15-750: Graduate Algorithms January 20, 2016

Lecture 2: Amortized Analysis, Heaps and Binomial Heaps

Lecturer: Gary Miller *Scribe: Zhong Zhou, Daniel Keenan*

1 Amortized Analysis

This section is written with reference to CLRS Chapter 17, and Avrim Blum's Amortized Analysis notes.

1.1 Introduction

There are various time analysis methods: worst-case analysis, e.g., Strassen Algorithm for matrix multiplication, best-case analysis, and average case, e.g., Quick Sort pivoting on the first element. Amortized analysis is focused on worse-case input and average run time. For example, the randomized algorithm is focusing on worst-case input and average over coin flips like randomized quicksort. The problems we can apply amortized analysis is often the time where we have a series of operations, and the goal is to analyze the time taken per operation. For example, rather than being given a set of n items up front, we might have a series of n insert, lookup, and remove requests to some database, and we want these operations to be efficient.

Definition 1.1. Amortized Cost. The amortized cost per operation for a sequence of n operations is the total cost of the operations divided by n

There are generally three methods for performing amortized analysis: the aggregate method, the accounting method, and the potential method. All of these give the same answers, and their usage difference is primarily circumstantial and due to individual preference.

- Aggregate analysis determines the upper bound T(n) on the total cost of a sequence of n operations, then calculates the amortized cost to be T(n) / n.
- The accounting method determines the individual cost of each operation, combining its immediate execution time and its influence on the running time of future operations. Usually, many short-running operations accumulate a "debt" of unfavorable state in small increments, while rare long-running operations decrease it drastically.
- The potential method is like the accounting method, but overcharges operations early to compensate for undercharges later.

For our course, the focus is on Potential Method which we will look closely in the next section.

1.2 Potential Method

In the potential method, a function Φ is chosen that maps states of the data structure to non-negative numbers. If S is a state of the data structure, $\Phi(S)$ may be thought of intuitively as an amount of potential energy stored in that state; alternatively, $\Phi(S)$ may be thought of as representing the amount of disorder in state S or its distance from an ideal state. The potential value prior to the operation of initializing a data structure is defined to be zero. Intuitively, a potential function is much like a bank account: if we can take our cheap operations (those whose cost is less than our bound) and put our savings from them in a bank account, use our savings to pay for expensive operations (those whose cost is greater than our bound), and somehow guarantee that our account will never go negative, then we will have proven an amortized bound for our procedure.

Definition 1.2. Potential Method. A potential function is a function of the state of a system, that generally should be non-negative and start at 0, and is used to smooth out analysis of some algorithm or process.

Let o be any individual operation within a sequence of operations on some data structure, with S_{before} denoting the state of the data structure prior to operation o and S_{after} denoting its state after operation o has completed. Then, once Φ has been chosen, the amortized time for operation o is defined to be

$$T_{\text{amortized}}(o) = T_{\text{actual}}(o) + C \cdot (\Phi(S_{\text{after}}) - \Phi(S_{\text{before}})),$$

where C is a non-negative constant of proportionality (in units of time) that must remain fixed throughout the analysis. That is, the amortized time is defined to be the actual time taken by the operation plus C times the difference in potential caused by the operation.

2 Heap

2.1 Priority Queue: Eager to Lazy Algorithm

With Input keys $k_1, k_2, ..., k_n$ with priority $P(k_1), P(k_2)..., P(k_n)$, we have the following table for comparison of time analysis for priority queue:

Operation	Number of Ops	Worst-Case time per Op	Heap total cost
makeHeap(s)	1	1	1
findMin(s)	n	1	n
insert(k,s)	n	lgn	nlgn
deleteMin(s)	n	lgn	nlgn
decreaseKey(k,s)	m	lgn	mlgn

Note that our goal here is to improve the decreaseKey(k,s) time complexity. The idea is to replace worst case with average case. Faster priority queue's table is listed below.

Operation	Heap Total Cost	Binomial Heap Total Cost	Fibonacci Heap Total Cost
makeHeap(s)	1	1	1
findMin(s)	1	1	1
insert(k,s)	lgn	lgn	1
deleteMin(s)	lgn	lgn	lgn
unionMeld(s)	n	lgn	1
decreaseKey(k,s)	lgn	lgn	1

We see the advantage of binomial heaps and fibonacci heaps here. In the following sections, we are going to introduce more on heaps and binomial heaps.

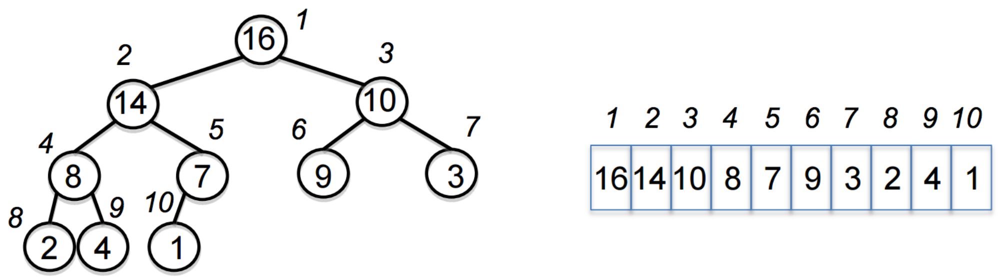

Figure 1: Example of a Heap

2.2 Formally introducing Heaps

Heaps is a partially ordered complete binary tree. It is a balanced binary trees with priority $P(k_1), P(k_2)..., P(k_n)$. It is the implementation of the priority queue. It is an array, visualized as a nearly complete binary tree

There are many properties of heaps:

- root of tree: first element in the array, corresponding to i = 1
- parent(i) = $i/2$: returns index of node's parent
- left(i)= $2i$: returns index of node's left child
- right(i)=$2i + 1$: returns index of node's right child
- Max Heap Property: The key of a node is larger or equal than the keys of its children. (Min Heap defined analogously)

2.3 Basic Heap Operations

2.3.1 Insertion

- Add a new node x to the parent y, a leaf in the tree
- if key(x) $\leq$ key (parent(x)), then stop
- if key(x) $<$ key (parent(x)), then trickle up
- repeat till the condition is met

2.3.2 Deletion

- Find a leaf node in the tree
- if the node to be deleted, x, is equal to y , then just delete it and stop
- if key(x) $<$ key (parent(x)), then trickle up
- if key(x) is higher than any of its children, then swap

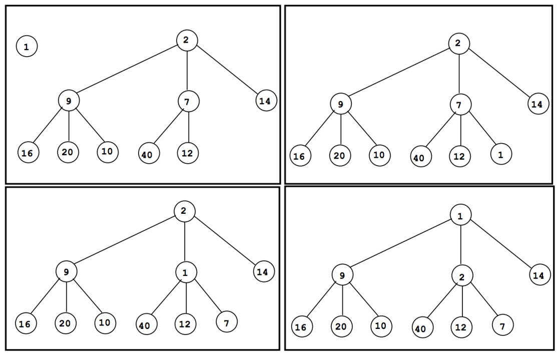

Figure 2: Example of a Heap Insertion

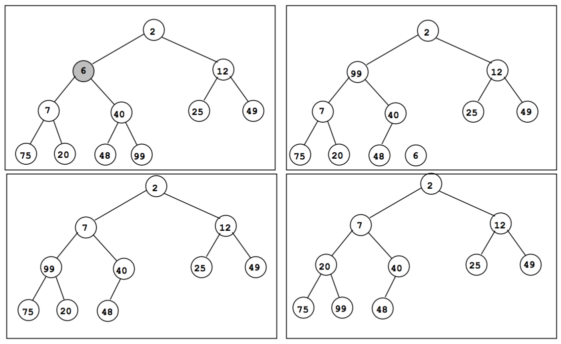

Figure 3: Example of a Heap Deletion

- repeat till the condition is met

The remaining parts of the Notes, we will look closely at Binomial Heap, more operations will be discussed in the following section.

3 Binomial Heap

To be finished: Thanks, Daniel