Advanced Data Structures Jan-Apr 2012

Lecture 1 January 4, 2012

Lecturer: Venkatesh Raman *Scribe: Venkatesh Raman*

1 Overview

We first looked at an overview of the course and discussed the requirements (see the separate sheet giving these details). Then we started on Self Adjusting data structures. We had a 'warm up' with 'skew heaps' and discussed amortized analysis using the potential method.

Finally, we started on self adjusting lists, and discussed a few self adjusting heuristics and their performance.

2 Self Adjusting Data Structures

We started on the topic of *self adjusting data structures* reviewing the slides of John Iacono in the talk he gave on this topic at the data structures workshop (www.imsc.res.in/dsmeet).

As discussed before, these are structures that don't keep any balance information, are easy to implement, but harder to analyze.

2.1 Skew Heaps

In particular, we looked at 'skew heaps' [3] which are basically binary trees with values in the nodes satisfying the 'min heap' property, with no other structural information.

2.1.1 Implementing Merge

In order to merge two skew heaps, we simply merge the elements of the right most path of the two heaps, make the merged list a left most path, making each of the subtrees which were left subtrees of the elements in the path before, the right subtrees of the corresponding elements in the resulting heap.

Insert can be implemented by creating a new heap with the single element and merging it with the given heap. Deletemin can be implemented by deleting the root and merging the two subtrees (which are self adjusting heaps themselves) using the procedure above.

See the figures in the slides of John Iacono's talk.

2.1.2 Amortized Analysis

Clearly in the worst case, a merge operation can take linear (in the number of nodes in the tree) time, as the rightmost paths can be linear in size. However, intuitively, if a merge operation takes a 'long' time, then the two trees have very little besides the right most paths, and part of them (whatever 'on the left') become the rightmost paths of the new resulting tree, so subsequent merge operations become cheaper.

Amortized complexity is a paradigm precisely to model this sort of behaviour. In particular, in amortized analysis, the aim is to analyse the (worst case) time for a (typically long) sequence of operations rather than for one operation. For example, if one is interested in the monthly expenses of a student, estimating it by thirty times her expense on a single day can be a gross overestimate. For example, she might have just paid her rent and food expenses on the first day. But the expense on that day is high precisely to avoid paying for them every day. I.e. because of the high expense on the single day (as food and rent have been paid for for the month), her expenses for other days are low.

We analyze the merge operation using the potential argument (see the chapter on Amortized complexity in [2]). In particular, the potential Φ of a data structure is simply a non negative number associated with the data structure.

Once we have a potential Φ, the amortized cost $\hat{a}$, of an operation is defined to be the actual time plus the potential of the data structure after the operation minus the potential of the data structure before. While it is often not easy to come up with the potential function, the following interpretations of the potential function can be of help.

- We can think of the amortized time as the amount of 'charge' each operation comes with to 'pay for' its execution. If it takes less time than the charge it brings, it keeps the balance charge in the data structure (which increases the potential of the data structure). The potential of the data structure is the charges kept in the structure by these operations. If an operation takes more time than the charge it brings, then it uses the charges in the data structure to 'pay for' its execution, thereby decreasing the potential of the data structure. The potential function we come up with should ensure that there is enough potential in the structure to pay for expensive operations.
- When we do the analysis for a sequence of operations, we undercount some operations and overcount some of them. We can think of the potential as the amount of 'overcounting' we have done so that those operations that are going to be undercounted can be compensated by them.

2.1.3 Amortized Analysis of Merge

To analyze the merge operation, we start with defining a node as 'right heavy' if the right subtree of the node has more than half the nodes in the subtree (rooted at that node), and call it 'right light' otherwise.

It is easy to observe that there can be at most $\log n$ 'right light' nodes in the right most path. Thus if we want to prove an amortized cost of $O(\log n)$ for the merge operation, we let the 'operation's

charge' pay for working on the 'right light' nodes (as there are only $O(\log n)$ of them) and let the potential pay for working on the 'right heavy' nodes.

This suggests using the number of right-heavy nodes in the tree as the potential of the data structure. But note that we need to make sure that the operation comes up with enough charges to keep on the 'extra' right-heavy nodes that are created after the operation. Fortunately, this is ensured as only the nodes that were right-light before can become right-heavy now, because we have switched the left and right subtrees after the merge. In fact, some of the right-light nodes may remain right-light as some extra subtrees may get added to their left. Thus the amortized complexity of the merge, the amount of the charge that the operation should come with, is $O(\log n)$ – to pay for working on the light nodes (which are $O(\log n)$ in number), and to pay for the newly created heavy nodes (which are $O(\log n)$ in number).

Hopefully this example illustrates the simplicity of the implementation of merge operation, and the subtlety of the analysis.

2.2 Self Organizing Lists

Consider the *dictionary* problem of maintaining a set of items to support the following three operations:

- **search(i)** – Locate item i in the set if exists.
- **insert(i)** – Insert item i in the set if it is already not there.
- **delete(i)** – Delete item i from the set if it is there.

We consider the implementation of this set of operations in a singly linked unsorted list. To search for item i, we simply start from the front of the list and scan the list until we find i. And to insert i, we simply insert it at the end of the list after ensuring that it is not there. To delete the list, we scan to find it and simply delete it.

We are interested in the total time for a sequence of operations, and let n be the maximum number of items ever in the set at any time.

If there are n elements in the list occupying the first n positions of the list, clearly the worst case time for a search operation is n. If every element is equally likely to be searched, then the average time for a search is $(n+1)/2$.

Suppose we know the frequency f_i of accessing item $i, 1 \leq i \leq n$. Then one way to minimize the total cost for the sequence of searches is to arrange the list in nonincreasing order of the frequencies, and it is easy to see that this order minimizes the total cost yielding a cost of $\sum_{i=1}^{n} i f_i$ where we call i as the item in the i-the location of this list (ordered by freqencies). Let us call this optimal cost the 'static optimal cost' (static because the access operations are not allowed to move the elements).

The question we would like to address is whether there is a way for 'self organizing the list' so that the total cost of the sequence of operations is close to the 'static optimal' even if we don't know the frequencies of access. By 'self organizing' we allow each of the operations to occasionally rearrange

the list (after access, insert or delete) by exchanging pairs of consecutive items to speed up later operations. Of course, we will charge a cost of 1 for each such exchange.

It is easy to see that in the worst case, an adversary can always request the last element in the list, resulting in a worst case cost of $\Omega(n)$. But we would like to compare this cost with respect to the cost of an 'optimal' 'offline' scheme that knows the request sequence in advance.

The following are some self-organizing heuristics that have been well studied.

- *Move-to-front (MF)*: After accessing or inserting an element, move it to the front of the list without changing the relative order of the other items.
- *Transpose (T)*: After accessing or inserting an item, exchange it with the immediately preceding item.
- *Frequency Count (FC)*: Maintain a frequency count for each item, which is initially 0 and is incremented whenever it is accessed or inserted (and reduced to 0 when deleted). Maintain the list so that the elements are in nonincreasing order by frequency count.

Before the work of Bentley and McGeoch[1] and Sleator and Tarjan[4], these heuristics were analysed for their asymptotic expected behaviour for various distributions of the access sequence. In particular, it was known that

- 'expected' behaviour of the transpose heuristic is at least as good as that of 'move to front',
- the expected behaviour of 'move to front' is at most twice that of the static optimal expectation, and
- the 'frequency count' as one can expect, converges asymptotically to static optimal over a long sequence.

However, if one is interested in the amortized complexity of these operations (i.e. the worst case time for a sequence of operations) without the knowledge of the frequencies of access, we show examples below that 'move to front' is significantly better than 'frequency count' and 'transpose' heuristics.

Consider the sequence that repeatedly accesses n and $n-1$ after inserting elements 1 to n in that order. After an insertion cost of $n(n+1)/2$, every other access costs n when we do 'transpose', resulting in an amortized cost of n for a long sequence of such accesses.

However, 'move to front' (and 'frequency count') heuristic will move n and $n-1$ to the front after their first accesses resulting an amortized cost of 1.5 for such a sequence.

Similarly, consider the sequence that accesses i, $k+n-i$ times, where k is some non-negative integer (after inserting items 1 to n in that order). The request sequence has length $nk + O(n^2)$. Frequency count (and the static optimal) heuristic will keep the sequence as they are, resulting in a total cost of $\sum_{i=1}^{n} i(k+n-i) = \Omega(kn^2)$, with an amortized cost of $O(n)$ for large enough k. However, the move to front heuristic will bring an element to the front after the first access resulting in a total cost of $nk + O(n^2)$ resulting in an amortized cost of $O(1)$.

Bentley and McGeoch[1] had shown that the cost of 'move to front' heuristic is at most twice the cost of the static optimal cost (for example, if an adversary keeps asking for the last element in the list, the static optimal can not do much better). In fact, Sleator and Tarjan[4] show something much stronger. The cost of 'move to front' heuristic for a (long) sequence of operations is at most twice the cost of *any* self-adjusting heuristic that even knows the future access sequence and can perform exchanges after every operation. We will show this (or a version of this) in the next class after defining the relevant notions precisely.

References

[1] J. Bentley and C. C. McGeoch, Amortized analyses of self-organizing sequential search heuristics, *Communications of the ACM* **28**:404-411, 1985.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, Introduction to Algorithms, Third edition, Prentice-Hall India Pvt Ltd (2010).

[3] D. D. Sleator and R. E. Tarjan, Self-Adjusting Heaps, *SIAM Journal on Computing***15**(1):52-69 (1986).

[4] D. D. Sleator and R. E. Tarjan, Amortized efficiency of list update and paging rules, *Communications of the ACM* **28**: 202-208 (1985).