Chapter 4. Priority queue implementations (우선순위 큐 구현)

4.1 Array (배열)

-Simplest implementation of a priority queue is an unordered array of key values for all potential elements

우선순위 큐의 가장 쉬운 구현은 모든 잠재적 요소에 대한 키 값의 정렬되지 않은 배열

-These values are set to ∞ initially

-**insert or decreasekey** is fast because just adjusting a value → O(1)

-**deletemin** requires a linear-time scan of the list

4.5.2 Binary heap (이진 힙)

Complete binary tree(완전한 이진트리)

- Each level filled in from left to right, and must be full before next level is started
- Key value of any node is less than or equal to that of its children
  So, root is the smallest

Insert : Place the new element at the bottom, and let it bubble up(if smaller than parent, swap the two and repeat)

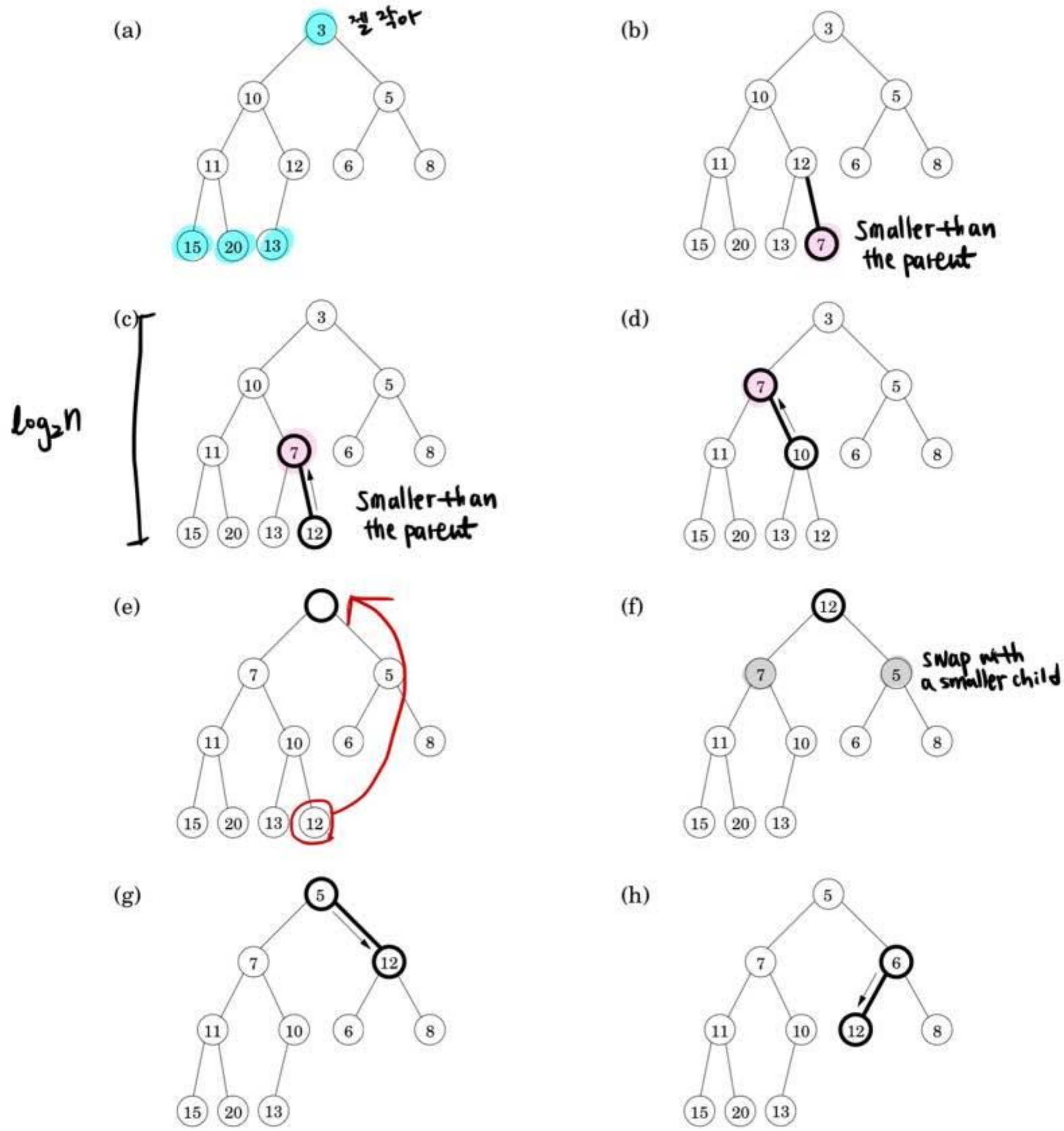
[log2n]: # of swaps at most the height of the tree

Decreasekey: similar but element already in the tree

Deletemin: return the root value, take the last node and place it at the root, sift down(if bigger than either children, swap with smaller one and repeat)

O(logn) time

Figure 4.11 (a) A binary heap with 10 elements. Only the key values are shown. (b)–(d) The intermediate "bubble-up" steps in inserting an element with key 7. (e)–(g) The "sift-down" steps in a delete-min operation.



-The regularity of a complete binary tree: Has a natural ordering

4.5.3 d-ary heap

-Identical to a binary heap except that nodes have d children instead of two

-Reduces the height of a tree with n elements to O(logdn)

-Inserts: O(logd)

-Deletemin: O(dlogdn)

B/c we have to find the minimum child to promote, whereas up-heaps just compare with the parent

4.6 Shortest paths in the presence of negative edges

4.6.1 Negative edges

-Dejkstra's algorithm doesn't work when **edge lengths can be negative**

-Keep in mind that **dist values** are either overestimates or exactly correct

Start off at ∞ and change by updating along an edge:

$$\underline{\texttt{procedure update}}\texttt{(}(u,v) \in E\texttt{)}$$
$$\texttt{dist}(v) = \min\{\texttt{dist}(v), \texttt{dist}(u) + l(u,v)\}$$

-Update operation: an expression that the distance to v cannot possibly be more than the distance to u, plus l(u, v)

Properties: 1. It gives correct distance to v in the particular case where u is the second-last node in the shortest path to v, and dist(u) is correctly set

2. It will never make dist(v) too small, and in this sense it is safe.

-Dijkstra's algorithm: simply as a sequence of update's

-If we don't know all the shortest paths beforehand, how can we be sure to update the right edges in the right order?

Solution: Simply update **all** of the edges |V|-1 times

The resulting O(|V|*|E|) procedure is called the **Bellman-Ford algorithm**

Figure 4.13 The Bellman-Ford algorithm for single-source shortest paths in general graphs.

```
procedure shortest-paths(G, l, s)
Input:    Directed graph G = (V, E);
          edge lengths {l_e : e ∈ E} with no negative cycles;
          vertex s ∈ V
Output:   For all vertices u reachable from s, dist(u) is set
          to the distance from s to u.

for all u ∈ V:
   dist(u) = ∞
   prev(u) = nil

dist(s) = 0
repeat |V| − 1 times:
   for all e ∈ E:
      update(e)
```

4.6.2 Negative cycles

-The **shortest-path problem** is **ill-posed**(잘 정의되지 않는) in graphs with negative cycles

-*Negative cycles allow us to* ***endlessly apply rounds of update operations, reducing dist estimates every time***

(가중치의 합이 음수인 사이클이 존재하게 되면 최단 경로가 음의 무한대로 발산하게 된다는 것)

4.7 Shortest paths in dags

-Two subclasses that **exclude** possibility of **negative cycles**: 1. Graphs w/o negative edges, 2. Graphs without cycles

-See how the single-source shortest-path problem can be solved in linear time on dags

Need to perform a sequence of updates that includes every shortest path as a subsequence

Key source of efficiency: *In any path of a dag, the vertices appear in increasing linearized order*

Therefore, it is enough to linearize the dag by DFS, then visit the vertices in sorted order, updating the edges out of each

Figure 4.15 A single-source shortest-path algorithm for directed acyclic graphs.

```
procedure dag-shortest-paths(G, l, s)
Input:     Dag G = (V, E);
           edge lengths {l_e : e ∈ E}; vertex s ∈ V
Output:    For all vertices u reachable from s, dist(u) is set
           to the distance from s to u.

for all u ∈ V:
   dist(u) = ∞
   prev(u) = nil

dist(s) = 0
Linearize G
for each u ∈ V, in linearized order:
   for all edges (u, v) ∈ E:
      update(u, v)
```

-Doesn't require edges to be positive. In particular, can find longest paths in a dag by negating all edge lengths