1 Buckets

review shortest path algorithm.
In shortest paths, often have edge lengths small integers (say max C).
Observe heap behavior:

- heap min increasing (monotone property)
- max C distinct values
- (because don't insert $k + C$ until delete k).

Idea: lots of things have same value. Keep in buckets.
How to exploit?

- standard heaps of buckets. $O(m \log C)$ (slow) or $O(m + n \log C)$ with Fib (messy).
- Dial's algorithm: $O(m + nC)$.

space?

- use array of size $C + 1$
- wrap around

2-level buckets.

- make blocks of size b
- add layer on top: nC/b block summaries
- in each summary, keep count of items in block
- insert updates block and summary: $O(1)$
- ditto decrease key
- delete min scans summaries, then scans one block
  - over whole algorithm, nC/b scanning summaries
  - also, scan one block per delete min: b work
  - over n delete mins, work $nb + nC/b$
  - clearly, optimize with $b = \sqrt{C}$
- result: SP in $O(m + n\sqrt{C})$
- as before "space trick" to keep array sizes to C
- Generalize: 3 tiers?
  - blocks, superblocks, and summary

- block size $C^{1/3}$
- runtime $O(m + nC^{1/3})$
- how far can this go? To $m + n$? no, because insert cost rises.

Tries.

- depth k tree over array of size Δ
- depth k
- expansion factor $\Delta = (C + 1)^{1/k}$ (power of 2 simplifies)
- insert: $O(k)$ (also find, delete-non-min, decrease-key)
- delete-min: $O(k\Delta) = O(kC^{1/k})$ to find next element
- Shortest paths: $O(km + knC^{1/k})$
- Balance: $nC^{1/k} = m$ so $C = (m/n)^k$ so $k = \log(C)/\log(m/n)$
- Runtime: $m \log_{m/n}(C)$
- Space: $\Delta^k = C$ using circular array trick.

Problems: space and time
Idea: be lazy! (Denardo and Fox 1979)

- unique array on each level active
- keep other stuff piled up in list
- keep count of items in each block (not counting below)
- expand bucket when reach (and update block count)
- note: items descend once per touch, but never rise, so $O(k)$ expansion per item
- Insert
  - walk item down tree till stop in bucket
  - increment block count
  - real cost $O(k)$
  - also covers k cost of future expansions
- Decrease key
  - Remove from current bucket (decrement block count)
  - maybe descend (paid for already)
  - put in proper bucket (increment block count)

- $O(1)$

- Delete-min
  - remove item, advance to next
  - if no more items in block
  - rise to first nonempty block
  - traverse to first nonempty bucket
  - expand till find min
  - (may do pushdowns, but those are paid for)
  - (and identifying min happens during pushdowns)
  - so, scan only one block
  - cost $C^{1/k}$

- space to linear

- New time analysis:
  - $O(k)$ insert (charge expansions to insert)
  - $O(1)$ decrease key
  - $O(C^{1/k})$ delete-min

- paths runtime: $O(m + n(k + C^{1/k}))$, choose $k = 2\log C/\log\log C$: $O(m + n(\log C)/\log\log C)$

- Further improvement: heap on top (HOT) queues get $O(m + n(\log C)^{1/3})$ time. Cherkassky, Goldberg, and Silverstein. SODA 97.

- Implementation experiments—good model for project

2 VEB

Van Emde Boas, "Design and Implementation of an efficient priority queue" Math Syst. Th. 10 (1977)
Thorup, "On RAM priority queues" SODA 1996.
Idea

- idea: in bucket heaps, problem of finding next empty bucket was heap problem. Recurse!

- b-bit words

- $\log b$ running times

- thorup paper improves to $\log\log n$

- consequence for sorting.

Algorithm.

- need constant time hash table. non-trivial complexity theory, but can manage with randomization or slight time loss.
- queue Q on b bits is struct
  - $Q.\min$ is current min, *not* stored recursively
  - Array $Q.low[]$ of $\sqrt{U}$ queues on low order bits in bucket
  - $Q.high$, vEB queue on high order bits of elements other than current min in queue
- Insert x:
  - if $x < Q.\min$, swap
  - now insert x in recursive structs
  - expand $x = (x_h, x_l)$ high and low half words
  - If $Q.low[x_h]$ nonempty, then insert x_l in it
  - else, make new queue holding x_l at $Q.low[x_h]$, and insert x_h in $Q.high$
  - note two inserts, but one to an empty queue, so constant time
- Delete-min:
  - need to replace $Q.\min$
  - Look in $Q.high.\min$. if null, queue is empty.
  - else, gives first nonempty bucket x_h
  - Delete min from $Q.low[x_h]$ to finish finding $Q.\min$
  - If results in empty queue, Delete-min from $Q.high$ to remove that bucket from consideration
  - Note two delete mins, but second only happens when first was constant time.