CS 261: Data Structures

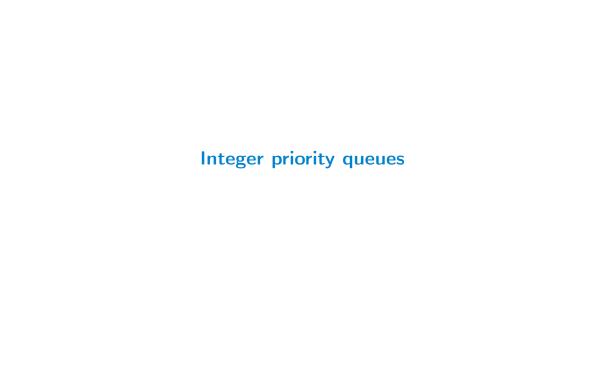# Week 5: Priority queues

Lecture 5b: Integer priority queues

David Eppstein
University of California, Irvine

Spring Quarter, 2025

Integer priority queues

Main idea

k-ary heaps, Fibonacci heaps only use comparison of priorities

If priorities are small integers, we can use bitwise binary operations, array indexing, etc

Sometimes this can be faster

Integer priority queues versus sorting

Many non-comparison-based sorting algorithms are known

E.g. radix sort: Sort n integers in range $[0, X]$ in time $O\left(n\dfrac{\log X}{\log n}\right)$

If we can do priority queue operations in time T, we can use them to sort in time $O(nT)$ ("heap sort")

Reverse is true!

If we can sort in time $nT(n)$, we can do priority queue operations in time $T(n) + T(\log n) + T(\log \log n) + \cdots$

[Thorup 2007]

Theoretically fast but completely impractical

Current best known (expected) time bound for integer sorting, when priorities are small enough to be represented by machine integers: $O(n\sqrt{\log \log n})$

[Han and Thorup 2002]

Corollary: Can do integer priority queue operations in $O(\sqrt{\log \log n})$ expected time per operation

We will look at simpler data structures whose time depends on the range of values, not just on n

Open: Can we do better or is there a non-constant lower bound?

Bucket queues

Bucket queue

Priority queue version of bucket sort

Main idea: arrange items into buckets by their priorities

Scan buckets in priority order to find the first nonempty one

Bucket queue, more detail

When all priorities are small non-negative integers:

▶ Make an array of buckets (lists, sets, or arrays), indexed by priority

▶ Each bucket: collection of elements with that priority

▶ Add element; put it into the bucket for its priority

▶ Change priority: move it from one bucket to another

▶ Find-minimum: scan array for first non-empty bucket

Time: O(current minimum priority) for find-min, $O(1)$ otherwise

Monotonic priorities and Dial's algorithm

In Dijkstra's algorithm, priorities never decrease

$\Rightarrow$ Can start scan for non-empty bucket at the last one we found, instead of starting at the beginning of the array

Total time for priority queue operations: $O(\#\text{ operations} + \text{max priority})$

Total time for shortest paths in graphs with integer edge lengths: linear in size of graph $+ O(\text{max path length})$

[Dial 1969]

Flat trees
(also known as Van Emde Boas trees)

Main idea

Think of priorities as b-bit binary numbers, for some b

Partition each priority x into two $b/2$-bit numbers
(the two digits in its base-$2^{b/2}$ representation)

▶ x_{low} : `x & ((1 << b/2) – 1)`

▶ x_{high} : `x >> (b/2)`

Build recursive priority queues for the low parts and the high parts

Each operation recurses into low or high but not both
$\Rightarrow$ time $O(\log b) = O\big(\log\log(\text{max priority})\big)$

[van Emde Boas 1975]

Simplifying assumption: elements = priorities

If we have elements that are not the same thing as the priorities of the elements, then:

▶ Make a dictionary mapping each priority to a collection of elements with that priority (like bucket queue)

▶ Keep a priority queue of the set of priorities whose collections are non-empty

▶ To find an element of minimum priority: use the priority queue to find the priority, and the dictionary to find the element

The dictionary part is $O(1)$ expected time per operation (using a hash table) so we only need to worry about priority queues whose elements are integers and whose priorities are their values

Flat tree structure

The flat tree for a set S of integers stores:
- ▶ The minimum element in S
- ▶ Recursive flat tree H of x_{high} for $x \in S \setminus \{min\}$
- ▶ Dictionary D mapping each x_{high} to a recursive flat tree of values x_{low} for the subset of elements $x \in S \setminus \{min\}$ that have that value of x_{high}

To find the minimum priority:
Look at the value stored as minimum at root-level structure

To make one-element flat tree:
Set minimum to element, H to None, D to empty dictionary

Example

Suppose we are storing a flat tree for the nine 8-bit keys 00000001, 00000011, 00001010, 00001111, 10100101, 10101010, 11011110, 11011111, and 11101010. Then:

The minimum element is 00000001

The values x_{high} for the remaining keys are 0000, 1010, 1101, and 1110. The recursive flat tree H stores these four 4-bit keys.

The dictionary D maps each of these four values x_{high} to a recursive flat tree $D[x_{\text{high}}]$:

▶ $D[0000]$ stores the low 4-bit keys whose high 4 bits are 0000, not including the minimum element 00000001. These 4-bit keys are: 0011, 1010, and 1111.

▶ $D[1010]$ stores the two low 4-bit keys 0101 and 1010

▶ $D[1101]$ stores the two low 4-bit keys 1110 and 1111

▶ $D[1110]$ stores as its single element the low 4-bit key 1010.

To insert an element x:

1. If $x <$ minimum, swap x with minimum
2. If x_{high} is in D:

   Recursively insert x_{low} into $D[x_{\text{high}}]$
3. Else:

   Make one-element flat tree containing x_{low}

   Add the new tree to D as $D[x_{\text{high}}]$

   Recursively insert x_{high} into H

   (or make one-element flat tree for x_{high} if H was empty)

To find the second-smallest element

1. If D is empty, raise an exception
2. Let x_{high} be the minimum stored in H
3. Let x_{low} be the minimum stored in $D[x_{high}]$
4. Combine x_{high} and x_{low} and return the result

Expected time $O(1)$

Key subroutine in element removal

To remove an element x:

1. If x is the stored minimum:

    Set x and stored minimum to second-smallest value

    (still need to remove from recursive structure,
    just as if we were deleting it)

2. Compute x_{high} and x_{low}

3. If $D[x_{high}]$ is a one-element flat tree (its recursive D is empty):

    Recursively delete x_{high} from H

    Remove x_{high} from D

4. Else:

    Recursively delete x_{low} from $D[x_{high}]$

Summary

Summary

- Priority queue operations, application in Dijkstra's algorithm
- Limited set of operations in Python and how to work around it
- Binary heaps, $O(\log n)$ time per operation, $O(n)$ heapify
- k-ary heaps, slower delete-min and faster decrease-priority
- Optimal choice of k for k-ary heaps in Dijkstra's algorithm
- Fibonacci heaps, $O(\log n)$ delete-min, $O(1)$ decrease-priority
- Integer priority queues and their relation to integer sorting
- Bucket queues and Dial's algorithm
- Flat trees, $O(\log \log (\text{max priority}))$ per operation

References and image credits, I

Robert B. Dial. Algorithm 360: shortest-path forest with topological ordering. *Communications of the ACM*, 12(11):632–633, 1969. doi: 10.1145/363269.363610.

Yijie Han and Mikkel Thorup. Integer sorting in $O(n\sqrt{loglogn})$ expected time and linear space. In *43rd Symposium on Foundations of Computer Science (FOCS 2002), 16–19 November 2002, Vancouver, BC, Canada, Proceedings*, pages 135–144. IEEE Computer Society, 2002. doi: 10.1109/SFCS.2002.1181890.

Jeremy Segrott. GWR Fire Buckets, Severn Valley Railway. CC-BY licensed image, May 2 2015. URL https://commons.wikimedia.org/wiki/File: GWR_Fire_Buckets,_Severn_Valley_Railway_(17351340581).jpg.

Mikkel Thorup. Equivalence between priority queues and sorting. *Journal of the ACM*, 54(6):A28:1–A28:27, 2007. doi: 10.1145/1314690.1314692.

Peter van Emde Boas. Preserving order in a forest in less than logarithmic time. In *16th Annual Symposium on Foundations of Computer Science, Berkeley, California, USA, October 13–15, 1975*, pages 75–84. IEEE Computer Society, 1975. doi: 10.1109/SFCS.1975.26.