Priority Queues

Arthur Madore-Boisvert, Will Zahary Henderson

March 9, 2022

This is an augmented transcript of two lectures given by Luc Devroye on the 9th of March 2022 for the Honours Algorithms and Data Structures class (COMP 252, McGill University). The subject was priority queues.

Introduction and Definitions

A **priority queue** is an abstract data type. It consists of a set of **elements** $(x_1, ..., x_n)$ that can be ordered by their respective **keys**, along with the following defining operations:

1. INSERT(k, PQ) adds a new key k to priority queue PQ
2. DELETEMIN(PQ) deletes the minimum key

Additional but less important operations can also be implemented:

DECREASEKEY(x, k, PQ) changes the key of item x to a smaller value k

DELETE(x, PQ) deletes element x

Do note that SEARCH is *not* one of the operations here, as it is not part of the purpose of priority queues.

Implementations

	Complexity in Θ notation	
	INSERT	DELETEMIN
Sorted list	n	1
Unsorted list	1	n
Balanced search tree	$\log n$	$\log n$
Binary heap	$\log n$	$\log n$
Tournament tree	$\log n$	$\log n$
Fibonacci heap	1	$\log n$

It is worth noting that Fibonacci heap insertion is constant time in an *amortized* sense (see the lecture on amortized analysis), while it is not guaranteed to always perform in constant time.

Uses

Example 1. In *operating systems*, priority queues are often used for job scheduling. The priority queue holds jobs, and the priorities are determined by the system. Jobs are swapped in and out to get access to the CPU for a short time quantum.

Example 2. Priority queues can be used to construct an *optimal tree*, and more specifically are used for *Huffman code*, a type of optimal prefix code utilizing a greedy algorithm that is very efficient in compressing data[1].

Example 3. In *discrete event simulation*, a priority queue can be used as a future event set to model complex real-world systems. One relevant situation could be modelling a bank: customers entering the bank and lining up to visit the teller, and customers leaving.

Example 4. In a generic manner, one can sort the keys by first inserting them in an empty priority queue and then performing DELETEMIN until all keys are deleted.

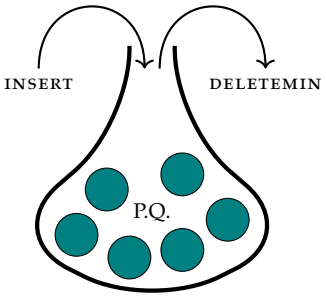

Figure 1: An illustration of how an operating system inserts and removes processes (teal circles) from a priority queue; alternatively, this priority queue can represent a future event set, where each key signifies time.

[1] Cormen et al. [1989]

Binary Heap

Definition 5. A **binary heap** is a binary tree with the **heap property**.

Definition 6. The **heap property** states that $\forall x : \text{key}[x] < \text{key}[y]$ where y represents any descendant of x in the heap.

Binary heaps are often implemented as an array where the index corresponds to its element number. However, they can be well-visualized as a complete binary tree (as in fig. 2).

For clarity, in the remaining discussion on binary heaps, when references are made to "height" and "children," we visualize the heap in its complete binary tree form, but also provide justification for index calculations that relate the tree format to the array format.

Remark 7. Suppose the key of some node x in the heap is stored at some index i in the array. Then the left and right children, assuming they exist, will be stored at indices $2i$ and $2i + 1$, respectively. Take note that in order to preserve this relationship, the root is stored at index 1 in the array.

Remark 8. The height of a binary heap is $\lfloor \log_2 n \rfloor$ (same as for a complete binary tree).

We also propose the following conventions: $H[i]$ refers to the key in position i in heap H, and heapsize$[H]$ refers to the size of the heap.

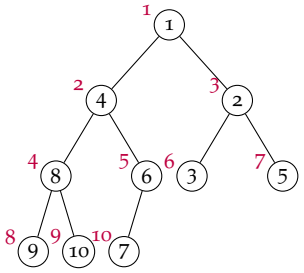

Figure 2: A binary heap. Keys are written in black within the node, and elements are written in red to the left of their corresponding node.

0	1	2	3	4	5	6	7	8	9	10
	1	4	2	8	6	3	5	9	10	7

Figure 3: The corresponding array implementation of fig. 2.

Atomic Operations

Binary heaps have two main familiar atomic operations. Assume heap H is give that item x has its key altered so it is out of position (e.g., the node has a key value smaller than the key value of its parent). To fix our heap and ensure the heap property is preserved, we could either move x up in the tree, or move it down in the tree. For this purpose, we define the following atomic operations:

1. SIFTUP[2] moves x up the heap
2. HEAPIFY[3] moves x down the heap

These algorithms are programmed as follows.

[2] In COMP 250, we called this "upheap"

[3] In COMP 250, we called this "downheap"

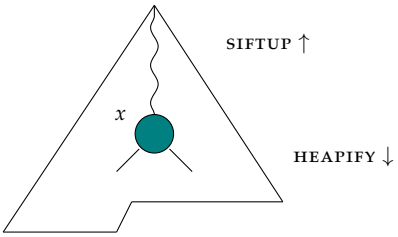

Figure 4: A heap where x just had its key altered.

SIFTUP(i, H): /sifting up the element at index i in the heap array/

```
1    while i ≠ 1
2        if key[parent[i]] ≤ key[i] then halt
3        else
4            swap keys of i and parent[i]
5            i ← parent[i]
```

HEAPIFY(i, H):

```
1    while i ≤ heapsize[H]
2        j ← argmin(key[i], key[2i], key[2i + 1])
         / if 2i and 2i + 1 are heap items /
3        if j = i then halt
4        else
5            swap keys of j and i
6            i ← j
```

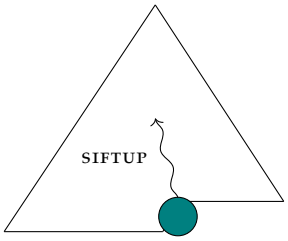

Figure 5: Inserting a new element into a heap.

Other Operations

Using the atomic operations, we can define INSERT and DELETEMIN for binary heaps quite simply as follows.

INSERT(k, H): / *k is the value of the key we want to insert* /

```
1    heapsize[H] += 1
2    H[heapsize[H]] ← k
3    SIFTUP(heapsize[H])
```

The time complexity of INSERT is $\mathcal{O}(\log_2 n)$ as it performs at most $\lfloor \log_2 n \rfloor$ comparisons, where n is the new size of the heap.

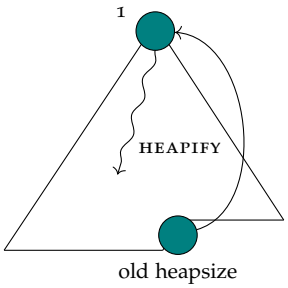

Figure 6: Deleting the minimum element from a heap.

DELETEMIN(H):
1 $x \leftarrow H[1]$
2 $H[1] \leftarrow H[\text{heapsize}[H]]$
3 heapsize$[H]$ −= 1
4 **HEAPIFY**(1)
5 **return** x

The time complexity of DELETEMIN is also $\mathcal{O}(\log_2 n)$ as it performs at most $2\lfloor \log_2 n \rfloor$ comparisons, where n is the old size of the heap. This due to the fact that a binary heap is a balanced binary tree which has the property to have a maximal height of $\lfloor \log_2 n \rfloor$.

Building a Heap

Definition 9. The operation BUILDHEAP takes an array H of n unsorted keys and turns H into a valid heap in which the key values respect **Definition 6**.

BUILDHEAP(H):
1 **for** $i = \lfloor \text{heapsize}[H]/2 \rfloor$ down to 1
2 **do HEAPIFY**(i)

Theorem 10. *The* BUILDHEAP *operation performs a total of fewer than* $4n$ *comparisons.*

Proof. The number of comparisons performed by BUILDHEAP is bounded by (see fig. 7):

$$2n \cdot \sum_{i=1}^{\infty} \frac{i}{2^i}. \tag{1}$$

Note the following are also true for power series of this form in general.

$$\sum_{i\geq 0}^{\infty} x^i = \frac{1}{1-x}, \quad \sum_{i\geq 0}^{\infty} i \cdot x^{i-1} = \frac{1}{(1-x)^2}. \tag{2}$$

With this in mind, we have that

$$\sum_{i=0}^{\infty} \frac{i}{2^i} = \frac{\frac{1}{2}}{(\frac{1}{2})^2} = 2. \tag{3}$$

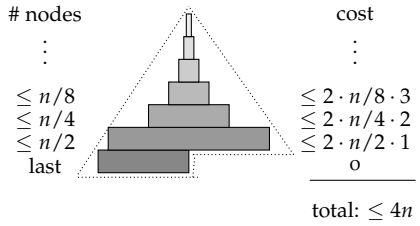

Figure 7: A visualization of BUILDHEAP, displaying the number of nodes at each level relative to the total number of nodes n, as well as the total cost to build each level.

Putting it all together, we have

$$2n \cdot \sum_{i=1}^{\infty} \frac{i}{2^i} = 2n \cdot 2 = 4n. \tag{4}$$

Thus, BUILDHEAP performs at most $4n$ comparisons. □

Exercise 11. We leave it as an exercise to show that the second to last level in the heap has at most $\frac{n}{2}$ items. This implies that i levels above the last level, we have at most $\frac{n}{2^i}$ items. This fact was used in eq. 1.

Heapsort

Heapsort takes an unsorted array H with n items and sorts it using a heap. The algorithm is as follows.

HEAPSORT(H):
1 **BUILDHEAP**(H) / *make H into a binary heap* /
2 heapsize$[H] \leftarrow n$
3 **for** $x = n$ down to 2 **do**
4 swap key$[x]$ **and** key$[1]$
5 heapsize$[H] \leftarrow$ −= 1
6 **HEAPIFY**(1)

The number of comparisons HEAPSORT performs is bounded by above by

$$\underbrace{4n}_{\text{BUILDHEAP cost}} + \underbrace{2\sum_{i=1}^{n-1} \lfloor \log_2 i \rfloor}_{\textbf{for loop}} \leq 4n + 2n \log_2 n.$$

Thus, heapsort has $\Theta(n \log_2 n)$ complexity. However, it is worth noting that the number of comparisons here is suboptimal, and is much further away from the lower bound than some other sorting algorithms like merge sort, which comes within $\mathcal{O}(n)$ of the lower bound, which is roughly $n \cdot \log_2(n)$.

Pointer-Based Binary Heaps

Although binary heaps are typically implemented as arrays, they can equivalently be implemented as cells with parent and children pointers.

The trick to find the path from the root to an element at index x in a pointer-based binary heap is by considering the binary representation of x, discarding the leftmost 1 digit, and then walking through the digits from left to right, considering 0 as meaning "left child" and 1 as meaning "right child." This is illustrated in fig. 9.

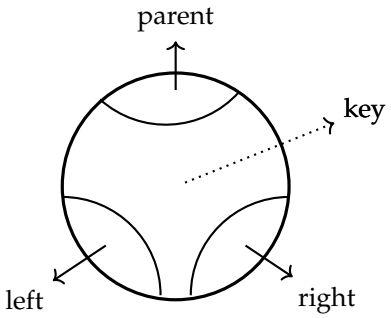

Figure 8: The cell of a pointer-based binary heap.

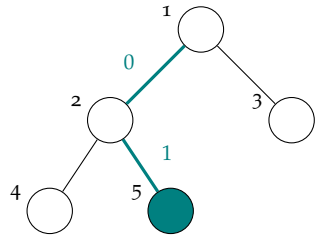

Figure 9: The path from the root node to element 5. $5 = (101)_2$, so the path is derived from 01, which instructs us to go to the left child, and then to the right child.

k-ary Heap

Definition 12. A ***k*-ary heap** is a complete k-ary tree with the heap property.

k-ary heaps are functionally very similar to binary heaps, but operations take a different amount of time depending on the value of k. Though programmed almost identically, the time complexities are as follows.

DELETEMIN takes roughly $k \log_k n$ comparisons

INSERT uses about $\log_k n$ comparisons

In a k-ary heap, we create the notion of *oldest child*, i.e. the leftmost child of a node, and *youngest child*, i.e. the rightmost child of a node. The following define the relationships between a node at index x and its children and parent nodes:

$$\text{parent}[x] = \left\lfloor \frac{x+k-2}{k} \right\rfloor$$

$$\text{oldestchild}[x] = kx - (k-2)$$

$$\text{youngestchild}[x] = kx + 1$$

Remark 13. If the number of items inserted is equal to the number of items deleted, then one can optimize k by minimizing the sum of the root of one INSERT and one DELETEMIN, i.e., $\frac{1+k}{\log k} \cdot \log n$. This is smallest for $k = 4$.

Remark 14. The height of a k-ary heap is approximately $\log_k n$.

Exercise 15. Derive the exact value of the height of a k-ary heap as a function of k and n.

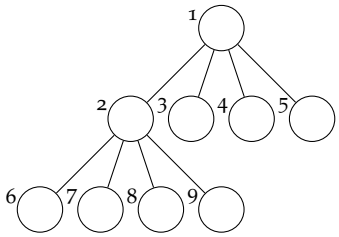

Figure 10: A 4-ary heap.

Tournament Tree

Definition 16. A **tournament tree** is a complete binary tree consisting leaves representing data and internal nodes representing "matches." Each internal node contains a pointer $\sigma[i]$ to the "winner" (smaller element) of each binary comparison, which also turns out to be the smallest descendant of the node.

All internal nodes in a tournament tree have two children.

Remark 17. The total number of nodes in a tournament tree is $2n - 1$, as there are n leaves and $n - 1$ internal nodes.

Operations

The tournament tree has one atomic operation called UPDATE that mirrors SIFTUP from binary heaps.

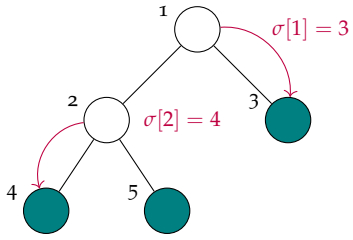

Figure 11: A tournament tree where the element at key 4 is smaller than the element at key 5, and the element at key 3 is smaller than the element at key 4. The teal nodes represent the data as they are the leaves, and the white internal nodes are pointers to their smallest descendants.

Definition 18. UPDATE updates the tree to adjust to a new key k at a leaf node i.

UPDATE(i, k):
1 key$[i] \leftarrow k$
2 $\sigma[i] \leftarrow i$
3 **while** $i \neq 1$ **do**:
4 $\quad j \leftarrow$ sibling(i) / *sibling*(i) *is* $i+1$ *or* $i-1$ /
5 $\quad$ **if** key$[\sigma[j]] <$ key$[\sigma[i]]$
6 $\quad\quad$ **then** σ[parent$[i]] \leftarrow \sigma[j]$
7 $\quad\quad$ **else** σ[parent$[i]] \leftarrow \sigma[i]$
8 $\quad i \leftarrow$ parent$[i]$

Using this operation, we can recreate INSERT, DELETEMIN, and BUILDTOURNAMENT in the context of tournament trees.

INSERT(k, n): / *insert new key* k *in a tree with* n *leaves* /
1 key$[2n] \leftarrow$ key$[n]$
2 $\sigma[2n] \leftarrow 2n$
3 $\sigma[2n+1] \leftarrow 2n+1$
4 **UPDATE**$(2n+1, k)$

DELETEMIN(k, n) [4] :
1 **return** key$[\sigma[1]]$
2 key$[\sigma[1]] \leftarrow \infty$
3 **UPDATE**$(\sigma[1])$

[4] This is a *lazy delete*, which means it does not really delete the element, but instead just marks it as being deleted.

Exercise 19. Write an algorithm DELETEMIN for tournament trees that is *not* a lazy delete but still runs in $\mathcal{O}(\log n)$ time.

BUILDTOURNAMENT(n): / n *known* /
1 Create an array of size $2n-1$ **for** $\sigma[\cdot]$ **and** key$[\cdot]$
2 **for** $i = n$ to $2n-1$ **do**
3 $\quad$ fill **in** key$[i]$
4 $\quad \sigma[i] \leftarrow i$
5 **for** $i = n-1$ down to 1 **do** / *play the tournament* /
6 $\quad$ **if** key$[\sigma[2i]] <$ key$[\sigma[2i+1]]$
7 $\quad\quad$ **then** $\sigma[i] \leftarrow \sigma[2i]$
8 $\quad\quad$ **else** $\sigma[i] \leftarrow \sigma[2i+1]$

Building the tournament takes $n-1$ comparisons, as there are $n-1$ matches to be played between n competitors.

Tournament Tree Sort

One can sort an array of items using a tournament tree, creating a system similar to heapsort. In fact, using the operations above, this is

a very simple task; all one needs to do to sort an array of n items is build the tournament tree using BUILDTOURNAMENT and then call DELETEMIN n times.

An algorithm that performs this is seen below.

```
TTSORT(N): / N is an unsorted array of n items /
1   Create an empty array A of length n
2   BUILDTOURNAMENT(N)
3   for i = 1 up to n do
4       A[i] ← DELETEMIN(N)
```

This takes fewer than $n + n \log_2 n$ operations as the heap is built in $n - 1$ comparisons and there are n calls to DELETEMIN which each take $\leq \log_2(2n - 1)$ comparisons. It is worth noting that although heapsort and tournament tree sort are both $\Theta(n \log_2 n)$ sorting methods, tournament tree sort is much closer to the lower bound for sorting than heapsort.

References

T.H Cormen, C.E. Leiserson, R.L.Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 1989. ISBN 9780262033848.