Lecture Notes on Data Structures and Algorithms: Priority Queues

Conrado Martínez
U. Politècnica Catalunya

February 17, 2016

Part IV

Priority Queues

1. Priority Queues
2. Heapsort

Introduction

A priority queue (cat: *cua de prioritat*; esp: *cola de prioridad*) stores a collection of elements, each one endowed with a value called its priority.

Priority queues support the insertion of new elements and the query and removal of an element of minimum (or maximum) priority.

Introduction

```
template <typename Elem, typename Prio>
class PriorityQueue {
public:
   ...
// Adds an element x with priority p to the priority queue.
void insert(cons Elem& x, const Prio& p);

// Returns an element of minimum priority; throws an
// exception if the queue is empty.
Elem min() const;

// Returns the priority of an element of minimum priority; throws an
// exception if the queue is empty.
Prio min_prio() const;

// Removes an element of minimum priority; throws an
// exception if the queue is empty.
void remove_min();

// Returns true iff the priority queue is empty
bool empty() const;
};
```

Introduction

```
// We have two arrays Weight and Symb with the atomic
// weights and the symbols of n chemical elements, e.g.,
// Symb[i] = "C" y Weight[i] = 12.2, for some i.
// We use a priority queue to sort the information in alphabetic
// ascending order

PriorityQueue<double, string> P;
for (int i = 0; i < n; ++i)
   P.insert(Weigth[i], Symb[i]);
int i = 0;
while(not P.empty()) {
   Weight[i] = P.min();
   Symb[i] = P.min_prio();
   ++i;
   P.remove_min();
}
```

Introduction

We can use the kth smallest element in an unsorted array. Insert the first k elements in a max-priority queue. For each remaining element, compare the current element with the maximum element in the PQ; if it is larger or equal then continue, if it is smaller, remove the maximum and insert the current element in the PQ.

The priority queue keeps, after the first k insertions, the k smallest elements seen so far in the sequence. Its maximum element is the kth smallest element.

Introduction

- Several techniques that we have seen for the implementation of dictionaries can also be used for priority queues (not hash tables).
- For instance, AVLs can be used to implement a PQ with cost $\mathcal{O}(\log n)$ for insertions and deletions

Heaps

Definition

A heap is a binary tree such that

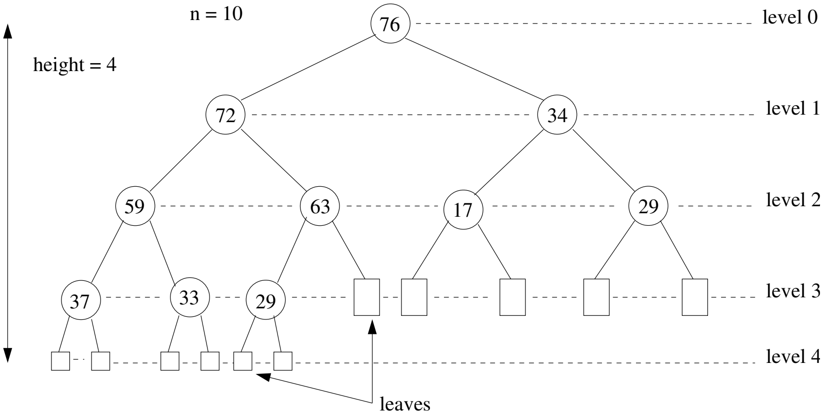
1. All empty subtrees are located in the last two levels of the tree.
2. If a node has an empty left subtree then its right subtree is also empty.
3. The priority of any element is larger or equal than the priority of any element in its descendants.

Heaps

Because of properties 1–2 in the definition, a heap is a quasi-complete binary tree. Property #3 is called heap order (for max-heaps).

If the priority of an element is smaller or equal than that of its descendants then we talk about min-heaps.

Heaps


n = 10
height = 4
76
72
34
59
63
17
29
37
33
29
level 0
level 1
level 2
level 3
level 4
leaves

Heaps

Proposition

1. *The root of a max-heap stores an element of maximum priority.*
2. *A heap of size n has height*

$$h = \lceil \log_2(n+1) \rceil .$$

If heaps are used to implement a PQ the query for a max/min element and its priority is trivial: we need only to examine the root of the heap.

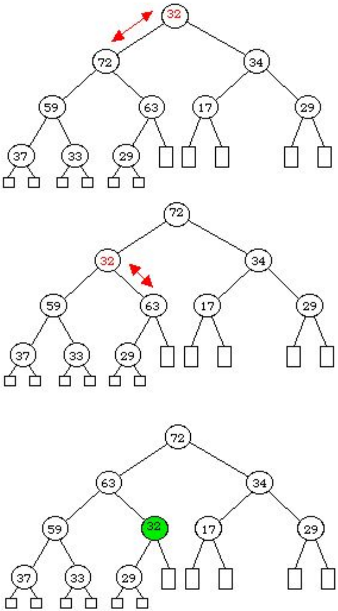
Heaps: Removing the maximum

1. Replace the root of the heap with the last element (the rightmost element in the last level)
2. Reestablish the invariant (heap order) sinking the root: The function `sink` exchanges a given node with its largest priority child, if its priority is smaller than the priority of its child, and repeats the same until the heap order is reestablished.

Heaps: Removing the maximum

1. Replace the root of the heap with the last element (the rightmost element in the last level)
2. Reestablish the invariant (heap order) sinking the root: The function `sink` exchanges a given node with its largest priority child, if its priority is smaller than the priority of its child, and repeats the same until the heap order is reestablished.

Heaps: Removing the maximum



Heaps: Adding a new element

1. Add the new element as rightmost node of the last level of the heap (or as the first element of a new deeper level)
2. Reestablish the heap order sifting up (a.k.a. *floating*) the new added element:
   The function `siftup` compares the given node to its father, and they are exchanged if its priority is larger than that of its father; the process is repeated until the heap order is reestablished.

Heaps: Adding a new element

1. Add the new element as rightmost node of the last level of the heap (or as the first element of a new deeper level)
2. Reestablish the heap order sifting up (a.k.a. *floating*) the new added element:
   The function `siftup` compares the given node to its father, and they are exchanged if its priority is larger than that of its father; the process is repeated until the heap order is reestablished.

The Cost of Heaps

Since the height of a heap is $\Theta(\log n)$, the cost of removing the maximum and the cost of insertions is $\mathcal{O}(\log n)$.

We can implement heaps with dynamically allocated nodes, and three pointers per node (left, right, father) ... But it is much easier and efficient to implement heaps with vectors!

Since the heap is a quasi-complete binary tree this allows us to avoid wasting memory: the n elements are stored in the first n components of the vector, which implicitly represent the tree.

Implementing Heaps

To make the rules easier we will use a vector A of size $n + 1$ and discard $A[0]$. Resizing can be used to allow unlimited growth.

1. $A[1]$ contains the root
2. If $2i \leq n$ then $A[2i]$ contains the left child of $A[i]$ and if $2i + 1 \leq n$ then $A[2i + 1]$ contains the right subtree of $A[i]$
3. If $i \geq 2$ then $A[i/2]$ contains the father of $A[i]$

Implementing Heaps

To make the rules easier we will use a vector A of size $n+1$ and discard $A[0]$. Resizing can be used to allow unlimited growth.

1. $A[1]$ contains the root
2. If $2i \leq n$ then $A[2i]$ contains the left child of $A[i]$ and if $2i+1 \leq n$ then $A[2i+1]$ contains the right subtree of $A[i]$
3. If $i \geq 2$ then $A[i/2]$ contains the father of $A[i]$

Implementing Heaps

To make the rules easier we will use a vector A of size $n + 1$ and discard $A[0]$. Resizing can be used to allow unlimited growth.

1. $A[1]$ contains the root
2. If $2i \leq n$ then $A[2i]$ contains the left child of $A[i]$ and if $2i + 1 \leq n$ then $A[2i + 1]$ contains the right subtree of $A[i]$
3. If $i \geq 2$ then $A[i/2]$ contains the father of $A[i]$

Implementing Heaps

```
template <typename Elem, typename Prio>
class PriorityQueue {
public:
 ...
private:
   // Component of index 0 is not used
   vector<pair<Elem, Prio> > h;
   int nelems;

   void siftup(int j) throw();
   void sink(int j) throw();
};
```

Implementing Heaps

```
template <typename Elem, typename Prio>
bool PriorityQueue<Elem,Prio>::empty() const {

   return nelems == 0;
}

template <typename Elem, typename Prio>
Elem PriorityQueue<Elem,Prio>::min() const {

   if (nelems == 0) throw EmptyPriorityQueue;
   return h[1].first;
}

template <typename Elem, typename Prio>
Prio PriorityQueue<Elem,Prio>::min_prio() const {

   if (nelems == 0) throw EmptyPriorityQueue;
   return h[1].second;
}
```

Implementing Heaps

```
template <typename Elem, typename Prio>
void PriorityQueue<Elem,Prio>::insert(cons Elem& x,
                                      cons Prio& p) {
   ++nelems;
   h.push_back(make_pair(x, p));
   siftup(nelems);
}

template <typename Elem, typename Prio>
void PriorityQueue<Elem,Prio>::remove_min() const {

   if (nelems == 0) throw EmptyPriorityQueue;
   swap(h[1], h[nelems]);
   --nelems;
   h.pop_back();
   sink(1);
}
```

Implementing Heaps

```
// Cost: O(log(n/j))
template <typename Elem, typename Prio>
void PriorityQueue<Elem,Prio>::sink(int j) {

   // if j has no left child we are at the last level
   if (2 * j > nelems) return;

   int minchild = 2 * j;
   if (minchild < nelems and
       h[minchild].second > h[minchild + 1].second)
      ++minchild;

   // minchild is the index of the child with minimum priority
   if (h[j].second > h[minchild].second) {
      swap(h[j], h[minchild]);
      sink(minchild);
   }
}
```

Implementing Heaps

```
// Cost: O(log j)
template <typename Elem, typename Prio>
void PriorityQueue<Elem,Prio>::siftup(int j) {

   // if j is the root we are done
   if (j == 1) return;

   int father = j / 2;
   if (h[j].second < h[father].second) {
      swap(h[j], h[father]);
      siftup(father);
   }
}
```

Part IV

Priority Queues

Heapsort

Heapsort (Williams, 1964) sorts an array of n elements building a heap with the n elements and extracting them, one by one, from the heap (cif. our example of the atoic weights and chemical symbols).

The originally given array is used to build the heap; heapsort works in-place and only some constant auxiliary memory space is needed.
Since insertions and deletions in heaps have cost $\mathcal{O}(\log n)$ the cost of the algorithm is $\mathcal{O}(n \log n)$.

Heapsort

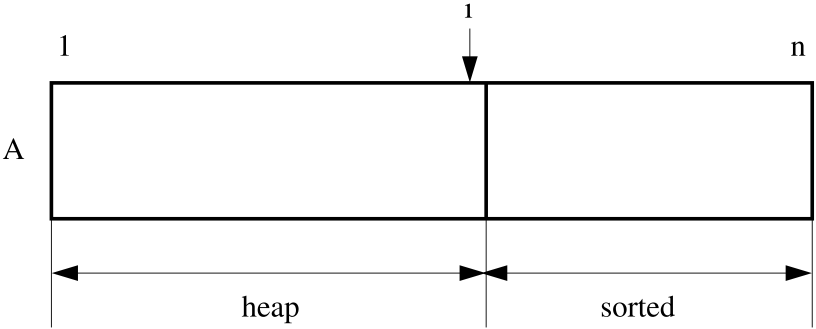
```
template <typename Elem>
void sink(Elem v[], int sz, int pos);

// Sort v[1..n] in ascending order
// (v[0] is unused)
template <typename Elem>
void heapsort(Elem v[], int n) {

   make_heap(v, n);
   for (int i = n; i > 0; --i) {
       // extract largest element from the heap
       swap(v[1], v[i]);

       // sink the root to reestablish max-heap order
       // in the subarray v[1..i-1]
       sink(v, i-1, 1);
   }
}
```

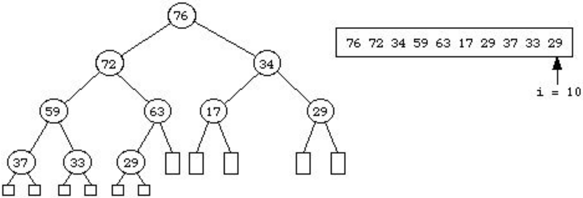
Heapsort

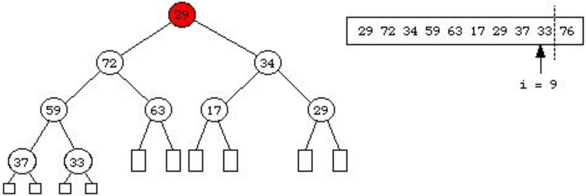


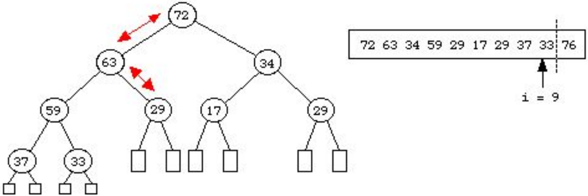
$$A[1] \le A[i+1] \le A[i+2] \le \ldots \le A[n]$$

$$A[1] = \max_{1 \le k \le i} A[k]$$

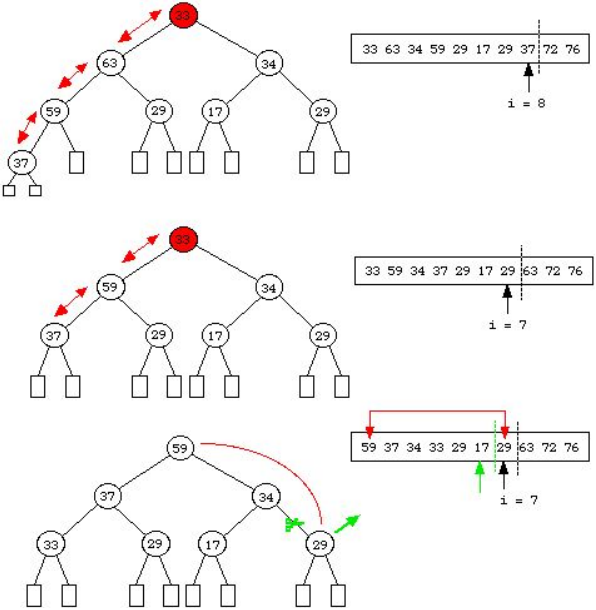
Heapsort


76
72
34
59
63
17
29
37
33
29
76 72 34 59 63 17 29 37 33 29
i = 10


29
72
34
59
63
17
29
37
33
29 72 34 59 63 17 29 37 33 76
i = 9


72
63
34
59
29
17
29
37
33
72 63 34 59 29 17 29 37 33 76
i = 9

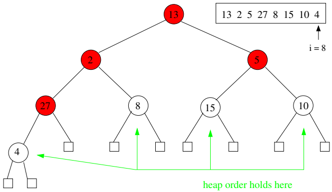
Heapsort

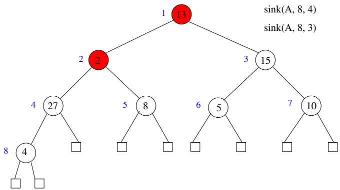

33 63 34 59 29 17 29 37 72 76
i = 8
33 59 34 37 29 17 29 63 72 76
i = 7
59 37 34 33 29 17 29 63 72 76
i = 7

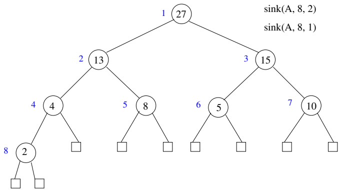
Heapify

```
// Establish (max) heap order in the
// array v[1..n] of Elem's; Elem == priorities
// this is a.k.a. as heapify
template <typename Elem>
void make_heap(Elem v[], int n) {
   for (int i = n/2; i > 0; --i)
       sink(v, n, i);
}
```

Heapify


13 2 5 27 8 15 10 4
i = 8
heap order holds here


sink(A, 8, 4)
sink(A, 8, 3)


sink(A, 8, 2)
sink(A, 8, 1)

The Cost of Heapsort

Let $H(n)$ be the worst-case cost of `heapsort` and $B(n)$ the cost `make_heap`. Since the worst-case cost of $\texttt{sink}(v, i-1, 1)$ is $\mathcal{O}(\log i)$ we have

$$
\begin{aligned}
H(n) &= B(n) + \sum_{i=1}^{i=n} \mathcal{O}(\log i) \\
&= B(n) + \mathcal{O}\left(\sum_{1 \le i \le n} \log_2 i \right) \\
&= B(n) + \mathcal{O}(\log(n!)) = B(n) + \mathcal{O}(n \log n)
\end{aligned}
$$

A rough analysis of $B(n)$ shows that $B(n) = \mathcal{O}(n \log n)$ since it makes $\Theta(n)$ calls to `sink`, each one with cost $\mathcal{O}(\log n)$. Hence, $H(n) = \mathcal{O}(n \log n)$; actually, $H(n) = \Theta(n \log n)$ in any case if all elements are different.

The Cost of Heapify

A refined analysis of $B(n)$:

$$
\begin{aligned}
B(n) &= \sum_{1 \le i \le \lfloor n/2 \rfloor} \mathcal{O}(\log(n/i)) \\
&= \mathcal{O}\left(\log \frac{n^{n/2}}{(n/2)!}\right) \\
&= \mathcal{O}\left(\log (2e)^{n/2}\right) = \mathcal{O}(n)
\end{aligned}
$$

Since $B(n) = \Omega(n)$, we conclude $B(n) = \Theta(n)$.

The Cost of Heapify

Alternative proof: Let $h = \lceil \log_2(n+1) \rceil$ the height of the heap. Level $h-1-k$ contains at most

$$2^{h-1-k} < \frac{n+1}{2^k}$$

elements; in the worst-case each one will sink down to level $h-1$ with cost $\mathcal{O}(k)$

The Cost of Heapify

$$
\begin{aligned}
B(n) &= \sum_{0 \le k \le h-1} \mathcal{O}(k) \frac{n+1}{2^k} \\
&= \mathcal{O}\left(n \sum_{0 \le k \le h-1} \frac{k}{2^k} \right) \\
&= \mathcal{O}\left(n \sum_{k \ge 0} \frac{k}{2^k} \right) = \mathcal{O}(n),
\end{aligned}
$$

since

$$\sum_{k \ge 0} \frac{k}{2^k} = 2.$$

In general, if $0 < |r| < 1$,

$$\sum_{k \ge 0} k \cdot r^k = \frac{r}{(1-r)^2}.$$

The Cost of Heapify

Despite $H(n) = \Theta(n \log n)$, the refined analysis of $B(n)$ is important: using a *min-heap* we can get the smallest k elements in an array in ascending order with cost:

$$\begin{aligned} S(n,k) &= B(n) + k \cdot \mathcal{O}(\log n) \\ &= \mathcal{O}(n + k \log n). \end{aligned}$$

If $k = \mathcal{O}(n / \log n)$ then $S(n,k) = \mathcal{O}(n)$.