Prim’s Algorithm

* Start with vertex 0 and greedily grow tree T
* Add to T the min weight edge with exactly one endpoint in T
* Repeat until V-1 edges

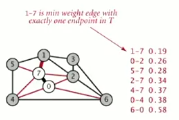
Proposition: Prim’s algorithm computes the MST

Proof: Prim’s algorithm is a special case of the greedy MST algorithm

* Suppose edge e= min weight edge connecting a vertex on the tree to a vertex not on the tree
* Cut = set of vertices connected on tree
* No crossing edge is black (from tree vertices to non-tree vertex)
* No crossing edge has lower weight (by design, we are choosing the minimum)

Challenge: find the min weight edge with exactly one endpoint in T

Difficulty?



* E : try all edges
* Log E : use a priority queue

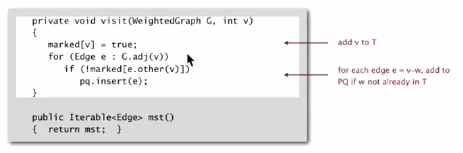
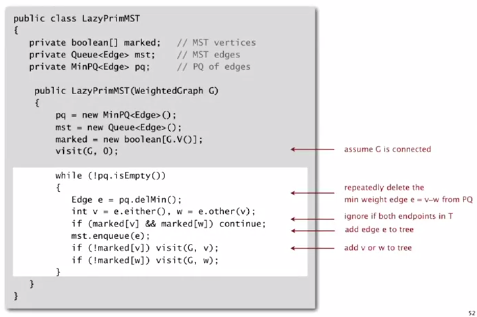
Lazy implementation   
*(Lazy because we allow everything in queue even if obsolete, then test whether it’s in the tree)*

Challenge: find the minimum weight edge with exactly one endpoint in T

Lazy solution: Maintain a PQ of edges with (at least) one endpoint in T

* Key = edge; priority = weight of edge
* Delete-min to determine next edge e = v-w to add to T
* Disregard if both endpoints v and w are in T
* Otherwise, let w be the vertex not in T:
  + Add to PQ any edge incident to w (assuming other endpoint not in T)
  + Add w to T

Prim’s Algorithm Lazy Implementation



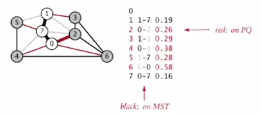
Running time:

Proposition: Lazy Prim’s algorithm computes the MST in time proportional to E log E and extra space proportional to E (in the worst case)



Prim’s algorithm Eager Implementation

Challenge: Find the minimum weight edge with exactly one endpoint in T

Eager solution: Maintain a PQ of vertices (pq has at most one entry per vertex) connected by an edge to T, where priority of vertex v = weight of shortest edge connecting v to T. 

* Delete min vertex v and add its associated edge e = v-w to T
* Update PQ by considering all edges e = v-x incident to v
  + Ignore if x is already in T
  + Add x to PQ if not already on it
  + Decrease priority of x if v-x becomes shortest edge connecting x to T

1. Start with vertex 0 and greedily grow tree T
2. Add to T the min weight edge with exactly one endpoint in T
3. Repeat until V -1 edges

Code is quite similar to code of lazy version

Key data structure needed for implementation:

Indexed priority queue

Associate an index between 0 and N – 1 with each key in a priority queue

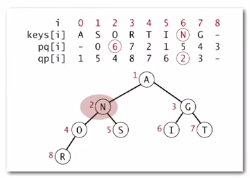
* Client can insert and delete-the-minimum
* Client can change the key by specifying the index

Indexed priority queue API

Public class IndexMinPQ<Key extends Comparable<Key>>  
IndexMinPQ(int N) : create indexed priority queue with indices 0, 1, … N-1  
void insert(int I, Key key) : associate key with index i  
void decreaseKey(int I, Key key) : decrease the key associated with index I   
boolean contains(int i) : is I an index on the priority queue?  
int delMin() : remove a minimal key and return its associated index  
boolean isEmpty() : is the priority queue empty?  
int size() : number of entries in the priority queue

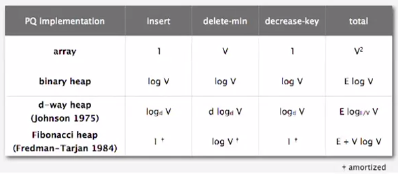
Implementation

* Start with same code as MinPQ
* Maintain parallel arrays keys[], pq[], and qp[] so that:
  + Keys[i] is the priority of i
  + Pq[i] is the index of the key in heap position i
  + Qp[i] is the heap position of the key with index i
* Use swim(qp[k]) implement decreaseKey(k, key)\



Running time

Depends on PQ implementation: V insert, V delete-min, E decrease-key



Bottom Line

* Array implementation optimal for dense graphs
* Binary heap much faster for sparse graphs
* 4-way heap worth the trouble in performance-critical situations
* Fibonacci heap best in theory, but not worth implementing