6.897: Advanced Data Structures Spring 2005

Lecture 14 — March 31, 2005

Prof. Erik Demaine *Scribe: Jelani Nelson*

1 Overview

In the last lecture, we discussed algorithms for integer sorting. In this lecture we will discuss the relationship between integer sorting algorithms and priority queue data structures. It is obvious that fast priority queues imply fast sorting, since if a priority queue can do INSERT and DELETE-MIN in $O(f(n))$, then we can sort in $O(nf(n))$ by inserting all elements into the queue then calling DELETE-MIN n times. So, we will focus in this lecture on the reverse — that is, how to make fast priority queues once we have fast sorting algorithms.

2 Priority Queue

A priority queue, also sometimes called a *heap*, is a data structure that performs the following operations:

- INSERT(x, key) – Insert the element x into the heap with key key.
- DELETE-MIN() – Return the smallest element in the heap under key order. Also, delete this element from the heap.
- DECREASE-KEY(x, key) – Change the key of item x in the heap to key. key must not be greater than x's current key value.
- MELD(H_1, H_2) – Takes two heaps as input and produces a new heap that contains all the elements of the two that are input.

The fundamental operations are INSERT and DELETE-MIN. The other two operations are motivated by applications to graph algorithms.

Two old but very important algorithms in computer science that require heaps are Dijkstra's shortest path algorithm in [1], and Prim's algorithm for minimum spanning tree, discussed in [2]. Note one difference in the types of priority queues needed in the two algorithms: in Dijkstra's algorithm the minimum key in the heap never decreases. We call heaps that only support such situations *monotone*. If we remember the way Dijkstra's algorithms and Prim's algorithm work, each of them performs $O(m)$ decrease-key operations, and $O(n)$ INSERT's and DELETE's. This motivates the need for DECREASE-KEY being faster than DELETE-MIN.

Another old algorithm that uses the priority queue is Edmonds' algorithm for directed MST [3]. This can be implemented efficiently using meldable priority queues.

2.1 Results

For the following results, you can assume that a fast MELD is not supported unless otherwise stated. In the comparison model, a normal binary heap can achieve $O(\log n)$ per operation. Fibonacci heaps, introduced in [4], achieve $O(1)$ DECREASE-KEY and Insert and $O(\log n)$ DELETE-MIN (all times amortized). Note, this implies an $O(n \log n + m)$ running time for Prim's and Dijkstra's algorithms. A worst-case version of the Fibonacci heap, which also achieves $O(1)$ inserts, was shown in [5].

For integer priority queues, the atomic heap was introduced in [6], which achieves $O(1)$ amortized time per operation for $O(\text{polylog}n)$ elements. From a high-level point of view, the atomic heap works much like a fusion tree (and thus even supports predecessor and successor queries). The paper discusses how to use this heap for an optimal $O(n + m)$ MST algorithm (note that this is not through direct implementation of any classic algorithm). The only linear time algorithm for MST known in the comparison model is randomized, found in [7]. The best deterministic MST algorithm, due to [8], runs in time $O((n + m)\alpha(n))$.

As far as relating sorting to priority queues, it was shown in [10] that $O(n \cdot S(n, w))$ sorting implies an $O(S(n, w))$ amortized per operation monotone priority queue. This result was later improved to worst-case and without the monotonicity condition in [11]. Work has also been done in showing how to use a fast priority queue to get a fast meldable priority queue. It is shown in [13] that an $O(P(n, w))$ time per operation priority queue implies an $O(P(n, w)\alpha(n))$ time per operation meldable priority queue. The $\alpha(n)$ was essentially removed (disappears for the $P(n, w)$ we currently know and care about) in [14].

The best known priority queue with constant DECREASE-KEY does not match the current sorting bound. Rather, it achieves $O(\lg \lg n)$ per DELETE-MIN [12]. Beside MST, another important application of such queues was solved through different techniques: single-source shortest paths were shown to be computable in linear time, $O(m + n)$, in [9]. However, this is only for the undirected case, and the directed version still relies of priority queues with constant DECREASE-KEY.

The rest of this lecture discusses what appears to be a simpler way to convert $O(n \cdot S(n, w))$ sorting to priority queues with operations supported in $O(S(n, w))$ time. This is an ad-hoc creation of your teachning staff and has not been checked extensively, so it should be trusted accordingly.

2.1.1 The Priority Queue

We will keep $\Theta(\lg n)$ buckets, $B_0, B_1, \ldots$. Each B_i will either be:

- Empty
- Sorted and of size at most 2^i, stored in a linked list (we will charge to pay for the sort).
- Semi-structured:
  - Unstructured part U_i of arbitrary size, stored in a linked list (we will be able to charge the elements of U_i).

- Structured part S_i split into $\Omega(2^{i/2})$ groups of size $\Theta(2^{i/2})$ each. Each group is unsorted, but the groups are in order. In our charging argument, we will charge for the groups beyond $\Theta(2^{i/2})$.

We will always keep B_0 semi-structured. For semi-structured buckets B_i, B_j, if $i < j$ then all elements of B_i will be less than all elements of B_j. For sorted B_i, we will maintain the invariant that $\min(B_{i-1}) < \max(B_i) < \min(B_{i+1})$.

We will keep an atomic heap on $\min(B_i)$ for all non-empty B_i to find which bucket contains the element with minimum key. Recall that atomic heaps have $O(1)$ time per operation on $O(\text{polylog}n)$ elements, and here there are $\Theta(\log n)$ buckets. DECREASE-KEY will remove from the element from the existing bucket then call INSERT again.

For DELETE-MIN:

- Find the lowest non-empty bucket B_i.
- if B_i is sorted, simply extract its minimum from the linked list.
- if B_i is semi-structured:
  - if $|B_i| = O(2^i)$, then sort everything and turn it into a sorted bucket.
  - if $|B_i| = \omega(2^i)$, then we do an "exponential blastoff"
    - for $j = i, i+1, ...$
      - Put elements of B_j into a bag.
      - Drop the smallest $\Theta(2^j)$ elements from the bag into B_j (we figure out which are smallest by going through the elements a first time and doing recursive medians on the set).
      - Stop when the bag is empty.

Note that the recursive medians on the set above runs in linear time. This is because the sizes of the B_j increase as a geometric series. So, our time for the recursive medians is the sum of a geometric series multiplied by $O(n)$, which is just $O(n)$. It was shown in [15] that structured parts can be built from unstructured parts in $O(2^i)$ time. Also, we can split and merge structured parts in $O(2^{i/2})$ time.

References

[1] E.W. Dijkstra. A note on two problems in connection with graphs. In Numerische Mathematik, pages 1:269-271, 1959.

[2] R.C. Prim. Shortest connection networks and some generalizations. Bell Sys. Tech. Journal, pages 1389–1401, 1957.

[3] J. Edmonds, "Optimum branchings", J. Research of the National Bureau of Standards, 71B, 1967, pp.233-240.

[4] Michael L. Fredman, Robert Endre Tarjan: Fibonacci heaps and their uses in improved network optimization algorithms. J. ACM 34(3): 596-615 (1987)

[5] Gerth Stlting Brodal: Worst-Case Efficient Priority Queues. SODA 1996: 52-58

[6] Michael L. Fredman, Dan E. Willard: Trans-Dichotomous Algorithms for Minimum Spanning Trees and Shortest Paths. J. Comput. Syst. Sci. 48(3): 533-551 (1994)

[7] David R. Karger, Philip N. Klein, Robert Endre Tarjan: A Randomized Linear-Time Algorithm to Find Minimum Spanning Trees. J. ACM 42(2): 321-328 (1995)

[8] Bernard Chazelle: A minimum spanning tree algorithm with Inverse-Ackermann type complexity. J. ACM 47(6): 1028-1047 (2000)

[9] Mikkel Thorup: Undirected Single-Source Shortest Paths with Positive Integer Weights in Linear Time. J. ACM 46(3): 362-394 (1999)

[10] Mikkel Thorup: On RAM Priority Queues. SIAM J. Comput. 30(1): 86-109 (2000)

[11] Mikkel Thorup: Equivalence between Priority Queues and Sorting. FOCS 2002: 125-134

[12] Mikkel Thorup: Integer priority queues with decrease key in constant time and the single source shortest paths problem. STOC 2003: 149-158

[13] Ran Mendelson, Mikkel Thorup, Uri Zwick: Meldable RAM priority queues and minimum directed spanning trees. SODA 2004: 40-48

[14] Ran Mendelson, Robert Endre Tarjan, Mikkel Thorup, Uri Zwick: Melding Priority Queues. SWAT 2004: 223-235

[15] Yijie Han, Mikkel Thorup: Integer Sorting in 0(n sqrt (log log n)) Expected Time and Linear Space. FOCS 2002: 135-144