Algorithms and Data Structures

Priority Queue

(c) Marcin Sydow

Topics covered by this lecture:

- Priority Queue
- Naive Implementations
- Binary Heap
- HeapSort and other examples
- Extended Priority Queues
- (*) Binomial Heap

Priority Queue Definition

Priority Queue (PQ) is an abstract data structure supporting the following operations:

- insert(T e) // add to PQ a new element with assigned priority
- T findMin() // return the element with minimum priority
- T delMin() // return and delete the elt. with min. prior.

(optional operation: construct(sequence<T> s) // create a PQ given set of elements)

Each element has associated "priority".

One can also consider a "max"-type priority queue, defined analogously

Note: `priority queue` is not a "specialised" `queue`

(why? hint: remember the definition of `queue`)

Implementations of Priority Queue

The implementations evolve, for example:

- "naive" (as an array or list)
- Binary Heap (1964 Williams; Floyd 1964)
- Binomial Heap (1978 Vuillemin)
- Pairing Heap* (1986 Fredman, Sedgewick, Sleator, Tarjan; 2000 Iacono; Pettie 2005)
- Fibonacci Heap* (1987 Fredman, Tarjan)
- "Thin" Heaps and "Fat" Heaps* (1999 Kaplan, Tarjan)

* - not in this lecture

Naive implementations

- unsorted sequence:
  insert: $O(1)$, deleteMin: $O(n)$, construct: $O(n)$
- sorted sequence:
  insert: $O(n)$, deleteMin: $O(1)$: construct: $O(n \log(n))$

(sequence can be an array or linked list)

Binary Heap

Binary Heap is a *complete*[1] binary tree satisfying the following **"heap-order condition"** (for each (non-root) node x):

(priority of $parent[x]$) $\leq$ (priority of x)

Observations:

- minimum priority is at root
- priorities on each path from the root to a leaf form a non-decreasing sequence
- height of n-element binary heap is $\Theta(log(n))$ (due to completeness)

(there is also a "max" variant of the above definition)

[1] Leaves (except may be the right-most ones, that can be 1 level higher), have equal depths

Array Representation of Binary Heap

Due to the fact that it is a **complete** binary tree, Binary Heap can be compactly represented as an array:

Navigation:

(assume the root is under index 1)

- $parent[i] == i/2$ (for non-root i)
- $i.left == 2i$, $i.right == 2i + 1$ (for non-leaf i)

Restoring the Heap Order

Two helper, internal operations.
Both assume the heap order is correct except the position i:

- upheap(i) (call when (key of parent[i] > key of i), assert: heap ok below i): the key under i goes up until ok
- downheap(i) (call when one of children of i has lower key than i, assert: heap ok above i and for both its subtrees): the key under i goes down until ok

Both operations use $O(log(n))$ key comparisons (n - number of elements)

Upheap

Example of upheap(i) implementation:

```
upheap(i)        // i > 0, heap ok under i
  key = heap[i]
  parent = i/2
  while((parent > 0) && (heap[parent] > key))
    heap[i] = heap[parent]
    i = parent
    parent /= 2
  heap[i] = key
```

Downheap

Example of downheap(i) implementation:

```
downheap(i)
  l = 2i          // left son
  r = 2i + 1      // right son
  if l <= n and heap[l] < heap[i]:
    min = l
  else:
    min = i
  if r <= n and heap[r] < heap[min]: // n is the size of heap
    min = r
  if min != i:
    swap(i,min)   // swap the elements under indexes (not indexes themse
    downheap(min) // go down
```

Remarks: recursion used here only for keeping the code short, swapping too. Both can be avoided to make the implementation more efficient.

Priority Queue implemented on Binary Heap

(data size: number of elements (n), dom. oper.: comparison of priorities)

- insert(x): add x to the bottom and upheap(bottom) $(O(log(n)))$
- findMin(): return root $(O(1))$
- delMin(): move the bottom element to the root and downheap(root) $(O(log(n)))$

What is the complexity of `construct`?

Priority Queue implemented on Binary Heap

(data size: number of elements (n), dom. oper.: comparison of priorities)

- insert(x): add x to the bottom and upheap(bottom) ($O(log(n))$)
- findMin(): return root ($O(1)$)
- delMin(): move the bottom element to the root and downheap(root) ($O(log(n))$)

What is the complexity of `construct`?

Interestingly, `construct` has fast, $\Theta(n)$ implementation.

Complexity of construct

(naive: $n \times insert$ (which gives $\Theta(nlog(n))$))
faster way:

```
for(i = n/2; i > 0; i--) downHeap(i)
```

Analysis: downHeap is called at most 2^d times for nodes of depth d, each such call costs $O(h-d)$ (where h is the height of heap). Thus, the total cost is:

$$O(\sum_{0 \leq d < h} 2^d(h-d)) = O(2^h \sum_{0 \leq d < h} \frac{h-d}{2^{h-d}}) = O(2^h \sum_{j \geq 1} \frac{j}{2^j}) = O(n)$$

(the last equation holds because: $\sum_{i \geq 1} i2^{-i} = 2$)

$\sum_{i\geq 1} i2^{-i} = 2$: a proof

finite geometric series: $\sum_{i=0}^{n-1} q^i = \frac{1-q^n}{1-q}$ for $q \neq 1$

infinite geometric series:
$\sum_{i\geq 0} q^i = lim_{n\to\infty} \frac{1-q^n}{1-q} = \frac{1}{1-q}$ for $0 \leq q < 1$

Thus:
$\sum_{i\geq 0} 2^{-i} = 1 + 1/2 + 1/4 + \ldots = 2$ (a geometric series with $q = 1/2$)

Now:[2]

$$\sum_{i\geq 1} i2^{-i} = \sum_{i\geq 1} 2^{-i} + \sum_{i\geq 2} 2^{-i} + \sum_{i\geq 3} 2^{-i} + \ldots = (1+1/2+1/4+\ldots) = 2$$

(the first equality is due to the re-grouping of terms (2^{-i} occurs in exactly i first sums))

[2] More generally, $\sum_{k\geq 0} kx^k = \frac{x}{(1-x)^2}$, for any $|x| < 1$ (take derivative of infinite geometric series to obtain it)

Example: HeapSort

How to sort a sequence s with a Priority Queue?

Example: HeapSort

How to sort a sequence s with a Priority Queue?
(pq is an object representing priority queue)

```
while(s.hasNext())
  pq.insert(s.next())

while(!pq.isEmpty())
  result.add(pq.delMin())
```

data size: # elements (n), dom. op.: comparison
time complexity: $\Theta(nlog(n))$, space complexity: $\Theta(n)$

Notice: if we put the min element to the last released place in the array, we obtain $O(1)$ space complexity!

Other examples of applications of priority queues

Priority queues are typically used in *greedy* algorithms (for selecting a next element in the solution in the efficient way), for example:

- Huffman Code computation
- Dijkstra's shortest-path algorithm (on other lecture)
- Prim's minimum spanning tree algorithm (on other lecture)
- etc.

Extensions of Priority Queue

Addressable Priority Queue

- construct(sequence<T> s)
- H insert(T e) // as before but returns a *handle* to the inserted element
- T findMin()
- T delMin()
- decreaseKey(H pointer, T newPriority)
- delete (H pointer)

In addition: Mergeable Priority Queue:

- merge(PQ priorityQueue1, PQ priorityQueue2)

Complexities of Priority Queue Operations

operation	unsort.	sort.	binary heap	**binomial heap**
insert	1	n	lg n	lg n
findMin	n	1	1	lg n
delMin	n	1	lg n	lg n
decreaseKey	1	n	lg n	lg n
delete	1	n	lg n	lg n
merge	1	n	**n**	**lg n**

(the entries have implicit $O(.)$ around them)

Binomial Heap: a bit more advanced implementation of priority queue that supports fast merge (and keeps other operations fast)

Binomial Trees

A *binomial tree* B_i of degree i is a rooted tree defined recursively as follows:

- B_0 consists of a single node
- B_i: the root has i sons: $B_{i-1}, B_{i-2}, ..., B_0$ (in such order)

Properties of B_i:

- height: i
- has exactly $\binom{i}{j}$ (binomial coefficient) nodes at level j
- has exactly 2^i nodes in total
- can be obtained by adding a B_{i-1} as a left-most son of a root of a B_{i-1}

(Notice the analogies with the properties of binomial coefficients!)

Binomial Heap

Binomial Heap is a list of binomial trees sorted decreasingly by degrees (from left to right), where each binomial tree satisfies the *heap order*.

Properties of a n-element binomial heap:

- it consists of $O(logn)$ binomial trees
- B_i is its part only if the $i-th$ bit in the binary representation of n is set to 1

(both properties are implied by the fact that $|B_i| == 2^i$ and properties of binary representation of numbers)

Operations on Binomial Heap

- findMin: min is among the $O(logn)$ roots
- merge: similar to adding two binary numbers. Summing two 'ones' on position i: merging two binomial trees B_i to obtain one B_{i+1} (remind the last property of binomial trees). The tree with lower key becomes the root, the other becomes its right-most son. The summing goes through both lists of binomial trees from left to right (thus it has $O(logn)$ complexity)
- insert: merge with a 1-element binomial heap ($O(logn)$)
- delMin: find the root with the lowest key ($O(logn)$), cut it out, merge the list of its sons (being a correct binomial heap itself!) with the rest of the remaining part ($O(logn)$)
- decreaseKey: similarly as in binary heap ($O(logn)$)
- delete: move it to the root, cut, then as in delMin ($O(logn)$)

Questions/Problems:

- Definition of Priority Queue
- Complexities on naive implementation (list, array)
- Binary Heap definition
- Binary Heap represented as Array
- Priority Queue operations on Binary Heap (+ complexities)
- HeapSort and other examples of applications
- Extended Priority Queues (operations)
- (*) Binomial Trees and Binomial Heap
- (*) Priority Queue implemented on Binomial Heap (operations + complexities)

Thank you for your attention