Amortized Analysis

Ariel Goodwin, Malcolm Sutcliffe

March 12th, 2020

This is the augmented transcript of a lecture given by Luc Devroye on the 12th of March 2020 for the Honours Data Structures and Algorithms class (COMP 252, McGill University). The subject was amortized analysis.

Principle

The principle of amortized analysis is to consider a collection of operations on a data structure over a given time period. Looking at the cost of an isolated operation on a data structure may be misleading when its average cost in a sequence of operations may be small. This method of analysis was first developed by Sleator and Tarjan in 1985.

Definitions and Notation

- t - time
- $\mathcal{D}_t$ - the data structure at time t
- $\Phi(\cdot)$ - the potential of the data structure, which must always be greater than or equal to 0. We associate 0 with the notion of "good", "stable", "empty", or "desired".

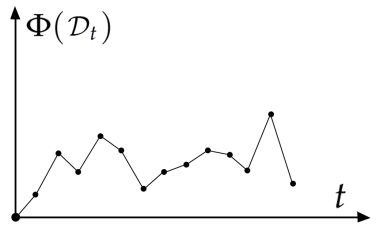

Figure 1: Potential of a data structure as a function of time.

Let us define the *amortized time* of an operation that takes the structure $\mathcal{D}_t$ to $\mathcal{D}_{t+1}$ as follows:

$$\text{Amortized Time} := \text{Actual Time} + \underbrace{\Phi(\mathcal{D}_{t+1}) - \Phi(\mathcal{D}_t)}_{\Delta\Phi}, \tag{1}$$

where the last two terms represent the "change in potential" of the data structure.

Naturally, we can extend this to the amortized time of operations that take $\mathcal{D}_0$ to $\mathcal{D}_t$. The intermediate values of the potential function vanish because they form a telescoping sum.

$$\begin{aligned} \text{Amortized Time} &= \text{Actual Time} + \Phi(\mathcal{D}_t) - \Phi(\mathcal{D}_0) \\ &= \text{Actual Time} + \Phi(\mathcal{D}_t) \\ &\geq \text{Actual Time}, \end{aligned} \tag{2}$$

where the second equality holds assuming that $\Phi(\mathcal{D}_0) = 0$. The requirement that Φ is nonnegative ensures that we are determining an upper bound for the actual time, as evidenced in the last line of equation (2).

Example 1. Stack with Multipop

OVERVIEW: We consider a stack $\mathcal{S}$ with the operations of PUSH(x, $\mathcal{S}$) and MULTIPOP(k, $\mathcal{S}$). PUSH is the standard operation, and MULTIPOP is a condensed operation that performs k consecutive pops. In this example, we define our potential function Φ to be the size of the stack at any given time.

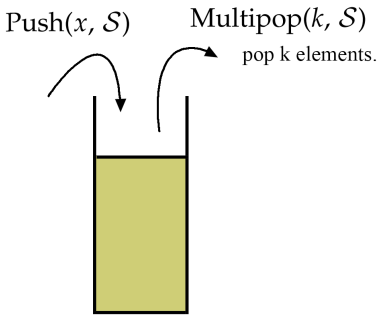

Figure 2: Visualizing the stack.

TABLE 1: OPERATION COSTS FOR STACK WITH MULTIPOP

	PUSH(x, $\mathcal{S}$)	MULTIPOP(k, $\mathcal{S}$)
Actual Time	1	$\min(k, \lvert\mathcal{S}\rvert)$
$\Delta\Phi$	1	$-\min(k, \lvert\mathcal{S}\rvert)$
Amortized Time	2	0

We take the minimum in the third column because either k elements are popped or the stack becomes empty. Note that the values in the fourth row are determined using equation (1) above. Over the course of t operations starting from an empty stack (i.e., $|\mathcal{S}| = 0$) we see that the amortized time is less than or equal to $2t$.

Example 2. Lazy-Delete

OVERVIEW: LAZY-DELETE is an alternative to the classical DELETE. The essence of LAZY-DELETE is that it postpones the heavy-lifting associated with deleting a single element. An algorithm outline would be the most illustrative refresher of how it works.

This example may remind you of our earlier lecture about red-black trees, when the LAZY-DELETE operation was first introduced. LAZY-DELETE is applicable to various different data structures. Here we define it in general and perform a specific analysis of its performance on a red-black tree.

LAZY-DELETE(x):

1. Mark x as deleted (x is now a ghost element).
2. If the number of ghost elements is greater than or equal to 50% of the structure's size, then reconstruct the data structure from scratch (without the ghost elements).

We define Φ accordingly:

- $\Phi(\mathcal{D}_t) = 2 \times$ (number of ghost elements in the data structure at time t)

which yields $\Phi(\mathcal{D}_0) = 0$ obviously. The coefficient of 2 may seem arbitrary at this point. To see why we chose 2, consider what would change in the fourth column of Table 2 below if we let the coefficient be 1 instead. Remember that we have freedom in defining Φ as long as it satisfies nonnegativity.

RED-BLACK TREE: Let n be the number of nodes in a red-black tree. Recall from the lecture on red-black trees that the reconstruction of the tree without ghost nodes takes linear time.

RECONSTRUCTION: This can be done by first performing an inorder traversal to retrieve all of the unmarked nodes (takes $O(n)$ time). Then, using this sorted list, select elements in a binary search-style pattern to build a complete binary tree, with all black nodes except for the bottom layer which should be red. This is also clearly $O(n)$, so the whole operation is $O(n)$. We will assume that reconstruction costs precisely n.

TABLE 2: OPERATION COSTS FOR LAZY-DELETE

	INSERT	LAZY-DELETE	RECONSTRUCT
Actual Time	$\log_2 n$	1	n
$\Delta\Phi$	0	2	$-n$
Amortized Time	$\log_2 n$	3	0

A more explicit way to see that $\Delta\Phi = -n$ for RECONSTRUCT is to view $-n$ as $-(2 \times \frac{n}{2})$ and recall that $\Phi = 2 \times$ (number of ghost elements).

Starting with an empty tree (i.e., $n = 0$), we see that the cost of t operations (INSERT, LAZY-DELETE) is less than or equal to:

$$t \log_2 t + 3t$$

where the factors of t arise because we are performing at most t INSERTS or DELETES, and the size of the tree cannot exceed t.

EXERCISE: A binary tree on n nodes is *perfectly balanced* if all leaves are at distance h or $h-1$ from the root where $h = \lfloor \log_2 n \rfloor$.

1. Given any binary search tree on n nodes, write an $O(n)$ algorithm for morphing it into a perfectly balanced binary search tree.
2. Show that for every binary tree on n nodes, its height h is greater than or equal to $\lfloor \log_2 n \rfloor$.
3. Given a perfectly balanced binary search tree, show how to add colors to the nodes without altering the shape so that it becomes a red-black tree.

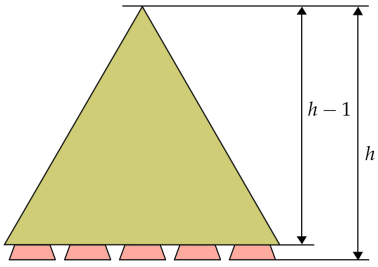

Figure 3: A perfectly balanced binary tree.

Example 3. Amortized Weight-Balanced Trees

G. Varghese first introduced the amortized approach for maintaining weight-balanced trees (Cormen, Leiserson, Rivest, and Stein, 2009, p. 473).

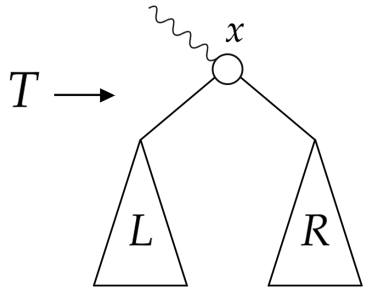

Figure 4: Subtree of an arbitrary node in a weight-balanced tree.

OVERVIEW: A *weight-balanced tree* is a binary search tree such that for all nodes x in the tree, with subtrees as shown in Figure 4, we have that:

- $|L| \leq \alpha|T|$ and $|R| \leq \alpha|T|$

where $\alpha \in [\frac{1}{2}, 1)$ is a fixed design constant, T is the subtree rooted at x, and L & R are the left and right subtrees of x respectively. Such trees are called *α-balanced*.

PROPERTIES:

- SEARCH: $O(\log_2 n)$ worst-case time
- INSERT, DELETE: $O(\log_2 n)$ amortized time per operation

Now we define the potential function for this example:

$$\Phi(T) = C \sum_{x:||L|-|R||\geq 2} ||L| - |R|| \tag{3}$$

where C is a constant that we shall select soon, and T is the tree.

CLAIM:

$$\text{Height} \leq \frac{\log_2 n}{\log_2 \frac{1}{\alpha}} \,. \tag{4}$$

Proof. A node at a distance k from the root has a subtree of size less than or equal to $\alpha^k n$. Thus $\alpha^k n \geq 1$ which implies $k \leq \frac{\log_2 n}{\log_2 \frac{1}{\alpha}}$ after a little bit of algebraic manipulation. □

CLAIM: $\Phi = 0$ for a $\frac{1}{2}$-balanced tree.

Proof. First notice that for any node x in the tree, $|T| = |L| + |R| + 1$ where T is the subtree rooted at x (see Figure 4). Now for each node x, we have

$$|L|, |R| \leq \frac{|T|}{2} = \frac{|L|+|R|}{2} + \frac{1}{2} \tag{5}$$

which implies that $|L| \leq |R| + 1$ and $|R| \leq |L| + 1$. Thus there are no x in our tree satisfying the conditions of the summation in equation (3), meaning that $\Phi = 0$. □

INSERT, DELETE: The performance of these operations is similar to their performance on a standard binary search tree (actual time is $O(\log_2 n)$), plus, if necessary, the cost of rebalancing at the highest unbalanced node. Note also that $\Delta\Phi = O(\log_2 n)$ since at most $O(\log_2 n)$ nodes change their contribution to Φ by one.

We will now choose the constant C in equation (3) that will allow us to find our desired upper bound.

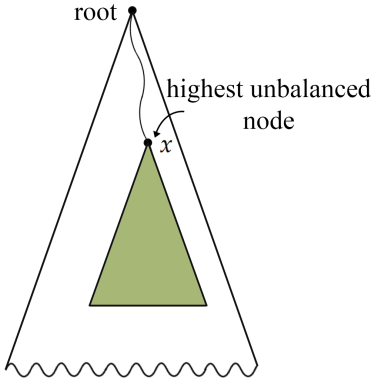

Figure 5: The node x and its subtree that needs rebalancing.

Let x be the highest unbalanced node, and T its subtree. Without loss of generality, $|L| > \alpha|T|$ and $|L| \geq |R| + 2$ because x is an unbalanced node. We will use the former inequality immediately, and the latter inequality in a moment.

Observe that $|R| = |T| - |L| - 1$ and apply the former inequality to yield:

$$|R| = |T| - |L| - 1 < (1-\alpha)|T| - 1 \implies |L| - |R| > (2\alpha - 1)|T| + 1$$

Now from the aforementioned latter inequality, we have that [1]$\Phi(\text{Tree}) \geq C||L| - |R||$. As well, $\Phi(\text{Tree after rebalancing}) = 0$. Thus we have:

$$\Delta\Phi \leq -C||L| - |R|| < -C(2\alpha - 1)|T| \,. \tag{6}$$

Since the cost of rebalancing is $|T|$, we see that we can make the amortized time of rebalancing less than or equal to 0 if $C = \frac{1}{2\alpha - 1}$, where we assume that $\alpha > \frac{1}{2}$.

The results of our analysis are summarized in the following table.

[1]Since the sum used to define Φ is guaranteed to be nontrivial.

Table 3: Operation Costs for a Weight-Balanced Tree

	Search	Insert	Delete	Rebalancing of Subtree T
Actual Time	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$	$\|T\|$
$\Delta\Phi$	0	$O(\log_2 n)$	$O(\log_2 n)$	$-\|T\|$
Amortized Time	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$	0

The remarkable result here is that rebalancing is free.

Example 4. Bitwise Addition

OVERVIEW: Given a number n, how much does it cost to count to n in the bit model? Starting at 0, and adding 1 each time, we obtain the sequence 0, 1, 10, 11, 100, 101, 110, $\ldots$, $(n)_2$

To see how we might choose Φ, suppose we have the number 10111111 in our sequence. Then:

$$\begin{array}{r} 10111111 \\ +\qquad 1 \\ \hline 11000000 \end{array} \tag{7}$$

A natural choice for Φ is the number of ones in the binary representation, because it is obviously nonnegative and we can see that it provides a reasonable measure of the amount of "disorder" at any given step.

If x is a number in our sequence, let k be the number of trailing 1's in its binary representation. Then we can see by example that:

$$x : 10\overbrace{1111}^{k} \quad \text{(a number in the sequence)} \tag{8}$$

Adding 1 in the standard fashion gives:

$$x+1 : 1\overbrace{10000}^{k+1} \tag{9}$$

Since this addition requires modifying at most $k+1$ bits, we have:

$$\begin{aligned} \text{Actual Time} &= k+1 \\ \Delta\Phi &= -k+1 \\ \implies \text{Amortized Time} &= 2 \end{aligned} \tag{10}$$

The amortized cost of one addition is 2. Thus, after performing n additions, the actual time is less than or equal to $2n$.

Example 5. Fibonacci Heaps

The Fibonacci heap was conceived by Fredman and Tarjan in 1984, and was first published in a journal in 1987.

OVERVIEW: Before delving into the analysis, let us contrast the time complexity of operations on a standard binary heap and a Fibonacci heap. Most of these operations should be familiar from our earlier lecture on priority queues, but there are some novel operations here that will be defined soon.

TABLE 4: COMPARING A BINARY HEAP AND A FIBONACCI HEAP

	Binary Heap (Actual Time)	Fibonacci Heap (Amortized Time)
INSERT	$\Theta(\log_2 n)$	$\Theta(1)$
DELETE	$\Theta(\log_2 n)$	$\Theta(\log_2 n)$
DELETE-MIN	$\Theta(\log_2 n)$	$\Theta(\log_2 n)$
DECREASE-KEY	$\Theta(\log_2 n)$	$\Theta(1)$
MELD	$\Theta(\log_2 n \times \log_2 m)$	$\Theta(1)$

Notice the improvements for the INSERT, DECREASE-KEY, and MELD operations. The n and m in the last row are the sizes of the two heaps being melded together.

THE DATA STRUCTURE: Relevant terminology and properties of the Fibonacci heap are listed below. Figure 6 illustrates these properties.

- The Fibonacci heap is a priority queue consisting of a *forest* (collection) of heap-ordered trees.[2]
- We interact with a given Fibonacci heap H with a pointer to the root of the tree with the minimum key.
- The *degree* of a node is the number of its immediate children. Each node stores its degree.
- Every node has a child pointer and a parent pointer.
- Each group of siblings is organized in a doubly-linked list.
- The *root-list* is the doubly-linked list of roots of all of the trees in the forest.
- Each node stores its *marked* or *unmarked* status. This is discussed below.

[2]Meaning that the key of each node is greater than or equal to the key of its parent.

REMARK: The notion of being *marked* will become important for ensuring that the max-degree is $O(\log_2 n)$. We will assume throughout, and prove later, that max-degree is less than or equal to $c \log_2 n$ for some c. A node x is marked if it has lost a child since the last time

x was made the child of another node. Also, root-list nodes are unmarked. For the time being, assume all nodes are unmarked. We will summarize the rules of marking after we have introduced the relevant operations.

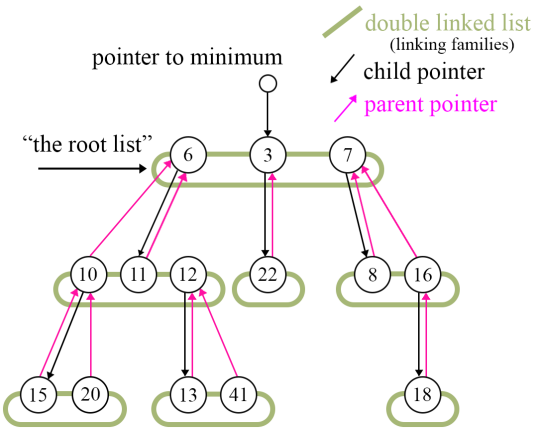

Figure 6: A sample Fibonacci heap.

SIMPLE OPERATIONS:

- [3]MELD($\mathcal{F}_1, \mathcal{F}_2$): Consists of combining the root-lists ($O(1)$) and finding the global minimum ($O(1)$).
- INSERT($x, \mathcal{F}$): Create a Fibonacci heap with 1 element and use MELD, which is clearly $O(1)$.

[3] $\mathcal{F}_1$ and $\mathcal{F}_2$ are two Fibonacci heaps. Note that finding the global minimum is $O(1)$ because we need only compare the previous minimum to the key in x.

POTENTIAL: In this example, we define our potential function as follows:

- $\Phi = \alpha$(Size of Root-List) + β(Number of Marked Nodes)

where α and β are scaling factors that we will choose later to give the desired result. The simple operations above take $O(1)$ amortized time.

NOTE: Performing INSERT n times creates the Fibonacci heap given in Figure 7. We may start to worry about how large the root-list can become. A sizeable root-list could cause problems when we extract the minimum element and need to find the second-smallest node. As we shall see below, we build the solution to this problem into the remaining operations.

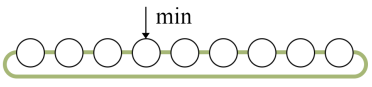

Figure 7: A Fibonacci heap that is entirely the root list.

OTHER OPERATIONS: Here we discuss the less trivial operations in Table 4 and their respective sub-operations.

The DECREASE-KEY operation changes the key of a node x to a lower value k, moving it to a higher priority in the queue. In the algorithm outline below, we assume that the input key is less than or equal to the key of x.

DECREASE-KEY$(x, k, \mathcal{F})$:

1. Set the key of x to k.
2. If $x \notin$ root-list and if the heap ordering between x and parent$[x]$ has been upset, then add x and its subtree to the root-list.
3. If x was moved to the root-list in step 2, check if x is the new minimum key and adjust the minimum pointer accordingly.

See Figures 8 and 9 to visualize this process. This operation has $O(1)$ actual time and $O(1)$ amortized time.

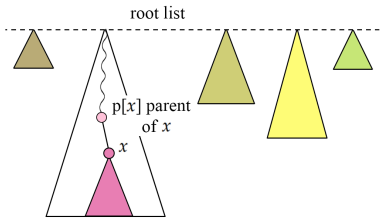

Figure 8: A Fibonacci heap before calling DECREASE-KEY on x.

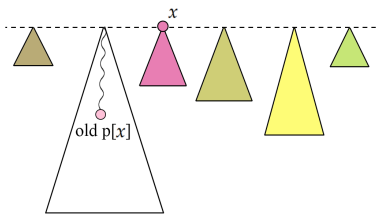

Figure 9: A Fibonacci heap after calling DECREASE-KEY on x.

REMOVING NODES: When carrying out the DELETE-MIN operation, it is straightforward to remove the minimum node and move its children to the root-list with their subtrees. The question that arises is how to deal with finding the new minimum. To ensure we have a reasonable handle on our root-list at all times, we employ a new operation CLEAN-UP-ROOT-LIST that will be detailed below.

DELETE-MIN$(\mathcal{F})$:

1. Report the minimum key node from the root-list, delete the node, and add its immediate children to the root-list as roots of their respective subtrees.
2. Perform CLEAN-UP-ROOT-LIST.

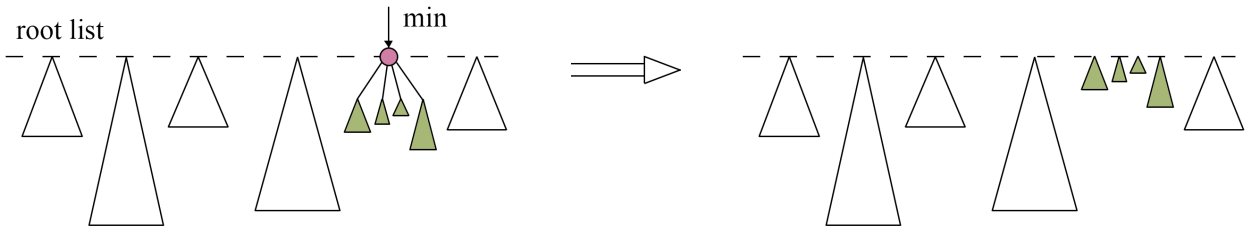

Figure 10: The first step of DELETE-MIN.

The first step has actual time that is $O(1)$ and $\Delta\Phi \leq \alpha c \log_2 n$. This is because the size of the root-list is growing by no more than

the max-degree, which is less than or equal to $c \log_2 n$ by our earlier remark.

DELETE($x, \mathcal{F}$):

1. Remove the node x from the heap and add its immediate children to the root-list as roots of their respective subtrees.
2. Perform CLEAN-UP-ROOT-LIST.

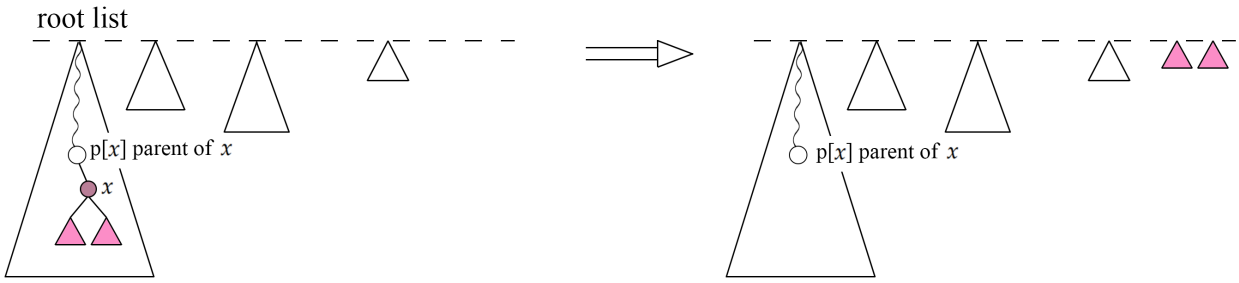

Figure 11: The first step of DELETE.

This can be done using two familiar operations in succession: DECREASE-KEY($x, -\infty, \mathcal{F}$) followed by DELETE-MIN($\mathcal{F}$). Thus the actual time is $O(1)$ and $\Delta\Phi \leq c \log_2 n$ for some c.

LET B be an array[4] of buckets, each initialized to nil. $B[i]$ is a pointer to a node in the root-list of degree i.

CLEAN-UP-ROOT-LIST($\mathcal{F}$):

```
1  Empty root-list into a stack S
2  while |S| > 0
3       x ← POP(S)
4       δ ← degree[x]
5       if B[δ] = nil
6            B[δ] ← x
7       else // root-list shrinks by 1
8            if key[x] < key[B[δ]] then exchange keys of x and B[δ]
9            make x a child of B[δ]
10           y ← B[δ]
11           degree[y] ← δ + 1
12           PUSH(y, S)
13           B[δ] ← nil
14 make a new root-list from the non-nil buckets in B
15 // size of this root-list is less than or equal
16 // to 1 + max-degree = O(log_2 n) (by earlier remark)
17 Find new min in O(log_2 n) time
```

[4] B should be smaller than the max-degree in the root-list + $\log_2$ (size of the root-list). We leave the management of the size of B as a small exercise.

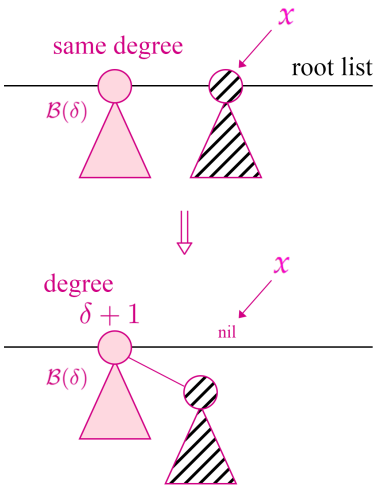

Figure 12: Illustrating the CLEAN-UP-ROOT-LIST operation.

Performing our analysis of this operation, we have:

$$\begin{aligned} \text{Actual Time} &\leq \text{Size of Root-List Before} \\ \Delta\Phi &\leq \alpha c \log_2 n - \alpha(\text{Size of Root-List Before}) \quad (11) \\ \implies \text{Amortized Time} &\leq \alpha c \log_2 n \ \text{ if we set } \alpha = 1. \end{aligned}$$

MARKED NODES: We now reintroduce the rules of marking.

1. Nodes in the root-list are unmarked.
2. A node that has lost a child (either through DECREASE-KEY or DELETE) is marked.
3. No node can lose 2 children.

CASCADING-CUT: To ensure that the third condition is never violated, we implement an operation called CASCADING-CUT. Whenever a node is about to lose a second child, we traverse up the tree. At every marked node along the path, we cut it from its parent and stick it in the root-list (with its subtree of course), causing the parent to become marked. The process finishes when we encounter an unmarked node, which then becomes marked. See Figure 13 below for a visual.

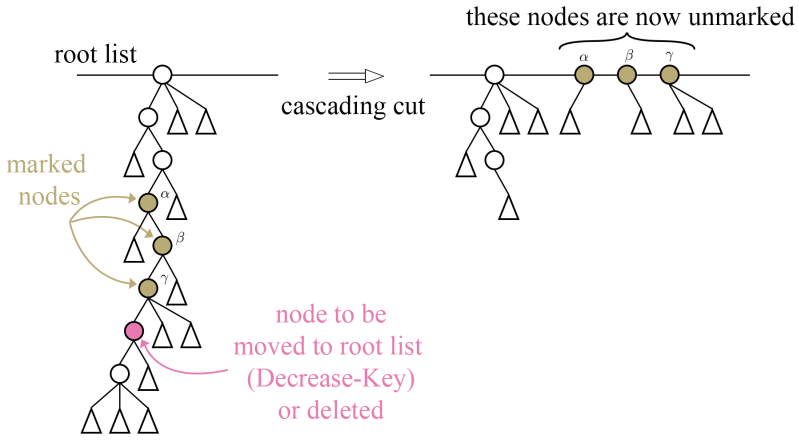

Figure 13: Illustrating the CASCADING-CUT operation.

Let k denote the number of marked nodes moved. Then we have:

$$\begin{aligned} \text{Actual Time} &= k \\ {}^{5}\Delta\Phi &= \alpha k - \beta k \quad (12) \\ \implies \text{Amortized Time} &= 0 \ \text{ if we set } \beta = \alpha + 1 \end{aligned}$$

Recall that we chose α to be 1 so we should take $\beta = 2$. As a result, this operation is free in amortized time so it is essentially harmless. Finally, we present the theorem (and proof) underlying the repeated assumption we made throughout this example.

[5] The root-list increases by k and k nodes become unmarked. Marking the node that triggers CASCADING-CUT to halt contributes to the number of marked nodes, but its contribution is negated by the starting node (pink in Figure 13) which becomes unmarked.

Theorem. *The maximum degree of a Fibonacci heap of size n is less than or equal to $\frac{\log_2 n}{\log_2 \varphi} \approx 1.44 \log_2 n$, where $\varphi = \frac{1+\sqrt{5}}{2}$.*

The number φ is the limit of the sequence of ratios of the terms in the famous Fibonacci sequence, hence the name Fibonacci heap.

Lemma. *The i-th child of a node x has degree greater than or equal to $i-2$.*

Proof. Let y denote the i-th child of x. When y was made a child of x, x already had at least $i-1$ children. Recalling the CLEAN-UP-ROOT-LIST operation, we know that y could only be made the child of x if they have the same degree. Thus the degree of y is at least $i-1$. Since y can only lose one child before CASCADING-CUT relocates it, we must have that the degree of y is greater than or equal to $i-2$. □

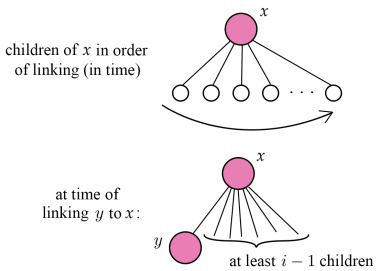

Figure 14: Visualizing the linking lemma.

Now for the proof of the theorem.

Proof. Let D_k denote the number of descendants of a node of degree k. We will show by induction on k that $D_k \geq \varphi^k$.

CASES:

1. $k = 0$: The node has 1 descendant (itself). $D_0 = 1 = \varphi^0$ so the inequality is satisfied.
2. $k = 1$: $D_1 \geq 2 \geq \varphi^1$ so the inequality is satisfied.
3. $k \geq 2$:

$$\begin{aligned} D_k &\geq D_{k-2} + D_{k-3} + \cdots + D_0 + 2 \\ &\geq \varphi^{k-2} + \varphi^{k-3} + \cdots + \varphi^0 + 2 \\ &= \frac{\varphi^{k-1} - 1}{\varphi - 1} + 2 \end{aligned} \tag{13}$$

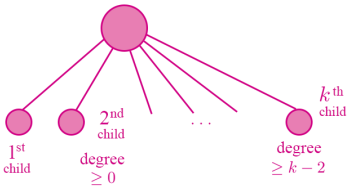

Figure 15: A node and its children, representing the first line of equation (13).

Here the first inequality holds by the lemma, the second inequality holds by the induction hypothesis, and the last equality comes from the finite geometric sum. The claim is that the final expression above is greater than or equal to φ^k.

$$\frac{\varphi^{k-1} - 1}{\varphi - 1} + 2 \geq \varphi^k \iff \varphi^{k-1} - 1 + 2\varphi - 2 \geq \varphi^{k+1} - \varphi^k$$

Bringing the φ^k term to the left-hand side and using the property that $\varphi^{k+1} = \varphi^k + \varphi^{k-1}$ we see that this is equivalent to the expression $2\varphi \geq 3$ which is true because $\varphi \approx 1.618$. Thus the claim is proved, and the inequality holds for all k. Now the relevant result is an immediate corollary. It is obvious that $D_k \leq n$, so we have $\varphi^k \leq D_k \leq n$. Taking the base-2 logarithm, then isolating for k in terms of $\log_2 n$ and $\log_2 \varphi$ yields the desired inequality. □

[6]The constant term "+2" in this line is attributed to the node itself and its first child, labelled in Figure 15.

References

T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009. ISBN 9780262033848.

M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

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

R.E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.