ICS 621: Analysis of Algorithms Fall 2019

Lecture 6

Prof. Nodari Sitchinava *Scribe: Evan Hataishi, Muzamil Yahia (from Ben Karsin)*

1 Overview (BST)

In the last lecture we reviewed on Binary Search Trees (BST), and used Dynamic Programming (DP) to design an optimal BST for a given set of keys $X = \{x_1, x_2, \ldots, x_n\}$ with frequencies $F = \{f_1, f_2, \ldots, f_n\}$ i.e. $f(x_i) = f_i$. By optimal we mean the minimal cost of $m = \sum_{i=1}^{n} f_i$ queries given by $C_m = \sum_{i=1}^{n} f_i \cdot c_i$ where c_i is the cost for searching x_i. Knuth designed $O(n^2)$ algorithm to construct an optimal BST on X given F [3]. In this lecture, we look at self-balancing data structures that optimize themselves without knowing in advance what those frequencies are and also allow update operations on the BST, such as insertions and deletions.

Note that if we define $p(x) = f(x)/m$, then $C_m = \sum_{i=1}^{n} f_i \cdot c_i$ is bounded from above by $m(H_n + 2)$ (a proof is given by Knuth [4, Section 6.2.2]), and bounded from below by $m\frac{H_n}{\log 3}$ (proved by Mehlhorn [5]) where H_n is the entropy of n probabilities given by $H_n = \sum_{i=1}^{n} p(x_i) \log \frac{1}{p(x_i)}$ and therefore $C_m \in \Theta(mH_n)$. Since entroby is bounded by $H_n \le \log n$ (see Lemma 2.10.1 from [1]), we have $C_m = O(m \log n)$.

2 Splay Trees

Splay trees, detailed in [6] are self-adjusting BSTs that:

- implement dictionary APIs,
- are simply structured with limited overhead,
- are c-competitive [1] 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

2.1 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.

The 3 rotate rules are zig, zig-zag, and zig-zig, and are detailed below:

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

Algorithm 1

```
1: function SEARCH($k$)
2:     $x$ = BST-FIND($k$)
3:     while $x \neq root$ do
4:         SPLAY($x$)
5:     end while
6: end function
```

2.1.1 Zig

The ZIG operation is applied if the element being moved, x, is the child of the root, as seen in Figure 1. When this is the case, the element, x, is simply rotated with the root node, making it the new root.

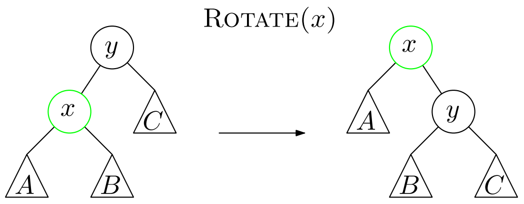


Figure 1: A case where the ZIG operation is used to move target node x to the root. This is only used when the target node is the direct child of the root node.

2.1.2 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 2. 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).

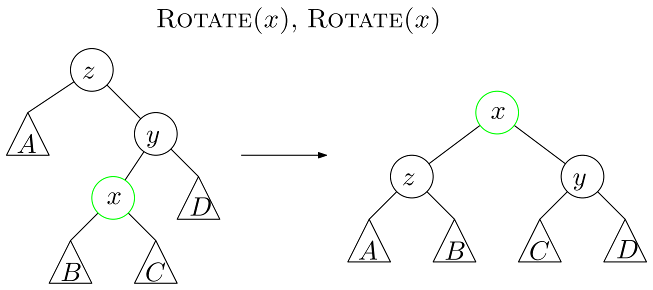
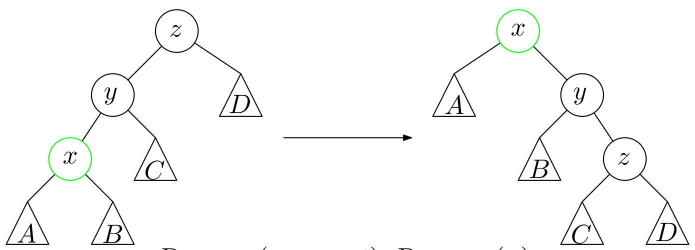


Figure 2: A case where the ZIG-ZAG operation is employed to move target node x up the tree. The ZIG-ZAG operation would also be used in the symmetric case (i.e., where x is the right child of y and y is the left child of z).

2.1.3 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 3 illustrates the result of the ZIG-ZIG operation on node x.



ROTATE($x.parent$), ROTATE(x)

Figure 3: 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.

2.2 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 4 illustrates how a Splay tree would perform when repeatedly searching such a target.

We see in Figure 4 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 $O(\log n)$.

2.3 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.

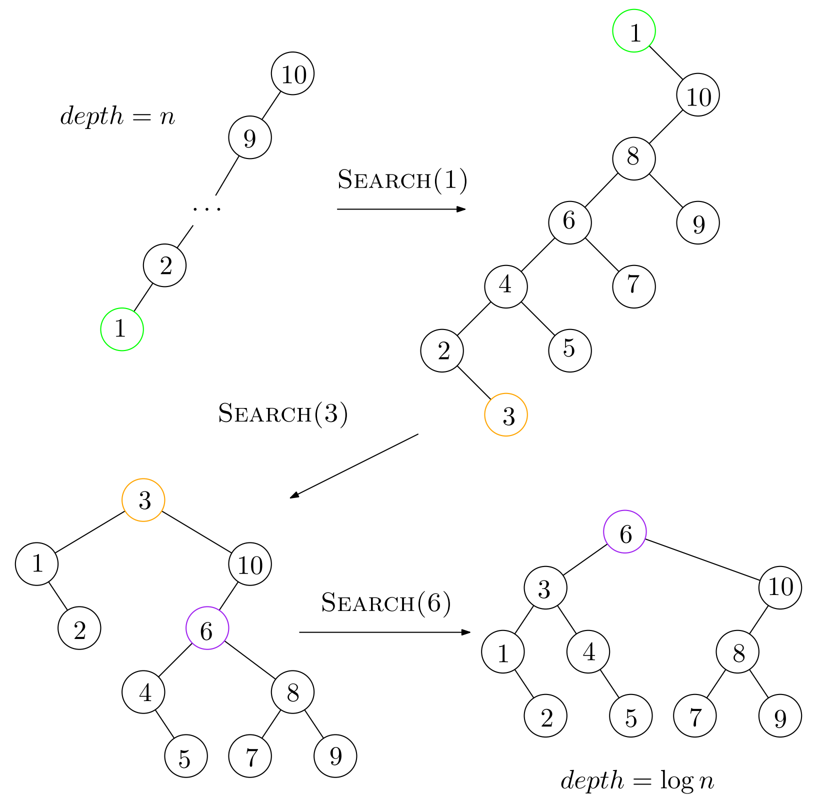


Figure 4: An example of a worst-case tree and result of repeatedly searching (and using the SPLAY operation) it. We see that, after searching just 3 times, we have a tree of optimal height for the input size.

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 2.1. Recall that the amortized cost using the potential method is defined as:

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

2.3.1 Zig-zig analysis

Consider the ZIG-ZIG operation defined in Section 2.1 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 3 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 3:

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

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[2] and convex[3] 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:

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

2.3.2 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.

2.3.3 Zig analysis

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

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

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

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 1 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 $\hat{c_i}(x) \leq 1 + 3(r_i(x) - r_{i-1}(x))$

2.3.4 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:

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

□

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

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

□

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 $c = c'/\bar{c}$.

2.4 Other Splay tree operations

Insertion (Algorithm 2) 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.

Algorithm 2

```
function INSERT($k$)
    $v$ = BST-INSERT($k$)
    while $v \neq root$ do
        SPLAY($v$)
    end while
end function
```

Algorithm 3

```
function DELETE($k$)
    $v$ = BST-DELETE($k$)
end function
```

3 Pairing heaps

In previous lectures, we discussed the Fibonacci heaps and their applications to improving the amortized run time of classical graph problems. While efficient in theory, Fibonacci heaps require complex implementations and are not as fast as other heaps in practice, for every node in the heap requires 4 different pointers [2]. To overcome these limitations, [2] introduces the *pairing heap*, which is both easy to implement and fast in practice.

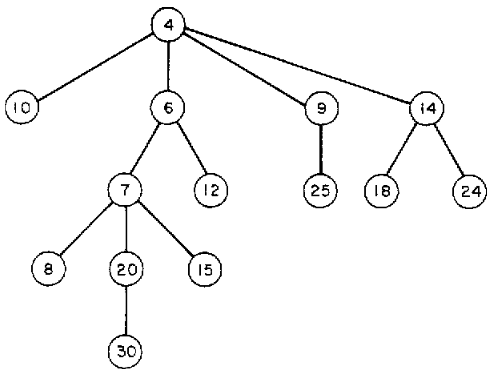


Figure 5: Example of a heap-ordered tree taken from [2].

3.1 Properties

As with Fibonacci heaps, pairing heaps are heap-ordered trees and do not maintain the invariants required by binomial trees. Each node in a pairing heap has a left pointer to its first child and a right pointer to its next older sibling. This property allows the pairing heap to adjust itself into a "half-ordered binary tree" after some operations that we will further explore below. For a tree to be half-ordered, the root has an empty right subtree, and the key of node any node x must be less than or equal to the key of every node in x's left subtree i.e. heap order is only guaranteed going down left substrees (new invariant). Lastly, a third pointer to the node's parent is also needed to perform an efficient DECREASE-KEY operation. With this representation, right children are treated as siblings and share the same parent.

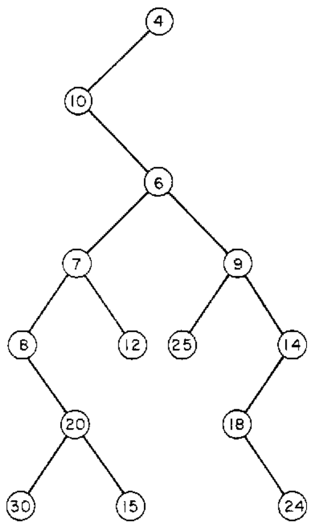


Figure 6: Half-ordered binary tree version of Fig. 5. [2]

3.2 Functions

Pairing heaps similarly implement all the heap operations we have seen before, such as UNION, INSERT, and LINK. However, to accomodate the differing structure and make the heap "self-adjusting", we must modify DECREASE-KEY and EXTRACT-MIN.

3.2.1 DECREASE-KEY

Pairing heaps implement DECREASE-KEY the same way Fibonacci heaps: decrease the key, cut the node out (if it is not already root) together with its subtree, and union the two trees (the removed subtree and the original tree without the subtree). However, since pairing heaps have no notion of "marking", the operation only does one cut rather than the RECURSIVE-CUT.

3.2.2 EXTRACT-MIN

EXTRACT-MIN turns out to be the most costly and complex operation for pairing heaps. First, we simply remove the root. The result is a collection of heap-ordered trees, each rooted at a child of the deleted node. Now the question becomes: how can the trees be re-linked efficiently? In the worst case, the extraction will result in $n - 1$ trees for a heap that contained n nodes before extraction. Arbitrarily linking the trees in succession will have a $\Theta(n)$ worst-case time bound. However, we can do better by linking the trees in two passes. On the first pass, the trees are linked in pairs (hence the name "pairing" heap) from left to right. The second pass links all the trees into one tree starting from the rightmost tree and linking to the left (Figure 7). The full analysis of the amortized cost of EXTRACT-MIN (shown in Algorithm 4) will be similar to the structure of Splay tree with slight difference in the order of operations.

Figure 7 shows how pairing heap extracts min and replaces it with a new min. By comparing the initial tree, and the tree after each union operation from right-to-left direction (after pairing), it is

Algorithm 4

```
function EXTRACT-MIN($Q$)
    $m \leftarrow$ MINIMUM($Q$)
    $k \leftarrow \lceil (\#\text{ of } m\text{'s children})/2 \rceil$    ▷ Recall that for any node $x$, nodes in $x$'s right spine are siblings of $x$
    for each pair of heaps $Q_i, Q_{i+1}$ from left to right, $Q'_{\lfloor i/2 \rfloor + 1} \leftarrow$ UNION($Q_i, Q_{i+1}$)
    $tmp = Q'_k$
    for $j = k - 1$ down to 1 do
        $tmp =$ LINK($tmp, Q'_j$)
    end for
    $Q = tmp$
    return $m$
end function
```

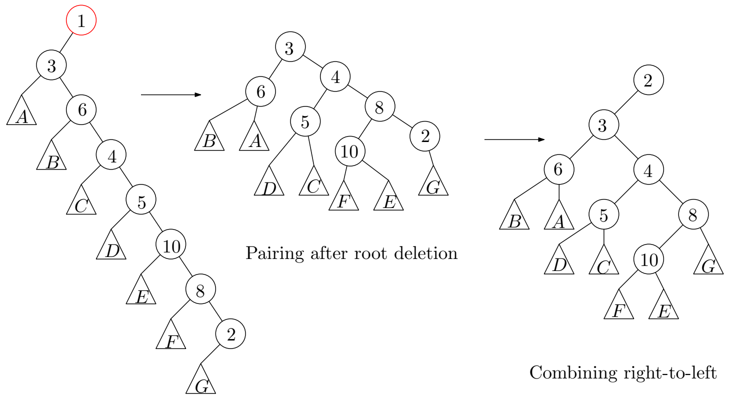


Figure 7: EXTRACT-MIN followed by pairing.

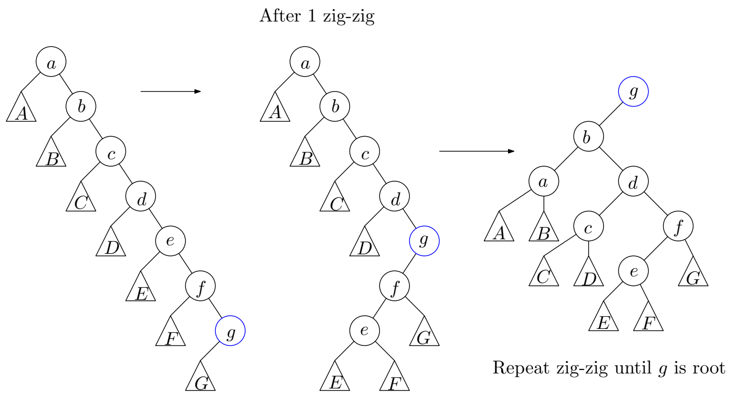


Figure 8: Splaying g obtains the same tree structure.

noticeable that the first union is equivalent to zig operation, while rest of unions are equivalent to zig-zig operations applied to the right spine. Compare that with Figure 8 that shows a BST with similar tree structure as Figure 7, but splays g to the root. The final tree is as balanced as the pairing heap tree, which gives an intuition why a potential function Φ for Splay tree would work for pairing heap. We leave it as an exercise that the potential $\Phi(T_i) = \sum_{x \in T_i} r(x)$ gives $O(\log n)$

amortized cost for EXTRACT-MIN for pairing heaps.

References

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

[2] M. Fredman, R. Sedgewick, D. Sleator, and R. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1:111–129, 11 1986.

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

[4] D. E. Knuth. *The art of computer programming, , Volume III, 2nd Edition*. Addison-Wesley, 1998.

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

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