Theory of Computing

Lecture 6

MAS 714

Hartmut Klauck

Data Structure: Priority Queue

- Store (up to) n elements and their keys (keys are numbers)
- Operations:
  - ExtractMin: Get (and remove) the element with minimum key
  - DecreaseKey(v,x): replace key(v) with a smaller value x
  - Initialize
  - Insert(v,key(v))
  - Test for emptiness

Priority Queues

- We will show how to implement a priority queue with time O(log n) for all operations
- This leads to total time O((n+m) log n) for the Dijkstra algorithm
- Slightly suboptimal : we would like O(n log n + m)
  - Much more difficult to achieve

Heaps

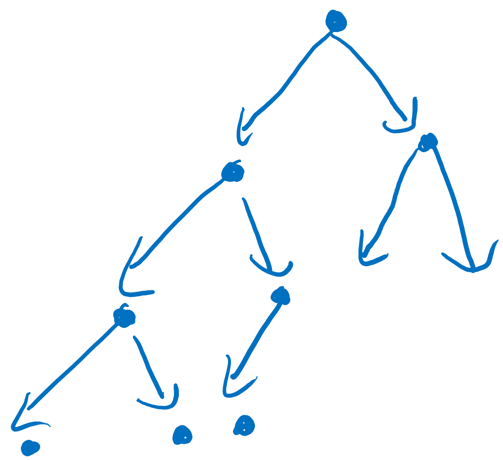
- We will implement a priority queue with a *heap*
- Heaps can also be used for sorting!
  - Heapsort:
    Insert all elements, ExtractMin until empty
- If all operations take time log n we have sorting in time O(n log n)

Heaps

- A heap is an array of length n
  - can hold at most n elements
- The elements in the array are not sorted by keys, but their order has the *heap-property*
- Namely, they can be viewed as a tree, in which parents are smaller than their children
  - ExtractMin is easy (at the root)
  - Unfortunately we need to work to maintain the heap-property after removing the root

Heaps

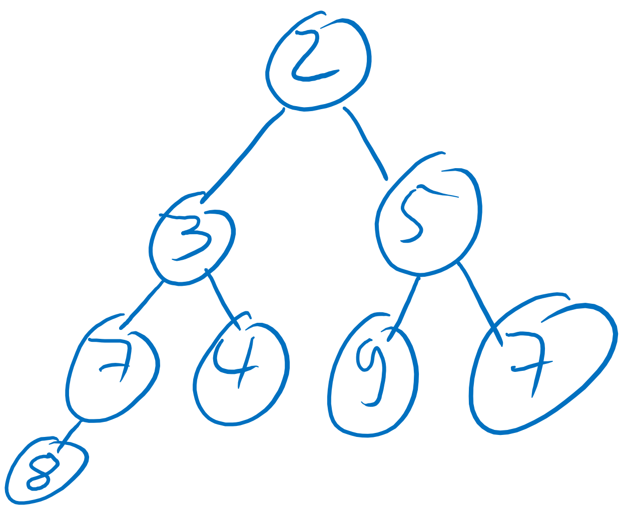
- Keys in a heap are arranged as a full binary tree where the last level is filled from the left up to some point
- Example:



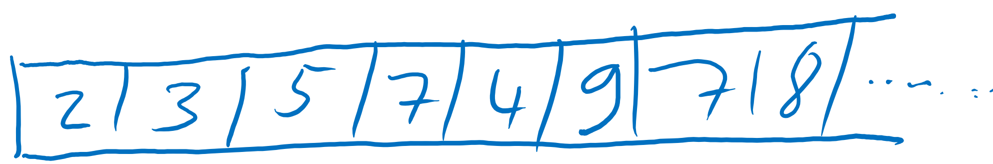
Heaps

- Heap property:
  - Element in cell i is smaller than elements in cells 2i and 2i+1
- Example:

Tree



Array



Heaps

- Besides the array storing the keys we also keep a counter SIZE that tells us how many keys are in H
  - in cells 1…SIZE

Simple Procedures for Heaps

- Initialize:
  - Declare the array H of correct length n
  - Set SIZE to 0
- Test for Empty:
  - Check SIZE
- FindMin:
  - Minimum is in H[1]
- All this in time O(1)

Simple Procedures for Heaps

- Parent(i) is $\lfloor i/2 \rfloor$
- LeftChild(i) is 2i
- Rightchild(i) is 2i+1

Procedures for Heaps

- Suppose we remove the root, and replace it with the *rightmost* element of the heap
- Now we still have a tree, but we (probably) violate the heap property at some vertex i, i.e., $H[i]>H[2i]$ or $H[i]>H[2i+1]$
- The procedure Heapify(i) will fix this
- Heapify(i) assumes that the subtrees below i are correct heaps, but there is a (possible) violation at i
- And no other violations in H (i.e., above i)

Procedures for Heaps

- Heapify(i)
  - l=LeftChild(i), r=Rightchild(i)
  - If l$\leq$SIZE and H[l]<H[i] Smallest=l else Smallest=i
  - If r$\leq$SIZE and H[r]<H[Smallest] Smallest=r
  - If Smallest$\neq$i
    - Swap H[i] and H[Smallest]
    - Heapify(Smallest)

Heapify

- Running Time:
  - A heap with SIZE=n has depth at most log n
  - Running time is dominated by the number of recursive calls
  - Each call leads to a subheap that is 1 level shallower
  - Time O(log n)

Procedures for Heaps

- ExtractMin():
  - Return H[1]
  - H[1]=H[SIZE]
  - SIZE=SIZE-1
  - Heapify(1)

- Time is O(log n)

Procedures for Heaps

- DecreaseKey(i,key)
  - If H[i]<key return error
  - H[i]=key \\Now parent(i) might violate heap property
  - While i>1 and H[parent(i)]>H[i]
    - Swap H[parent(i)] and H[i], i=parent(i)

      \\Move the element towards the root
- Time is O(log n)

Procedures for Heaps

- Insert(key):
  - SIZE=SIZE+1
  - H[SIZE]=∞
  - DecreaseKey(SIZE,key)

- Time is O(log n)

Note for Dijkstra

- DecreaseKey(i,x) works on the vertex that is stored in position i in the heap
- But we want to decrease the key for vertex v!
- We need to remember the position of all v in the heap H
- Keep an array pos[1...n]
  - Whenever we move a vertex in H we need to change pos

The single-source shortest-path problem with negative edge weights

- Graph G, weight function W, start vertex s
- Output: a bit indicating if there is a negative cycle reachable from s
  AND (if not) the shortest paths from s to all v

Bellman-Ford Algorithm

- Initialize: $d(s)=0$, $\pi(s)=s$, $d(v)=\infty$, $\pi(v)=NIL$ for other v
- For i=1 to n-1:
  - Relax all edges (u,v)
- For all (u,v): if $d(v)>d(u)+W(u,v)$ then output: „negative cycle!“

- Remark: d(v) and $\pi(v)$ contain distance from s and predecessor in a shortest path tree

Running time

- Running time is O(nm)
  - n-1 times relax all m edges

Correctness

- Assume that no cycle of negative length is reachable from s
- **Theorem:** After n-1 iterations of the for-loop we have $d(v)=\delta(s,v)$ for all v.

- **Lemma:** Let $v_0,\ldots,v_k$ be a shortest path from $s=v_0$ to v_k. Relax edges $(v_0,v_1)\ldots.(v_{k-1},v_k)$ successively. Then $d(v_k)=\delta(s,v_k)$. This holds regardless of other relaxations performed.

Correctness

- Proof of the theorem:
  - Let v denote a reachable vertex
  - Let s, …,v be a shortest path with k edges
  - $k \leq$ n-1 can be assumed (why?)
  - In every iteration all edges are relaxed
  - By the lemma d(v) is correct after $k \leq$ n-1 iterations
- For all unreachable vertices we have $d(v)=\infty$ at all times
- To show: the algorithm decides the existence of negative cycles correctly
- No neg. cycle present/reachable: for all edges (u,v):
  - $d(v)=\delta(s,v)\leq\delta(s,u)+W(u,v)=d(u)+W(u,v)$, pass test

Correctness

- If a negative cycle exists:
  - Let $v_0,\ldots,v_k$ be a (reachable) path with negative length and $v_0=v_k$
  - Assume the algorithm does NOT stop with error message, then
    - $d(v_i) \leq d(v_{i-1}) + W(v_{i-1},v_i)$ for all i=1...k
    - Hence

$$\sum_{i=1}^{k} d(v_i) \leq \sum_{i=1}^{k} \left(d(v_{i-1}) + W(v_{i-1}, v_i) \right)$$

Correctness

$$\sum_{i=1}^{k} d(v_i) \leq \sum_{i=1}^{k} d(v_{i-1}) + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- $v_0=v_k$, so

$$\sum_{i=1}^{k} d(v_i) = \sum_{i=1}^{k} d(v_{i-1})$$

- $d(v_i) < \infty$ in the end for all reachable vertices, hence

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) \geq 0$$

The Lemma

Lemma: Let $v_0,\ldots,v_k$ be a shortest path from $s=v_0$ to v_k. Relax edges $(v_0,v_1)\ldots.(v_{k-1},v_k)$ successively. Then $d(v_k)=\delta(s,v_k)$. This holds regardless of other relaxations performed.

Proof:

- By induction. After relaxing (v_{i-1}, v_i) the value $d(v_i)$ is correct.
- Base: $i=0$, $d(v_0)=d(s)=0$ is correct.
- Assume $d(v_{i-1})$ correct. According to an earlier observation after relaxing (v_{i-1},v_i) also $d(v_i)$ correct.
- Once $d(v)$ is correct, the value stays correct.
  - $d(v)$ is always an upper bound

Application of Bellman Ford

- Graph is a distributed network
  - vertices are processors that can communicate via edges
- We look for distance/shortest path of vertices from s
- Computation can be performed in a distributed way, without
  - global control
  - global knowledge about the network
- Dijkstra needs global knowledge
- Running time: n-1 phases, vertices compute (in parallel)