

Algorithm Theory, Tutorial 4

Johannes Kalmbach

University of Freiburg

johannes.kalmbach@gmail.com

November 2018

O(1) Priority Queue

- Assume we want to store $(key, data)$ -pairs in a priority queue.
- The priorities (keys) are only from the set $\{1, \dots, c\}$ and $c \in \mathbb{N}$ is constant.

Describe a priority queue that provides the operations $\text{Insert}(key, data)$, Get-Min , Delete-Min , and $\text{Decrease-Key}(pointer, newkey)$ all in constant time for the given scenario, and describe how these operations work on your data structure.

Solution

- We use an array $A[1, \dots, c]$ of size c .

Solution

- We use an array $A[1, \dots, c]$ of size c .
- Each array entry contains a reference to a doubly linked list of $(key, data)$ -pairs.

Solution

- We use an array $A[1, \dots, c]$ of size c .
- Each array entry contains a reference to a doubly linked list of $(key, data)$ -pairs.
- $\text{Insert}(key, data)$: simply append $(key, data)$ to the list in $A[key]$ in $\mathcal{O}(1)$.

Solution

- We use an array $A[1, \dots, c]$ of size c .
- Each array entry contains a reference to a doubly linked list of $(key, data)$ -pairs.
- $\text{Insert}(key, data)$: simply append $(key, data)$ to the list in $A[key]$ in $\mathcal{O}(1)$.
- Get-Min : We iterate the Array starting from the beginning (in $\mathcal{O}(c) = \mathcal{O}(1)$), until we find a non-empty list at index i . We return the first pair $(i, data)$ from that list.

Solution

- We use an array $A[1, \dots, c]$ of size c .
- Each array entry contains a reference to a doubly linked list of $(key, data)$ -pairs.
- $\text{Insert}(key, data)$: simply append $(key, data)$ to the list in $A[key]$ in $\mathcal{O}(1)$.
- Get-Min : We iterate the Array starting from the beginning (in $\mathcal{O}(c) = \mathcal{O}(1)$), until we find a non-empty list at index i . We return the first pair $(i, data)$ from that list.
- Delete-Min : We iterate the Array starting from the beginning (in $\mathcal{O}(c) = \mathcal{O}(1)$), until we find a non-empty list at index i . We remove the first pair $(i, data)$ from that list and return it.

- We use an array $A[1, \dots, c]$ of size c .
- Each array entry contains a reference to a doubly linked list of $(key, data)$ -pairs.
- $\text{Insert}(key, data)$: simply append $(key, data)$ to the list in $A[key]$ in $\mathcal{O}(1)$.
- Get-Min : We iterate the Array starting from the beginning (in $\mathcal{O}(c) = \mathcal{O}(1)$), until we find a non-empty list at index i . We return the first pair $(i, data)$ from that list.
- Delete-Min : We iterate the Array starting from the beginning (in $\mathcal{O}(c) = \mathcal{O}(1)$), until we find a non-empty list at index i . We remove the first pair $(i, data)$ from that list and return it.
- $\text{Decrease-Key}(pointer, newkey)$: Since we have a pointer to the $(key, data)$ -pair in question, we can remove and change its key in $\mathcal{O}(1)$. Afterwards we reinsert it into the correct list also in $\mathcal{O}(1)$.

- State how fast Prim's algorithm to compute a minimum spanning tree is, under the assumption that edge weights are in the set $\{1, \dots, c\}$ and $c \in \mathbb{N}$ is constant, using your implementation of a priority queue. Explain your answer.
- Prim's Algorithm now runs in $\mathcal{O}(|E| + |V|)$ using our implementation of the priority queue.
- The reason is that Prim's algorithm uses $\mathcal{O}(|E|)$ Decrease-Key operations and $\mathcal{O}(|V|)$ Delete-Min, Get-Min and Insert operations (see analysis in lecture slides).

Exercise 2

We are given a maximum flow network $G = (V, E)$ with integer capacities together with a maximum flow Φ . Describe an algorithm with time complexity $O(|V| + |E|)$ to compute a new maximum flow for each of the following cases:

- Ⓐ if the capacity of an arbitrary edge $(u, v) \in E$ increases by one unit.
- Ⓑ if the capacity of an arbitrary edge $(u, v) \in E$ decreases by one unit.

Solution 2a

- After increasing the capacity of one edge, the flow is still valid

Solution 2a

- After increasing the capacity of one edge, the flow is still valid
- But it might not be maximal anymore

Solution 2a

- After increasing the capacity of one edge, the flow is still valid
- But it might not be maximal anymore
- Maximum flow in new network can be at most bigger by one.

Solution 2a

- After increasing the capacity of one edge, the flow is still valid
- But it might not be maximal anymore
- Maximum flow in new network can be at most bigger by one.
- Run one iteration of Ford-Fulkerson ($O(|E| + |V|)$)

Solution 2a

- After increasing the capacity of one edge, the flow is still valid
- But it might not be maximal anymore
- Maximum flow in new network can be at most bigger by one.
- Run one iteration of Ford-Fulkerson ($O(|E| + |V|)$)
- If we find an augmenting path, augment Φ by this path.

Solution 2b

- After decreasing the capacity of an edge $e = (u, v)$ by one our flow might not be valid anymore.

Solution 2b

- After decreasing the capacity of an edge $e = (u, v)$ by one our flow might not be valid anymore.
- If our flow is still valid ($\Phi(e) \leq c_{orig}(e) - 1$) we have nothing to do (our max flow can get bigger by decreasing capacities).

Solution 2b

- After decreasing the capacity of an edge $e = (u, v)$ by one our flow might not be valid anymore.
- If our flow is still valid ($\Phi(e) \leq c_{orig}(e) - 1$) we have nothing to do (our max flow can get bigger by decreasing capacities).
- Otherwise we have to repair our flow: (one of many solutions)

Solution 2b

- After decreasing the capacity of an edge $e = (u, v)$ by one our flow might not be valid anymore.
- If our flow is still valid ($\Phi(e) \leq c_{orig}(e) - 1$) we have nothing to do (our max flow can get bigger by decreasing capacities).
- Otherwise we have to repair our flow: (one of many solutions)
- Search path from s to u and from v to t with positive flow. Reduce flow by one on each edge of those paths. (of course also reduce flow on (u, v) . (Reduces flow value by one)

Solution 2b

- After decreasing the capacity of an edge $e = (u, v)$ by one our flow might not be valid anymore.
- If our flow is still valid ($\Phi(e) \leq c_{orig}(e) - 1$) we have nothing to do (our max flow can get bigger by decreasing capacities).
- Otherwise we have to repair our flow: (one of many solutions)
- Search path from s to u and from v to t with positive flow. Reduce flow by one on each edge of those paths. (of course also reduce flow on (u, v)). (Reduces flow value by one)
- Afterwards run FF to see if we can again increase the flow to its "original" size, if possible, augment it.

Linear Chain in Fibonacci Heap

Show that for any positive integer n , there exists a sequence of Fibonacci Heap operations that can construct a Fibonacci Heap consisting of just one tree that is a linear chain of n nodes. Provide the pseudocode of a recursive procedure to construct such a Fibonacci Heap, and show its correctness.

- Hint: Search for easy recursive solutions.
- Assume we can build a linear chain of length n and extend it to $n + 1$.
- Recursion and Induction are basically the same then

Base case

- H is an empty Fib. Heap

Base case

- H is an empty Fib. Heap
- `H.insert(1)`

Base case

- H is an empty Fib. Heap
- `H.insert(1)`
- `H.insert(2)`

Base case

- H is an empty Fib. Heap
- `H.insert(1)`
- `H.insert(2)`
- `H.insert(3)`

Base case

- H is an empty Fib. Heap
- `H.insert(1)`
- `H.insert(2)`
- `H.insert(3)`
- `H.deleteMin()`

Base case

- H is an empty Fib. Heap
- `H.insert(1)`
- `H.insert(2)`
- `H.insert(3)`
- `H.deleteMin()`
- We have successfully constructed a chain of length 1.

Inductive Step, $n \mapsto n + 1$

- $H := \text{linChain}(n)$

Inductive Step, $n \mapsto n + 1$

- `H:=linChain(n)`
- `m := H.getMin()`

Inductive Step, $n \mapsto n + 1$

- `H:=linChain(n)`
- `m := H.getMin()`
- `H.insert(m-1)`

Inductive Step, $n \mapsto n + 1$

- `H:=linChain(n)`
- `m := H.getMin()`
- `H.insert(m-1)`
- `H.insert(m-2)`

Inductive Step, $n \mapsto n + 1$

- `H:=linChain(n)`
- `m := H.getMin()`
- `H.insert(m-1)`
- `H.insert(m-2)`
- `x:=H.insert(m+1)`

Inductive Step, $n \mapsto n + 1$

- `H:=linChain(n)`
- `m := H.getMin()`
- `H.insert(m-1)`
- `H.insert(m-2)`
- `x:=H.insert(m+1)`
- `H.deleteMin()`

Inductive Step, $n \mapsto n + 1$

- `H:=linChain(n)`
- `m := H.getMin()`
- `H.insert(m-1)`
- `H.insert(m-2)`
- `x:=H.insert(m+1)`
- `H.deleteMin()`
- `H.decreaseKey(x, m-2)`

Inductive Step, $n \mapsto n + 1$

- `H:=linChain(n)`
- `m := H.getMin()`
- `H.insert(m-1)`
- `H.insert(m-2)`
- `x:=H.insert(m+1)`
- `H.deleteMin()`
- `H.decreaseKey(x, m-2)`
- `H.deleteMin()`

Inductive Step, $n \mapsto n + 1$

- `H:=linChain(n)`
- `m := H.getMin()`
- `H.insert(m-1)`
- `H.insert(m-2)`
- `x:=H.insert(m+1)`
- `H.deleteMin()`
- `H.decreaseKey(x, m-2)`
- `H.deleteMin()`
- We have successfully transformed a linear chain of length n into a linear chain of length $n + 1$.

Algorithm 1 Chain-Construction(n)

if $n = 1$ **then**

 Initialize-Heap(F)

 ▷ assume F is now globally known

 Insert($F, 1$)

 ▷ inserting a node with key 1 into F .

 Insert($F, 2$)

 Insert($F, 3$)

 Delete-Min(F)

return

Chain-Construction($n - 1$)

$min \leftarrow \text{Get-Min}(F)$

$x \leftarrow \text{Insert}(F, min + 1)$

Insert($F, min - 1, null$)

Insert($F, min - 2, null$)

Delete-Min(F)

Decrease-Key($x, min - 3$)

Delete-Min(F)

return

Min cut with min number of edges

- This exercise will be considered as a bonus exercise, which earns points but does not count towards the threshold of exam admittance.
- Consider an undirected, weighted graph $G = (V, E)$ with integral edge weights. Among all cuts of G with minimum weight you want to find a cut $(S, V \setminus S)$ with the smallest number of edges (i.e. edges with exactly one endpoint in S).
 - a) Modify the weights of G to create a new graph G' in which any minimum cut in G' is a minimum cut with the smallest number of edges in G .
 - b) Prove that G' has the property claimed in part (a).

Solution

- Let $G = (V, E, w)$ be a weighted undirected graph with integer edge weights $w(e) \geq 0$ for $e \in E$.

Solution

- Let $G = (V, E, w)$ be a weighted undirected graph with integer edge weights $w(e) \geq 0$ for $e \in E$.
- We define $G' = (V, E, w')$ with edge weights $w'(e) := |E| \cdot w(e) + 1$.

Solution

- Let $G = (V, E, w)$ be a weighted undirected graph with integer edge weights $w(e) \geq 0$ for $e \in E$.
- We define $G' = (V, E, w')$ with edge weights $w'(e) := |E| \cdot w(e) + 1$.
- We have to prove two things:

Solution

- Let $G = (V, E, w)$ be a weighted undirected graph with integer edge weights $w(e) \geq 0$ for $e \in E$.
- We define $G' = (V, E, w')$ with edge weights $w'(e) := |E| \cdot w(e) + 1$.
- We have to prove two things:
- Every min cut of G has less weight in G' than every non-minimal cut of G

Solution

- Let $G = (V, E, w)$ be a weighted undirected graph with integer edge weights $w(e) \geq 0$ for $e \in E$.
 - We define $G' = (V, E, w')$ with edge weights $w'(e) := |E| \cdot w(e) + 1$.
 - We have to prove two things:
 - Every min cut of G has less weight in G' than every non-minimal cut of G
 - Of two min-cuts in G the one with fewer edges has less weight in G' .
- (2)

Proof of Claim 1

- Every min cut of G has less weight in G' than every non-minimal cut of G (1)
- Let M be a min cut in G and X a non-minimal cut in G
- Let $|M|, |X|$ be the number of edges of the two cuts and $w_G(M) < w_G(X)$ the weights of the two cuts in G ($w_{G'}(\dots)$ in analogy).
- It holds that $w_G(M) \leq w_G(X) + 1$ (because of the Integer weights)

Proof of Claim 2

- Of two min-cuts in G the one with fewer edges has less weight in G' .
(2)
- Let M and X be min cuts in G ($w_G(M) = w_G(X)$) and let M have fewer edges than X ($|M| < |X|$).

