The UNIVERSITY of VERMONT

Benjamin Fenster
CS-104 (Lee)
27 May 2004

Heaps

I. Priority Queue
   a. Introduction
      i. This is a specialized form of the general queue
      ii. It adds more interface features.
      iii. Instead of .Get() and .remove(), implement .findMin(), .RemoveMin().
      iv. Could have min-ordered or max-ordered priority queues.
   b. Runtimes
      i. Using an ordered array:
         1. Insert = O(N)
         2. deleteMin = O(N)
         3. Must shift all elements out of the way for each operation.
      ii. Linked List
         1. Insert = O(N)
         2. deleteMin = O(1).
      iii. Using a binary search tree
         1. Average insert = O(log N), worst insert = O(N)
         2. Average deleteMin = O(log N), worst deleteMin = O(N)
      iv. Using a binary heap
         1. Average insert = O(1), worst insert = O(log N)
         2. Average deleteMin = O(log N), worst insert = O(log N).
         3. There's no case with linear runtime.
         4. This is *the* data structure for implementing a priority queue – to the point that the terms are sometimes used interchangeably.

II. Binary Heaps
   a. A heap-ordered complete binary tree
      i. Complete means that every level is full except perhaps the leaf level.
      ii. Heap-ordered property for min-heap means:
         1. For each parent-child pair, parent ≤ child.
         2. This enforces a partial ordering: not ALL elements have the order relationship, just parents and children.
   b. This is typically implemented as an array
      i. parent(a[I]) = a[i/2]
      ii. children(a[I]) = a[2 * i] and a[2 * i + 1]
   c. Heapify Operations
      i. Fix violations of the heap-order property
         1. percolate up, fix up, bubble up (synonyms)
            a. O(log N)
            b. Have a parent-child pair where parent > child.
            c. Swap them!
            d. Swap violating node with its parent – percolating from the bottom up.
         2. percolate down, et cetera.
            a. O(log N)
            b. Swap the smallest child with the parent.
            c. Top-down
      ii. The heapify operations are private – they are needed by other operations, but not by client code.
   d. Interface Operations
      i. insert(Item n). Insert at the end of the heap (a new leaf), then percolate it up to its proper position.
      ii. findMin(). It's always at the root. Easy.
      iii. deleteMin().
         1. Delete the root.

2. Take a node from the end of the heap, place it at the root
3. Percolate down.

iv. delete(Item n)
1. Remove the given item.
2. Replace it with the last of the leaf nodes.
3. Percolate down.

v. decreasePriotiy(Item n, int d); Change the priority to d, percolate up.

vi. increasePriority(Item n, int d): Change priority, percolate down.

e. BuildHeap Operation
i. Not an interface operation.
ii. Build a complete binary tree in any order (fill up the array in any order)
iii. Then percolate nodes down starting with the last node that has a child, working back to the root.

f. Runtime
i. insert(n) = O(log N)
ii. findMin() = O(1)
iii. deleteMin, delete, increasePriority, decreasePriotiy = O(log N)
iv. buildHeap with N nodes: O(N)
1. Need to prove linearity
2. To percolate a node down takes (2h) comparisons where h is the height of the node. One comparison for each node at each level.
3. Total number of comparisons is bounded by twice the height of all nodes (N – log(N + 1)).
a. Worst case (total height max) is when the heap is completely full (and balanced) – the sum of the height of all nodes).
b. $h = 2^h$ at 0, 2^{h-1} at 1, 2^{h-2} at 2, 2^2 at h – 2, 2 at h – 1, 1 at h.
c. Sum of all heights = $S = 2^{h+1} - 1$.
i. $S = h + 2(h - 1) + 2^2(h - 2) + 2^3(h - 3) + \ldots + 2^{h-1}(1)$
ii. $2S = 2h + 2^2(h - 1) + 2^3(h - 2) + \ldots + 2^h - h$
iii. Then $S = -h + 2 + 2^2 + 2^3 + \ldots + 2^h$
iv. $S = (-h - 1) + (1 + 2 + 2^2 + \ldots + 2^h)$
v. $S = 2^{h+1} - 1 - (h + 1) = N - \log(N + 1)$

g. select(k)
i. Select the kth largest or smallest element out of a set of N elements.
ii. One way is to buildHeap() and then deleteMin k times.
iii. Runtime
1. O(N) + kO (log N) = O(N + k log N)
2. = O(N + N log N) because $k \leq N$.
3. = O(N log N)
4. A more precise analysis may yield a smaller bound but not likely.

III. d-heaps
a. A direct extension of binary heaps
b. d-ary heaps just have d > 2
c. Otherwise they're the same as with binary heaps
d. Array representation
i. parent(a[I]) = a[floor((i + d – 2) / d)]
ii. children(a[I]) = a[(i – 1) * d + 2] through a[(i)d + 1]
iii. Example for d = 3: i = 3, children = 8, …, 10
iv. Get the number of first and last child, then iterate between.
e. Runtime is $\log_d N$ instead of $\log_2 N$.
f. Dispite the runtime advantage it's harder to find the smallest child among three children than between two.
g. With d = 4, 8, 2^n, node numbers can be calculated (for parent to child and vice versa) by bit shifting.

IV. Merging Priority Queues

a. Using binary heaps, the runtime of the merge is $O(N^1 + N^2) = O(N)$ where $N = \max(N_1, N_2)$
b. Do this by appending the two arrays, then doing buildHeap().
c. Using leftist heaps, skew heaps, binomial queues, the merge takes O(log N). A clear advantage!

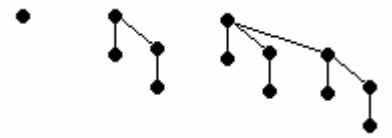
V. Leftist Tree
a. A binary tree with the "leftist property"
b. Leftist property: for each node v: rank(lchild(v) ≥ rank(rchild(v))
    i. rank(x) is the path from x to the nearest non-full (1 or 0 children) node.
    ii. rank is the same as 'null path length'
    iii. null nodes have rank –1.
c. Properties
    i. Rightmost path for any node is the shortest possible path length.
    ii. Thus the tree is skewed to the left.
    iii. Lemma: Given a tree with r nodes on the rightmost path, the minimum total number of nodes will occur if the tree is completely full. (See slides for proof)
    iv. Theorem: A leftist tree with r nodes on the rightmost path has at least $2^r - 1$ nodes total.
        1. Proof: The minimum number of nodes occurs if the tree is completely full. The number of nodes is $2^{h+1} - 1$ where $h = r - 1$ (convert from the number of nodes to the height).
        2. Number of nodes = $2^r - 1$
    v. Corollary: A leftist tree with N nodes has at most floor(log(N + 1) nodes on the rightmost path. Proof: This is immediate from the previous theorem.

VI. Leftist Heap
a. This is a heap-ordered leftist tree: has both heap-ordered property and leftist property
b. Typically represented with linked node (lchild, rchild) structure. Item key, int rank, *lchild, *rchild)
    i. Why? The tree is not complete. It's skewed significantly to the left, so a lot of space would be wasted in the array if it were represented as an array.
    ii. Even if storage is cheap, there's a bigger concern.
    iii. The merge operation requires swapping subtrees, which would require a lot of array element copies. Just switching two pointers is easier by far!
c. Merge Operation
    i. Algorithm
        1. Given pointers to two leftist heaps ($*h_1$, $*h_2$)
        2. If h_1 is null, return h_2
        3. If h_2 is null, return h_1
        4. If key(h_1) > key(h_2) then swap(h_1, h_2)
        5. h_1 = merge(h_1->rchild, h_2)
        6. If rank(h_1->lchild) < rank(h_1->rchild) then swap(l, r).
        7. return h_1.
    ii. Runtime
        1. Merge heaps with N_1, N_2 nodes
        2. Runtime O(log N) where $N = \max(N_1, N_2)$
        3. Proof
            a. Steps at each invocation of merge() are constant.
            b. Runtime is proportional to the number of recursive calls.
            c. That is bounded by the number of nodes on the rightmost paths of the two heaps, since merge() is called along that path.
            d. By corollary (earlier), these numbers are floor(log (N_1 + 1)) and floor(log(N_2 + 1)).
            e. sum = $O(\log N_1 + \log N_2) = O(\log N)$ where $N = \max(N_1, N_2)$
    iii. Other Operations (Runtime)

1. findMin() O(1)
2. h.insert(x) ≡ merge(h, x) O(log N)
3. h.deleteMin() ≡ merge(h->left, h->right) O(log N)
4. buildHeap() O(N)
    a. Consider each element as a one-node leftist heap
    b. Put in queue
    c. Dequeue two, merge them, put the result back in the queue
    d. Keep merging until only one heap remains

VII. Skew Heaps
- a. Heap-ordered binary tree with no structural property
- b. Not a binary heap (not stored in an array)
- c. Not leftist heap (doesn't have leftist tree proprety)
- d. Runtime
    - i. Rightmost path can be arbitrarily long, so worst-case runtime is O(N) per operation.
    - ii. When performing many (M) operations, O(M log N)
    - iii. So amortized per-operation runtime is O(log N)
- e. Merge Operation
    - i. Exactly the same as for leftist heaps, but ALWAYS swap the left and right children (unconditionally)
    - ii. This *tends* to skew the heap to the left.
- f. Other Operations
    - i. findMin() is easy
    - ii. insert(), merge()
        1. Same as leftist heap
        2. insert(key) ≡ merge(heap, key)
    - iii. deleteMin()

VIII. Binomial Queues
- a. Merge takes O(log N) time, not O(N)
- b. Heap-ordered tree (not necessarily binary)
- c. Binomial tree B_k has height k formed by attaching two B_{k-1} trees together
    - i.



    - ii. The number of nodes at some level l of B_k is the binomial coefficient C(k, l) = k! / (l! (k-l)!)
        1. Permutation
            - a. P(k, l) is the number of possible ordered combinations of l objects from k. P(k, l) = k!/ (k – l)!
            - b. Pick 1: k choices
            - c. Pick another: (k – 1) choices
            - d. Total for l selections = k * (k – 1) * (k – 2) * … * (k – l + 1)
        2. Combination
            - a. Unordered combinations
            - b. C(k, l) = k! / (l! (k – 1)!)
- d. Binomial Queues Introduction
    - i. Definition: A forest of binomial queues of distinct heights (note that the order doesn't matter).
    - ii. Representing Priority Queue
        1. Let N be the number of elements in the queue
        2. For each "1" bit in the binary representation of N at position p, need a binomial tree B_p
        3. $N = 13_{10} = 1101_2$ so need B_0, B_2, B_3

e. Property: Binomial queue of N elements has at most log(N + 1) binomial trees
   i. Proof: Consider n BTs storing N elements
   ii. N I sminimum if binary representation is 1111…11 (n bits)
   iii. So $N \geq 1 + 2 + 2^2 + \ldots + 2^{n-1}$
   iv. $N \geq 2^n - 1$ and $2^n \leq N + 1$
   v. So $n \leq \log (N + 1)$ QED

f. Implementation
   i. Array of binomial trees where arr[i] = B_i
   ii. Implement tree as binary using *child and *sibling pointers.

g. Merge
   i. Just combine all the trees into one forest.
   ii. If two trees have the same height, attach the tree with the larger root to the tree with the smaller root.
   iii. If that action causes two trees to have the same height, keep combining.
   iv. NB: Summing the binary representation of the numbers of nodes in each of the two queues yields the total number of nodes. The binary representation of that sum should therefore match the new set of trees.
   v. Runtime: log N where N = max(N_1, N_2) for the two binomial queues.
      1. Worst case is where both queues have all the same heights
      2. Best case: O(1)

h. Other Operations
   i. findMin() O(log N)
   ii. insert(x) O(log N)
   iii. merge(x, bq) O(log N)
   iv. deleteMIN() O(log N)
      1. Find B_i whose root is min O (log N)
      2. Remove B_i from the queue O(1)
      3. Remove the root, thus separating all its children O(1)
      4. Merge those children and the original queue O(log N)
      5. Children will always be {B_0, B_1, …, B_{i-1}}

IX. Heaps Runtime Summary

	findMin	deleteMin	insert	buildHeap	merge
Binary Heap	O(1)	O(log N)	O(log N)	O(N)	O(N)
D-ary Heap	O(1)	O(log N)	O(log N)	O(N)	O(N)
Leftist Heap	O(1)	O(log N)	O(log N)	O(N)	O(log N)
Skew Heap	O(1)	O(log N)	O(log N)	O(N)	O(log N)
Binomial Queue	O(log N)	O(log N)	O(log N0	O(N)	O(log N)