Priority Queues

Contents

Introduction

A priority queue is based on a normal queue, but each element has a priority associated with it. Depending on the type of priority queue, the highest or lowest priority element is removed first.

Priority queues as an ADT support the following operations (assuming a min-priority queue):

- **MAKENULL():** Creates a new, empty priority queue.
- **INSERT(k, pq):** Inserts `k` into the priority queue `pq`.
- **REMOVEMIN(pq):** Removes the minimum priority element from the priority queue.
- **UPDATEKEY(x, k, pq):** Updates the key of `x` to `k`.
- **DELETE(x, pq):** Deletes the node `x` from the priority queue.

Implementations:

A priority queue can be implemented in a variety of ways, which are shown below:

Data Structure	Insert	Remove Min
Heap	$O(logn)$	$O(logn)$
Sorted Linked List	$O(n)$	$O(1)$
Unsorted Linked List	$O(1)$	$O(n)$
Balanced Binary Search Tree	$O(logn)$	$O(logn)$
Tournament Tree	$O(logn)$	$O(logn)$
BEAP	$O(\sqrt{n})$	$O(\sqrt{n})$
Fibonacci Heap (*amortized time)	$O(1)$*	$O(logn)$*

Typically, a heap is used to implement a priority queue.

Applications:

Priority queues are useful for a variety of tasks, including:

- **Job Scheduling:** Operating systems can use priority queues to execute jobs in order of highest priority.
- **Huffman Tree Construction**
- **Discrete Event Simulation**
- **Heapsort:** Elements can be placed into a priority queue and then removed one by one with REMOVEMIN to get their sorted order.

Binary Heaps

A binary heap is a complete binary tree where every node follows the heap property. Since the tree is complete, implementation can be either pointer or array-based. We will assume an array-based implementation below.

The main operations for heaps are:

- `siftup(x)`: Moves the element at position `x` up the heap until it satisfies the heap property.
- `heapify(x)` or `siftdown(x)`: Moves the element at position `x` down the heap until it satisfies the heap property.
- `insert(k, H)`: Inserts `k` into the heap `H`. This is done by inserting the new element into the next available complete tree position and calling `siftup` until it is in a proper heap position. Requires $\leq \lfloor log_2 n \rfloor$ comparisons.
- `deleteMin(H)`: Removes the minimum element from `H`. Requires $\leq 2\lfloor log_2 n \rfloor$, where n is the size of the heap before the min is removed. The number of comparisons is doubled since the smaller child must be found at each level, which requires 2 comparisons.
- `buildHeap(H)`: Given an array `H` of keys, turn the array into a heap.

The naive method of building a heap would require $O(nlogn)$ comparisons, where each node that is inserted would make use of the `siftup` method. This would entail starting at the beginning of the array and moving towards the end.

However, it is possible to do better by only using the `siftdown` operation. To do this, we start at the end of the array instead and move backwards through it. The work required for this approach is:

$$(0 \times \tfrac{n}{2}) + (1 \times \tfrac{n}{4}) + (2 \times \tfrac{n}{8}) \ldots + (h \times 1)$$

Where the first term in each multiplication is the distance that each node needs to move and the second term is the number of nodes that need to move that far. In contrast, the work required with the `siftup` approach is:

$$(h \times \tfrac{n}{2}) + ((h-1) \times \tfrac{n}{4}) + ((h-2) \times \tfrac{n}{8}) \ldots + (0 \times 1)$$

Which is clearly a larger sum. It turns out that the first sum for `siftdown` actually gives us a **linear** time complexity, which can be shown as follows:

$(0 \times \frac{n}{2}) + (1 \times \frac{n}{4}) \ldots + (h \times 1) = \sum_{i=0}^{h} \frac{ni}{2^{i+1}} = \frac{n}{2} \sum_{i=0}^{h} \frac{i}{2^i}$

Sum of an infinite series: $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$

Take the derivative: $\sum_{i=1}^{\infty} ix^{i-1} = \frac{1}{(1-x)^2}$

$\sum_{i=1}^{\infty} ix^i = \frac{x}{(1-x)^2}$

$\sum_{i=0}^{\infty} \frac{i}{2^i} = \sum_{i=0}^{\infty} i(\frac{1}{2})^i = \frac{(\frac{1}{2})}{(1-\frac{1}{2})^2} = 2$

$\frac{n}{2} \sum_{i=0}^{\infty} \frac{i}{2^i} = \frac{n}{2}(2) = n$

This shows that it is possible to build a heap in $O(n)$ time.

HeapSort

When given an array of unsorted elements, it is preferred to sort them in place. This can be done with HeapSort. The process for the algorithm is given below:

- Call `buildHeap` to make the array a heap in $O(n)$ time.
- Repeatedly "remove" the minimum element by swapping it with the rightmost unsorted element, and then call `siftdown` on the new root element to fix up the heap.
- Repeat for all elements.

When doing this, the sorted array will grow from right to left. In a tree, this can be visualized as the leaves being sorted first and then moving up towards the root.

The pseudocode is given below:

```
func heapSort(H) {
    // Convert array into a heap
    H = buildHeap(H)

    for (i = H.size downto 2) {
        // Swap the root with the rightmost unsorted element
        H.swap(1, i)

        // Fix up the heap from the first to i-th element
        H.heapify(1, i)
    }
}
```

The time complexity is $\leq \Theta(n) + 2nlog_2 n = \Theta(nlogn)$. A min-heap will sort the array in descending order from largest to smallest.

It is possible to optimize HeapSort by reducing the number of comparisons. This can be done as follows:

- Build the heap in the same way as before.
- When fixing the new root, it follows some path to get to its final position. This path will naturally be the path of smaller children, and it will also be sorted.
    - To do this, find the path of smallest children from the root.
    - Search up from the bottom leaf of this path until the position is found for the new root.

This reduces the number of comparisons to $\leq 2\lfloor log_2 n \rfloor$. But we can do even better! Since the path of smallest children is sorted, it is possible to use binary search to find the position to insert at. This reduces the time to $\leq \lfloor log_2 (log_2 n) \rfloor$.

With this optimization, `removeMin` takes time $\lfloor log_2 n \rfloor + log_2 (log_2 n)$. Then, the new overall time for HeapSort is $\leq \Theta(n) + n(log_2 n + log_2 (log_2 n))$.

k-ary Heaps

So far we have looked at binary heaps exclusively. But what if we implement a heap where each node has k children? These are known as k-ary heaps.

A complete tree with k children at each node has a height of $\lfloor log_k n \rfloor$. In an array, the positions of a node's children are given by:

$parent[x] = \frac{x+k-2}{k}$

$left[x] = kx - (k-2)$

$right[x] = kx + 1$

Below are the time complexities of heap algorithms in a k-ary heap:

- **RemoveMin:** $O(klog_k n)$
- **Optimized RemoveMin:** $O((k-1)log_k n)$, since it takes $k-1$ comparisons to find the smallest child
- **Insert:** $\lfloor log_k n \rfloor$

Given these time complexities, it is natural to wonder what the best value for k is to minimize the time complexity of the above operations. Essentially, we would like to minimize the value of $O(insert) + O(deleteMin)$, which is given by:

$log_k n + (k-1)log_k n = klog_k n = k\frac{log_2 n}{log_2 k}$

Since $log_2 n$ is independent of k, the value to minimize is $\frac{k}{log_2 k}$. By setting the derivative of this to zero, the optimal value ends up being somewhere between 2 and 3, so binary heaps are sufficiently optimal in terms of time complexity.