17.3.8
Dijkstra using priority queues
Priority Queues

Data structure to store a set $S$ of $n$ elements where each element $\nu \in S$ has an associated real/integer key $k(\nu)$ such that the following operations:

1. **makePQ**: create an empty queue.
2. **findMin**: find the minimum key in $S$.
3. **extractMin**: Remove $\nu \in S$ with smallest key and return it.
4. **insert($\nu$, $k(\nu)$)**: Add new element $\nu$ with key $k(\nu)$ to $S$.
5. **delete($\nu$)**: Remove element $\nu$ from $S$.
6. **decreaseKey($\nu$, $k'(\nu)$)**: decrease key of $\nu$ from $k(\nu)$ (current key) to $k'(\nu)$ (new key). Assumption: $k'(\nu) \leq k(\nu)$.
7. **meld**: merge two separate priority queues into one.

All operations can be performed in $O(\log n)$ time.

**decreaseKey** is implemented via **delete** and **insert**.
Priority Queues

Data structure to store a set $S$ of $n$ elements where each element $v \in S$ has an associated real/integer key $k(v)$ such that the following operations:

1. **makePQ**: create an empty queue.
2. **findMin**: find the minimum key in $S$.
3. **extractMin**: Remove $v \in S$ with smallest key and return it.
4. **insert($v, k(v)$)**: Add new element $v$ with key $k(v)$ to $S$.
5. **delete($v$)**: Remove element $v$ from $S$.
6. **decreaseKey($v, k'(v)$)**: decrease key of $v$ from $k(v)$ (current key) to $k'(v)$ (new key). Assumption: $k'(v) \leq k(v)$.
7. **meld**: merge two separate priority queues into one.

All operations can be performed in $O(\log n)$ time.

**decreaseKey** is implemented via **delete** and **insert**.
Priority Queues

Data structure to store a set $S$ of $n$ elements where each element $v \in S$ has an associated real/integer key $k(v)$ such that the following operations:

1. **makePQ**: create an empty queue.
2. **findMin**: find the minimum key in $S$.
3. **extractMin**: Remove $v \in S$ with smallest key and return it.
4. **insert($v, k(v)$)**: Add new element $v$ with key $k(v)$ to $S$.
5. **delete($v$)**: Remove element $v$ from $S$.
6. **decreaseKey($v, k'(v)$)**: decrease key of $v$ from $k(v)$ (current key) to $k'(v)$ (new key). Assumption: $k'(v) \leq k(v)$.
7. **meld**: merge two separate priority queues into one.

All operations can be performed in $O(\log n)$ time.

**decreaseKey** is implemented via **delete** and **insert**.
Dijkstra’s Algorithm using Priority Queues

\[
Q \leftarrow \text{makePQ}()
\]
\[
\text{insert}(Q, (s, 0))
\]
\[
\text{for each node } u \neq s \text{ do}
\]
\[
\quad \text{insert}(Q, (u, \infty))
\]
\[
X \leftarrow \emptyset
\]
\[
\text{for } i = 1 \text{ to } |V| \text{ do}
\]
\[
\quad (v, \text{dist}(s, v)) = \text{extractMin}(Q)
\]
\[
X = X \cup \{v\}
\]
\[
\text{for each } u \text{ in } \text{Adj}(v) \text{ do}
\]
\[
\quad \text{decreaseKey}\left(Q, (u, \min(\text{dist}(s, u), \text{dist}(s, v) + \ell(v, u)))\right).
\]

Priority Queue operations:
1. **\(O(n)\)** insert operations
2. **\(O(n)\)** extractMin operations
3. **\(O(m)\)** decreaseKey operations
Implementing Priority Queues via Heaps

Using Heaps

Store elements in a heap based on the key value

- All operations can be done in $O(\log n)$ time

Dijkstra’s algorithm can be implemented in $O((n + m) \log n)$ time.
Implementing Priority Queues via Heaps

Using Heaps

Store elements in a heap based on the key value
* All operations can be done in $O(\log n)$ time

Dijkstra’s algorithm can be implemented in $O((n + m)\log n)$ time.
Priority Queues: Fibonacci Heaps/Relaxed Heaps

Fibonacci Heaps

1. extractMin, insert, delete, meld in $O(\log n)$ time
2. decreaseKey in $O(1)$ amortized time: $\ell$ decreaseKey operations for $\ell \geq n$ take together $O(\ell)$ time
3. Relaxed Heaps: decreaseKey in $O(1)$ worst case time but at the expense of meld (not necessary for Dijkstra’s algorithm)

Dijkstra’s algorithm can be implemented in $O(n \log n + m)$ time. If $m = \Omega(n \log n)$, running time is linear in input size.

Data structures are complicated to analyze/implement. Recent work has obtained data structures that are easier to analyze and implement, and perform well in practice. Rank-Pairing Heaps, for example.

Boost library implements both Fibonacci heaps and rank-pairing heaps.
Priority Queues: Fibonacci Heaps/Relaxed Heaps

### Fibonacci Heaps

1. **extractMin, insert, delete, meld** in $O(\log n)$ time
2. **decreaseKey** in $O(1)$ amortized time: $\ell$ decreaseKey operations for $\ell \geq n$ take together $O(\ell)$ time
3. Relaxed Heaps: **decreaseKey** in $O(1)$ worst case time but at the expense of **meld** (not necessary for Dijkstra’s algorithm)

### Additional Notes

1. Dijkstra’s algorithm can be implemented in $O(n \log n + m)$ time. If $m = \Omega(n \log n)$, running time is linear in input size.
2. Data structures are complicated to analyze/implement. Recent work has obtained data structures that are easier to analyze and implement, and perform well in practice. Rank-Pairing Heaps, for example.
3. Boost library implements both Fibonacci heaps and rank-pairing heaps.
Priority Queues: Fibonacci Heaps/Relaxed Heaps

**Fibonacci Heaps**

1. extractMin, insert, delete, meld in $O(\log n)$ time
2. decreaseKey in $O(1)$ amortized time: $\ell$ decreaseKey operations for $\ell \geq n$ take together $O(\ell)$ time
3. Relaxed Heaps: decreaseKey in $O(1)$ worst case time but at the expense of meld (not necessary for Dijkstra’s algorithm)

Dijkstra’s algorithm can be implemented in $O(n \log n + m)$ time. If $m = \Omega(n \log n)$, running time is linear in input size.

Data structures are complicated to analyze/implement. Recent work has obtained data structures that are easier to analyze and implement, and perform well in practice. Rank-Pairing Heaps, for example.

Boost library implements both Fibonacci heaps and rank-pairing heaps.
Priority Queues: Fibonacci Heaps/Relaxed Heaps

Fibonacci Heaps

1. extractMin, insert, delete, meld in $O(\log n)$ time
2. decreaseKey in $O(1)$ amortized time: $\ell$ decreaseKey operations for $\ell \geq n$ take together $O(\ell)$ time
3. Relaxed Heaps: decreaseKey in $O(1)$ worst case time but at the expense of meld (not necessary for Dijkstra’s algorithm)

Dijkstra’s algorithm can be implemented in $O(n \log n + m)$ time. If $m = \Omega(n \log n)$, running time is linear in input size.

Data structures are complicated to analyze/implement. Recent work has obtained data structures that are easier to analyze and implement, and perform well in practice. Rank-Pairing Heaps, for example.

Boost library implements both Fibonacci heaps and rank-pairing heaps.
THE END
...
(for now)