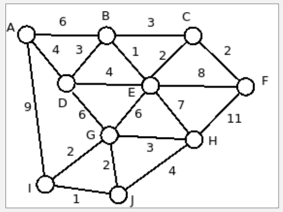
Algorithms for MST

Prim's algorithm –
- start with a tree T containing a single vertex S
- repeatedly add the cheapest edge connecting a vertex in S and a vertex in V-S to T



Prim's Algorithm

→ O(makePQ + n × removeMin + m × decreaseKey) total

operation	heap
makeQueue	O(n log n) – repeated insert O(n) – heapify
removeMin	O(log n)
decreaseKey	O(log n) (**)
total running time for Prim's algorithm	O(n + n log n + m log n) = O((n+m) log n) = O(m log n) for connected graphs

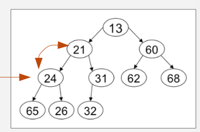
This is –
- O(n log n) for sparse graphs [m = O(n)]
- O(n^2 log n) for dense graphs [m = O(n^2)]

(**) assuming O(1) to locate element within PQ (which can be done with a *locator*), otherwise O(n) to search PQ to find entry

Locators

- decrease key can be done in O(log n) time if the location (index) of the key in the heap is known
  - update key, then bubble up (min heap) to fix ordering property

change 24 to 15



- searching for a key in a heap –
  - O(n) – may need to check all elements
- how to avoid search?
  - look up instead!
  - what if we add a map key → array index?
    - O(1) to locate key with hashtable
    - map must be updated for each bubble up or bubble down swap
      - but that only involves two elements, so two updates – +O(1) per swap doesn't change the big-Oh for bubbling

Prim's Algorithm

→ O(makePQ + n × removeMin + m × decreaseKey) total

operation	array – unsorted	heap
makeQueue	O(n) – take elements in whatever order	O(n log n) – repeated insert O(n) – heapify
removeMin	O(n) – search, then swap with last	O(log n)
decreaseKey	O(1) – update key value (**)	O(log n) (**)
total running time for Prim's algorithm	O(n + n^2 + m) = O(n^2)	O(n + n log n + m log n) = O((n+m) log n) = O(m log n) for connected graphs

Observation:
- for sparse graphs [m = O(n)], the heap is more efficient
- for dense graphs [m = O(n^2)], the unsorted array is more efficient

(**) assuming O(1) to locate element within PQ (which can be done with a *locator*), otherwise O(n) to search PQ to find entry

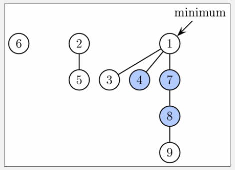
Other Options

(Binary) heaps are not the only choice for implementing priority queues.

- *d*-ary heaps
  - each node has *d* children instead of two, reducing the height of the tree by a factor of log d
  - insert – O(log n / log d)
  - removeMin – O(d log n / log d)
    - have to check all *d* children at each level to determine smallest element
  - *d* should be chosen to be m/n (the average degree of the graph)
  - → total time for Prim's algorithm:
    O((nd+m) log n / log d) = O(m log n / log (m/n))
    - for sparse graphs [m = O(n)], get O(n log n) – as good as a binary heap
    - for dense graphs [m = O(n^2)], get O(n^2) – as good as an unsorted array
    - in between [m = $n^{1+\delta}$], get O(m) - δ is a constant

Other Options

(Binary) heaps are not the only choice.



- Fibonacci heaps
  - maintain a forest of heaps
  - insert/remove involves splitting and merging heaps in order to keep the degree of each node low and the size of each subtree sufficiently high
  - achieves O(log n) removeMin and O(1) decreaseKey
    - *amortized time* – on average
  - → total time for Prim's algorithm: O(m + n log n)
    - for sparse graphs [m = O(n)], get O(n log n) – as good as a binary heap
    - for dense graphs [m = O(n^2)], get O(n^2) – as good as an unsorted array

MST

Running time?

- Prim's edge-based version is O(m log n) for a heap-based PQ
  - O(n log n) for sparse graphs, O(n^2 log n) for dense
- Prim's vertex-based (typical) version is O((n+m) log n) for a heap-based PQ
  - O(n log n) for sparse graphs, O(n^2 log n) for dense
  - can do better with a fancier PQ implementation
    - O(n log n) for sparse, O(n^2) for dense
- Kruskal's is O((n+m) log n) or O(m log n) amortized depending on union-find implementation
  - O(n log n) for sparse graphs, O(n^2 log n) for dense

MST

Prim's or Kruskal's?

- can achieve better running time with Prim's algorithm and a fancy PQ implementation
- (standard) PQ is a more common data structure than union-find (or a fancy PQ)
- need to repeat Prim's on each connected component if the graph is not connected
  - Kruskal's handles disconnected graphs without anything additional

Takeaways

- definitions: spanning tree, minimum spanning tree
- algorithms for MST – kruskal's, prim's
  - what the algorithm is – be able to trace
  - running time and pros/cons of each algorithm
- union-find data structure
  - operations – makeset, find, union
  - union-by-rank list implementation – what it is, running time
  - union-by-rank tree implementation – running time
  - as an example of an incremental approach to data structure development