# CS 261: Data Structures

# Week 5: Priority queues

# Lecture 5c: Integer priority queues

**David Eppstein**
University of California, Irvine

Spring Quarter, 2023

# 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 $O\left(n\frac{\log X}{\log n}\right)$

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, "Equivalence between priority queues and sorting", JACM 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 & Thorup, "Integer sorting in $O(n\sqrt{\log\log n})$ expected time and linear space", FOCS 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 queue

Priority queue version of bucket sort

When all priorities are small non-negative integers:

- Make an array of buckets, 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, "Algorithm 360: Shortest-path forest with topological ordering", CACM 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)

- $x_{\text{low}}$ : `x & ((1 << b/2) - 1)`
- $x_{\text{high}}$ : `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 $O(\log b) = O\big(\log\log(\text{max priority})\big)$

# 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 $x_{\text{high}}$ for $x \in S \setminus \{\min\}$
- Dictionary $D$ mapping each $x_{\text{high}}$ to a recursive flat tree of values $x_{\text{low}}$ for the subset of elements $x \in S \setminus \{\min\}$ that have that value of $x_{\text{high}}$

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 $x_{\text{high}}$ 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 $x_{\text{high}}$ 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 $x_{\text{high}}$ is in $D$:

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

   Make one-element flat tree containing $x_{\text{low}}$

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

   Recursively insert $x_{\text{high}}$ into $H$
   (or make one-element flat tree for $x_{\text{high}}$ if $H$ was empty)

# To find the second-smallest element

1. If $D$ is empty, raise an exception
2. Let $x_{\text{high}}$ be the minimum stored in $H$
3. Let $x_{\text{low}}$ be the minimum stored in $D[x_{\text{high}}]$
4. Combine $x_{\text{high}}$ and $x_{\text{low}}$ 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 $x_{\text{high}}$ and $x_{\text{low}}$
3. If $D[x_{\text{high}}]$ is a one-element flat tree (its recursive $D$ is empty):

   Recursively delete $x_{\text{high}}$ from $H$

   Remove $x_{\text{high}}$ from $D$
4. Else:

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

# 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