21120491 Advanced Data Structure and Algorithm Analysis **Spring-Summer 2022**

Lecture 4: Leftist Heap and Skew Heap

Lecturer: Deshi Ye *Scribe: D. Ye*

4.1 Overview

Heap is a binary tree that is completely filled, with the possible exception of the bottom level, which is filled from left to right [2]. For any node x, Let P be the parent node of x. In a min-heap, the key of P is at most of the key of node x. The root in a min-heap is the smallest key.

The min-heap maintains the Order Property that keys along each path are monotonically non-decreasing.

Heap is an efficient implementation of the priority queue ADT. Heap is not efficient for supporting the operation Find (or searching), but heap efficiently support the following operations:

- Insert: insert a new node into the heap
- FindMin: return the smallest key
- DeleteMin: return the smallest key and delete this node
- DecreaseKey: lowers the value of the key at a given position
- IncreaseKey: increase the value of the key at a given position
- Merge: combine two heaps into one

Mainly, heaps support the Insert and DeleteMin (or ExtractMin) operations.

When to use a heap: If our algorithm or application dynamically requires to get minimum (or maximum) values, the heap is a good choice. For example, heaps are used as a part of algorithms for the shortest path problem or the minimum spanning tree problem.

In this lecture, we are going to study two important heaps, the leftist heap and the skew heap.

Leftist heap is to going to keep the structure that the left subtree is more "heavy" than that of right subtree, this will benefit for the Merge operation. Null path length as defined in 4.1 is going to quantify the heaviness of a subtree.

Definition 4.1 *The null path length, $NPL(X)$, of any node X is the length of the shortest path from X to a node without two children. Define $NPL(NULL) = -1$.*

The null path length is the shortest distance to get to a null pointer.

Note that $NPL(X) = \min \{ NPL(C) + 1 \text{ for all } C \text{ as children of } X \}$.

Definition 4.2 *The leftist heap property is that for every node X in the heap, the null path length of the left child is at least as large as that of the right child.*

Remark: Any non-leftist tree can be made leftist by swapping left & right children at the node where the leftist condition is violated.

As the definition of the leftist heap 4.2, the leftist heap will guarantee a short right path (at most of $\log n$).

Theorem 4.3 *A leftist tree with r nodes on the right path must have at least $2^r - 1$ nodes. In other words, a leftist heap with n nodes, then the length of the right path is at most of $\log n$.*

Proof: We prove it by induction. The base case that $r = 1$, i.e., the right path has at least one node, thus the heap has at least one node. Suppose that the theorem holds for any $k \leq r$.

Now, let us consider that the right path has $r + 1$ nodes, in which A is the first node in the path. Let B and C be A's left child and right child respectively. Note that the subtree root at C has r nodes in its right path. Thus, the subtree C has nodes at least $2^r - 1$ by the induction assumption.

Since C's right path has r nodes and it is a leftist heap, the null path length of C, $NPL(C) \geq r$. Note that the tree A is a leftist heap, we have $NPL(B) \geq NPL(C) \geq r$, which means the right path of B is also at least of r, and then the subtree B has nodes at least $2^r - 1$ by the induction assumption.

In sum, the nodes of subtree A are at least of $2(2^r - 1) + 1 = 2^{r+1} - 1$. This concludes the proof of the theorem. ■

Insert operation can be regarded as a special case of Merge. In the following, we consider the Merge operation.

Let H_1 and H_2 be the two heaps that we are going to merge. Denote $Merge(H_1, H_2)$ to be the function that returns a new heap with the keys from H_1 and H_2.

The recursive version of Merge operation is given as below. Suppose that the root of H_1 is smaller than the root of H_2, then

Step 1: $Merge(H_1 \rightarrow Right, H_2)$, return H_2 as the resulted heap

Step 2: Attach (H_2, $H_1 \rightarrow Right$), which is to attach the result of step 1 (H_2) as the right child of H_1. Return H_1.

Step 3: Swap($H_1 \rightarrow Right, H_1 \rightarrow Left$) if necessary such that it is a leftist heap.

Remark: Since the Merge operation takes only on the right path, by Theorem 4.3, the running time of Merge operation is $O(\log N)$, where N is the total number of nodes in H_1 and H_2.

4.2 Skew heap

Skew heap was invented by Sleator and Tarjan [1].

Problems with leftist heap:

- Spaces to store Npl
- Extra complexity to maintain and check Npl

- right side is often "heavy" and requires a switch.

A skew heap is a self-adjusting node-based data structure, which is the resulting heap by merging the right paths of the two heaps and always swapping the left and right children except that the largest of all the nodes on the right paths does not have its children swapped.

Note that there is no Npl involved in the skew heap. Skew heaps have the advantage that no extra space is required to maintain path lengths and no tests are required to determine when to swap children.

4.2.1 Amortized analysis for Skew heap

Definition 4.4 *A node p is* **heavy** *if the number of descendants of p's right subtree is at least half of the number of descendants of p, and* **light** *otherwise. Note that the number of descendants of a node includes the node itself.*

Theorem 4.5 *We have two heaps H_1 and H_2 of n_1 nodes and n_2 nodes, respectively. The amortized runnig time of Merge H_1 and H_2 is $O(\log n)$, where $n = n_1 + n_2$.*

Proof: Let the potential function $\Phi(H_i)$ be the number of heavy nodes in the tree H_i. Let H_3 be the result of the merged heap.

Suppose that the right path of H_i has l_i light nodes and h_i heavy nodes, for $i = \{1, 2\}$, respectively.

Suppose that merging these two heaps performs m iterations. Let c_i and $\hat{c}_i$ be the actual cost and the amortized cost in ith iteration. The total amortized cost is given as below.

$$\sum_{i=1}^{m} \hat{c}_i = \sum_{i=1}^{m} c_i + \Phi(H_3) - (\Phi(H_1) + \Phi(H_2))$$

Note that the acutal time to merge is at most the sum of the right paths of H_1 and H_2, i.e., $\sum_{i=1}^{m} c_i \leq l_1 + l_2 + h_1 + h_2$.

Now let us consider the net change of potential $\Delta\Phi = \Phi(H_3) - (\Phi(H_1) + \Phi(H_2))$. It is worthy to note that the only nodes whose heavy/light status can change are nodes that are initially on the right path. Moreover, all heavy nodes in the right path of H_1 and H_2 will become the light nodes in H_3 by Lemma 4.6, which implies the potential decrease by $-(h_1 + h_2)$. Thus the heavy nodes increased at most of $l_1 + l_2$ from the light nodes in H_1 and H_2. In sum, $\Delta\Phi \leq l_1 + l_2 - h_1 - h_2$.

From the above analysis, we have

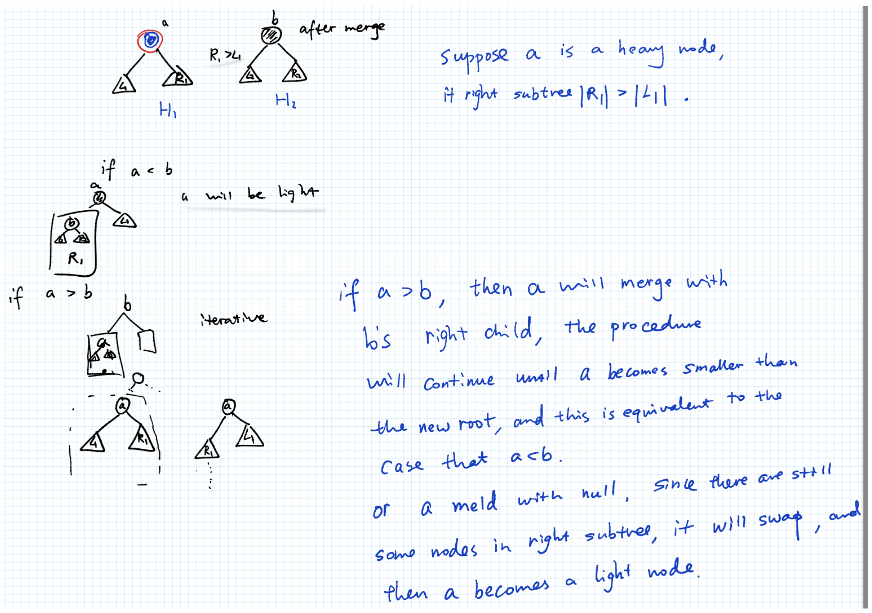
$$\begin{aligned}
\sum_{i=1}^{m} \hat{c}_i &= \sum_{i=1}^{m} c_i + \Phi(H_3) - (\Phi(H_1) + \Phi(H_2)) \\
&\leq l_1 + l_2 + h_1 + h_2 + l_1 + l_2 - h_1 - h_2 \\
&= 2(l_1 + l_2)
\end{aligned}$$

By Lemma 4.7, we know $l_1 + l_2 = O(\log n)$. Hence, the sum of amortized running time for Merge operation is $O(\log n)$, where n is the total nodes in two trees. ■

Remark: Note that Insert is a special case, FindMin takes $O(1)$, DeleteMin will follow from Merge. Suppose we begin with no heaps and carry out an arbitrary sequence of skew heap operations (Merge, Insert, FindMin, DeleteMin). The initial potential is zero and the final potential is nonnegative, so the total running time is bounded by the sum of the amortized times of the operations.

Lemma 4.6 *All heavy nodes in the right path of H_1 and H_2 will become the light nodes in H_3.*

Proof: The proof is illustrated in Fig. 4.1. ■



Suppose a is a heavy node, it right subtree $|R_1| > |L_1|$.

if $a > b$, then a will merge with b's right child, the procedure will continue until a becomes smaller than the new root, and this is equivalent to the case that $a < b$.

or a meld with null, since there are still some nodes in right subtree, it will swap, and then a becomes a light node.

Figure 4.1: Illustration of proof

Lemma 4.7 *In a tree with k light nodes in its right path, the tree has nodes at least $2^k - 1$. That means there are at most $O(\log n)$ light nodes in the right path of a n-node tree.*

Proof:

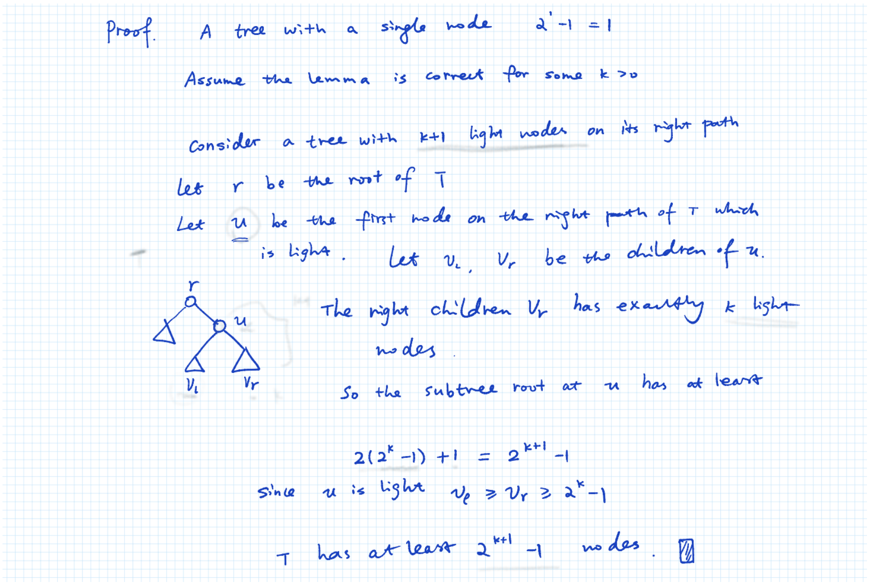
The proof is illustrated in Fig. 4.2.

■

References

[1] D. D. Sleator and R. E. Tarjan. Self-adjusting binary trees. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, page 235–245, New York, NY, USA, 1983. Association for Computing Machinery.

[2] M. A. Weiss. *Data Structures and Algorithm Analysis in C (2nd Ed.)*. Addison-Wesley Longman Publishing Co., Inc., USA, 1996.



Proof. A tree with a single node $2^1-1=1$

Assume the lemma is correct for some $k>0$

Consider a tree with $k+1$ light nodes on its right path

Let r be the root of T

Let u be the first node on the right path of T which is light. Let v_l, v_r be the children of u.

The right children V_r has exactly k light nodes.

So the subtree root at u has at least

$$2(2^k-1)+1 = 2^{k+1}-1$$

since u is light $v_l \geqslant v_r \geqslant 2^k-1$

T has at least $2^{k+1}-1$ nodes. ▨

Figure 4.2: Illustration of proof