generically for all weights.

4 Splay Trees

4.1 Splay Heuristic

- Splay trees use splay heuristic for search(x), insert(x) and delete(x).
- search(x): standard BST.search(x), BST.splay(x)

Definition: The splay subroutine is defined by

$$splayrot(x) := \begin{cases} \text{rotate(y), rotate(x)} & \text{if (z.left = y and y.left = x) or (z.right = y and y.right = x)} \\ \text{rotate(x), rotate(x)} & \text{if (z.left = y and y.right = x) or (z.right = y and y.left = x)} \\ \text{rotate(x)} & \text{otherwise.} \end{cases} \tag{4}$$

where y is the parent of x and z is the grandparent of x.

- The first case of splay is called Zig-Zig visualized in Figure 3.
- The second case of splay is called Zig-Zag visualized in Figure 4.
- The third case of splay describes a single rotation used if x is one rotation short of being the new root.

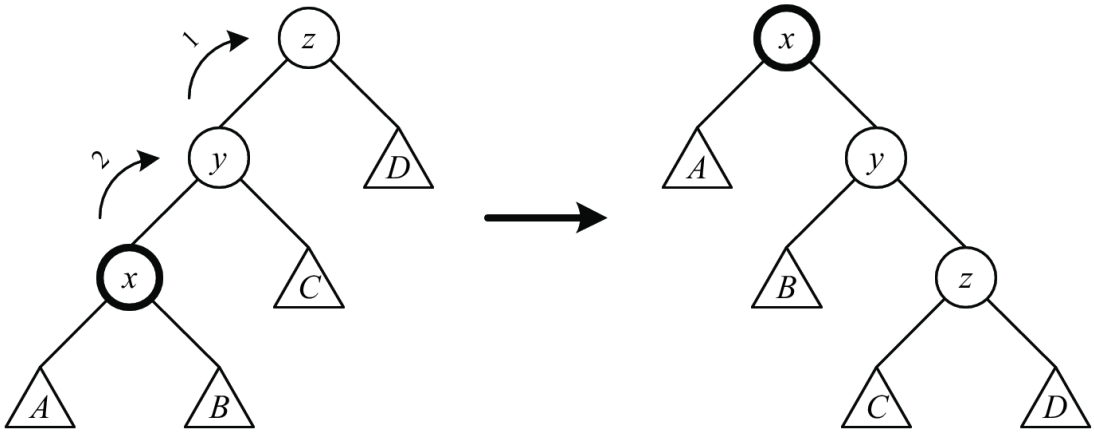


Figure 3: Zig-Zig Operation

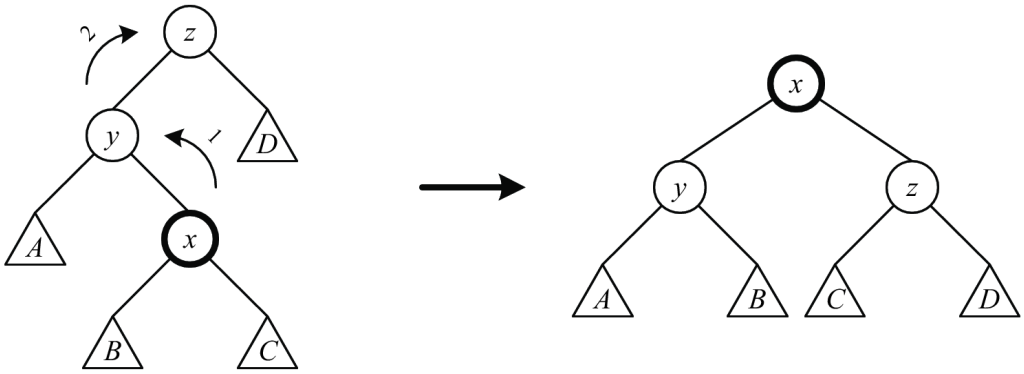


Figure 4: Zig-Zag Operation

Algorithm 2 T.splay(x)

```
1: T.splayrot(x)
2: if x.hasParent then
3:     T.splay(x)
4: end if
```

- One rotation $\in \mathcal{O}(1) \Longrightarrow$ one splay rotation $\in \mathcal{O}(1)$.
- To get the number of recursions we need the Access Lemma.

4.2 Access Lemma

Access Lemma Amortized time to splay a node x given root t is at most

$$3(r(t) - r(x)) + 1 = \mathcal{O}(\log \frac{s(t)}{s(x)}). \tag{5}$$

- Size $s(x) = \sum_D w_k$ where the set D consists all descendants of x.
- Rank $r(x) = \log s(x)$
- Potential $\Phi = \sum r(x)$

Proof: We analyze one Zig-Zig rotation. The cost for Zig-Zag rotations immediately follows from that since, Zig-Zig is the harder case. Further we argue that from one rotation

$$\text{rotation cost} = 3(r'(x) - r(x)) \tag{6}$$

(where $r'(x)$ and $s'(x)$ denotes r(x) and s(x) after the first double rotation), from one rotation we can telescope the total cost by

$$\sum \text{cost} = 3(r'(x) - r(x)) + 3(r''(x) - r'(x)) + .. + 3(r^{\text{final}}(x) - r^{\text{final-1}}(x)) \tag{7}$$

$$= 3(r^{\text{final}}(x) - r^{\text{initial}}(x)) \tag{8}$$

$$= 3(r(t) - r(x)) \tag{9}$$

where we have to do the amount operations until we splay x to the root t. Since the final splay rotates x to the root follows that $r^{final} = r(x)$. Finally the + 1 in our bound 5 of the lemma comes from the final single zig operation which might be necessary.
So which nodes are changing ranks in one Zig-Zig operation? The ones of x,y and z such that we get:

$$\Delta\Phi = r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) \tag{10}$$

while the real cost is 2 such that we get for the amortized cost is given by

$$r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) + 2 \tag{11}$$

Since r'(x) = r(z) (check figure 3) these terms cancel each other out.
Further we notice that $r'(x) > r'(y)$ and $r(x) < r(y)$ (again check figure 3 to see this) we can upper bound the amortized cost by replacing the terms

$$\leq 2 + r'(x) + r'(z) - r(x) - r(x) \tag{12}$$

$$= 2 + (r'(z) - r(x)) + \underbrace{(r'(x) - r(x))}_{\varphi} \tag{13}$$

Notice that φ is what we want for our claimed amortized cost just multiplied by 3. From this we can see that is sufficient to prove

$$2 + (r'(z) - r(x) \leq 2(r'(x) - r(x)) \tag{14}$$

in order to see that our claimed potential function is valid.
From rearranging the formula above we get

$$r'(z) + r(x) - 2r'(x) \leq -2 \tag{15}$$

$$\underbrace{(r'(z) - r'(x))}_{\log \frac{s'(z)}{s'(x)}} + \underbrace{(r(x) - r'(x))}_{\log \frac{s(x)}{s'(x)}} \leq -2 \tag{16}$$

from the definitions of r and z. Now we look on the subtree of s'(z) as a root and the subtree with s(x) as root. Notice from Figure 3 we can see that the sets of these subtrees are disjoint. This means that we can upper bound the two quantities by

$$\leq \log\theta + \log(1 - \theta) \tag{17}$$

with $0 < \theta < 1$. Notice that the logarithm is a concave function this term is maximized by $\theta = 0.5$ such that we get

$$\underbrace{\log 0.5}_{-1} + \underbrace{\log 0.5}_{-1} = -2 \tag{18}$$

What is exactly what we have claimed and those proves our Lemma. ■

Since the amortized cost of a sequence of operations is $\geq \sum$ real costs + $\Delta\Phi$ what is the upper bound of $\Delta\Phi$ hence, the maximal $\Delta\Phi$?
This is the difference $max(\Phi) - min(\Phi)$ what is given by $\mathcal{O}(\sum \log \frac{W}{w_k})$ with $W = \sum_k w_k$ bounding $max(\Phi) \leq n \log W$ and $min(\Phi) \geq \sum \log w_k$.
This bound represents the case of splaying a leaf with k up to the root.

- **Corollary 1:** Splay trees have $\mathcal{O}(\log n)$ amortized time.
- **Proof:** Set all weights $w_k = 1$ then $s(t) = n$ and by the Acces Lemma follows the bound splay $\in \mathcal{O}(\log n)$. ■
- **Corollary 2:** Splay trees are statical optimal.
- **Proof:** We can express statical optimality by the entropy of information $\mathcal{O}(\sum_{i=1}^{n} p_k \log \frac{1}{p_k})$. Normalize our weights by the sum of all weights hence, $W = \sum_k w_k$ and applying the Access Lemma the result immediately follows. ■

4.3 Insert and Delete

First we define the to operations Split and Join for Insert and Delete.

$$T.split(x) \coloneqq \begin{cases} y \in T_1 & \text{if } y \leq x \\ y \in T_2 & \text{otherwise.} \end{cases} \tag{19}$$

such that $T_1 = \{y \in T : y \leq x\}$ and $T_2 = \{y \in T : y > x\}$.

$$T_1.join(T_2) \coloneqq T_1 \cup T_2 \tag{20}$$

such that $\forall x \in T_1$ and $\forall y \in T_2$ holds $x \leq y$.

Algorithm 3 T.split(x)

1: $T.splay(x)$
2: $T_2 \leftarrow T.root.right$
3: $T_1 \leftarrow T.root$
4: **return** T_1, T_2

Detaching decreases potential Φ. This is because it only changes the potential of the root which gets smaller.

Algorithm 4 $T_1.join(T_2)$

1: $T_1.splay(max(T_1))$
2: $T_2 \leftarrow T_1.right$
3: $T \leftarrow T_1.root$
4: **return** T

Attaching increases potential Φ but lesser than n. This is since the potential of the root increases by T_2 such that $rank(T.root) < \log n$ with uniform weights.

Algorithm 5 T.insert(x)

1: $T_1, T_2 \leftarrow T.split(x)$
2: $T \leftarrow (T_1.join(x)).join(T_2)$
3: **return** T

Algorithm 6 T.delete(x)

1: $T_1, T_2 \leftarrow T.split(x)$
2: $T_1.root \leftarrow T_1.left$
3: $T \leftarrow (T_1.join(T_2))$
4: **return** T

Since split and join $\in \mathcal{O}(\log n)$ also insert and delete $\in \mathcal{O}(\log n)$ amortized.

5 Properties of Splay Trees

5.1 Entropy Bound

A BST algorithm is Statically Optimal if, given an input sequence where element k appears a p_k fraction of the time, the amortized access time per search is

$$\mathcal{O}(-\sum_{i=1}^{n} p_k \log p_k) \tag{21}$$

As seen splay trees hold this property even without knowing p_k beforehand such that we can solve the problem generically.

5.2 Working Set

- A BST algorithm has the Working Set Property if for a given search for x_k , if t_k distinct elements were accessed since the last access of x_k , the search takes an amortized $\mathcal{O}(\log t_i)$.
- The Working Set Property implies Static Optimality. If a few items are accessed often in some subsequence, the Working Set Property guarantees that these accesses are fast.
- The Working Set Property says that keys accessed recently are easy to access again which is achieved by splay trees as shown by Sleator and Tarjan.

5.3 Dynamic Finger Property

- A BST algorithm has the Dynamic Finger Property if, for any sequence of operation $x_1, x_2, .., x_m$, the amortized access time for $x_k \in \mathcal{O}(|(x_k - x_{k-1})|)$.
- The Dynamic Finger Property tells me that as long as my queries remain close in space, the time needed will be small.
- The Dynamic Finger Property is achieved by splay trees as shown by Richard Cole et. al. and Richard Cole.

(a) Geometric view

(b) Binary search tree

Figure 5: Traversing a binary search tree with sequence 3, 1, 4, 2

5.4 Unified Property

- The Working Set Property says that keys accessed recently are easy to access again. The Dynamic Finger Property says that keys close in space to a key recently accessed are also easy to access. The following property will combine these.
- A BST algorithm has the Unified Property if, given that $t_{i,j}$ unique keys were accessed between x_i and x_j , then search x_j costs an amortized

$$\mathcal{O}(\log \min_{i \leq j}(|x_i - x_j| + t_{i,j} + 2)) \tag{22}$$

- This is a generalization of both the Working Set Property and the Dynamic Finger Property proposed by Iacono implying both. If there is a key that was accessed somewhat recently and is somewhat close in space, this access will be cheap.
- It is unknown whether or not there is any BST that achieves the Unified Property. The best upper bound known is a BST that achieves an additive $\mathcal{O}(\log \log n)$ factor on top of every operation as shown by Bose et. al.

5.5 Dynamic Optimality

- Let $OPT(\sigma)$ be the optimal cost of servicing access sequence σ in the BST model. Then the BST is dynamic optimal $\iff \forall \sigma BST(\sigma) = \mathcal{O}(OPT)$. That is it achieves a time bound similar to the best offline BST having access to sequence σ beforehand. Hence, it is constant O(1) competitive.
- Splay trees are conjectured to achieve dynamic optimality by Sleator and Tarjan. This would imply that splay trees rule over all other BST's.
- Tango trees achieve an $\mathcal{O}(\log \log n)$ competitive ratio relative to the offline optimal binary search tree as shown by Demaine et. al.

6 Sources

The main sources for the notes are the lectures given by Erik Demaine on Dynamic Optimality for definitions and geometric view, David Karger on Splay Trees 1 and David Karger on Splay Trees 2 for the proof of the Access Lemma and insertion/deletion for splay trees, Jelani Nelson on Splay Trees for Zig-Zig/Zig-Zag rotations and Srinivas Devadas on Dynamic Programming for the section on static optimal binary search trees. For general background on binary search trees the textbook Introduction to Algorithms by Cormen, Leiserson, Rivest and Stein has been used. The original paper for Treaps was written by Aragon and Seidel. Further papers for data structures, definitions and proofs have been linked within their respective section which are:

Aragon, Cecilia R.; Seidel, Raimund: Randomized Search Trees. Proc. 30th Symp. Foundations of Computer Science, Washington, D.C.: IEEE Computer Society Press, pp. 540–545 (1989)

Prosenjit Bose, Karim Douïeb,Vida Dujmović, Rolf Fagerberg: An O(log log n)-Competitive Binary Search Tree with Optimal Worst-Case Access Times. Scandinavian Workshop on Algorithm Theory SWAT 2010: Algorithm Theory - SWAT 2010 pp 38-49

Richard Cole, Bud Mishra, Jeanette P. Schmidt, Alan Siegel: On the Dynamic Finger Conjec ture for Splay Trees. Part I: Splay Sorting log n-Block Sequences. SIAM J. Comput. 30(1): 1-43 (2000)

Richard Cole: On the Dynamic Finger Conjecture for Splay Trees. Part II: The Proof. SIAM J. Comput. 30(1): 44-85 (2000)

Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Patrascu. Dynamic optimality — almost. In FOCS '04: Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science (FOCS'04), pages 484–490. IEEE Computer Society, 2004.

John Iacono: Alternatives to splay trees with O(log n) worst-case access times. SODA 2001: 516-522

Daniel Dominic Sleator, Robert Endre Tarjan: Self-Adjusting Binary Search Trees J. ACM 32(3): 652-686 (1985)