# CMPUT 275 Wi17 - INTRO TO TANGIBLE COMPUT II Combined LBL Wi17

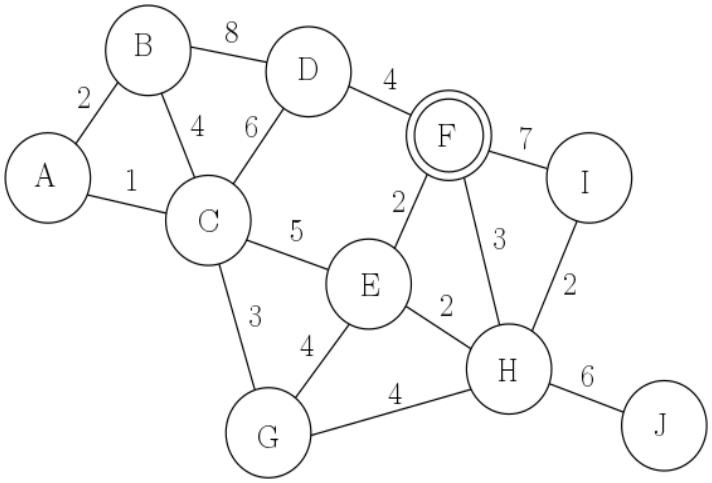
## Towards Min-Heaps

### Dijsktra: Trace it by Hand!

Recall Dijkstra's algorithm: In a weighted graph, Dijkstra's algorithm will find the shortest path from some node to some (or all) other node(s).

Our mental model for Dijkstra's algorithm was to start runners running with equal speed from the start node (the weights determine the distances between the nodes) and let them spawn new runners when they reach any node. The time when the first runner reaches a node is the distance of the node from the start node.

Activity: Trace how Dijkstra runs on the following graph:




Keep track of the set of runners, the reached structure. The question we answered: What is the 4th runner extracted from the set of runners in the main loop of Dijkstra?

### Minimum Heaps

Dijsktra, the way it was implemented runs in O(*m**2*) time where *m* is the number of edges in the graph: Eventually, there will be a runner passing along any edge in the graph, so the outer loop runs *m* times. Extracting the minimum with the current implementation runs in time linear in the number of runners, which can be again as large as *m*, giving rise to the *m*m* total running time (adding to the set of runners is essentially O(1)).

Can we speed this up? We cannot avoid having m runners in the worst-case. So the only potential is saving time on extracting the runner with the smallest destination time (the "min"-extraction).

Can we speed this up by using a different strategy of keeping track of the runners?

What do we need?

- We need to be able to access the minimum and extract it fast (efficient `pop_min`)
- We need to be able to add new elements efficiently (efficient `add`).

If we can do both in significantly less time than O(*m*) (m is the number of runners now) then we will win.

Any data structure that supports the abstract requirement of being able to add and extract the minimum from a list of ordered values is called a **priority queue** (priority queues are widely used and are generally quite useful). A priority queue is an abstract data structure (introduced here). So we want to implement a priority queue. How? Dictionaries, sets, etc. are two expensive.

First idea: Finding the minimum was causing the bottleneck. Let's make sure that we know where the minimum is, so we don't need to search for it.

How can we do this? What do we need? When we extract the minimum, we should be able to find the minimum without expensive searches. But wait, this is similar to what the problem we started with was, just for now this is for the set after the minimum is removed? How to solve this impass? Second idea: Divide and conquer! Let's divide the rest of the set into two (roughly) equal parts and assume we have the minimums there readily (because we use the same magic, yet to be discovered data structure). If we do this, we can just get the minimum for the remaining elements by choosing the smaller of these two minimums. Once this is found, we can put this to the special position where the minimum of the whole set should always be. Now the "hole" is in the subset where we removed the minimum from. But then we can recurse. This in turns leads to a recursive structure, which is also equivalent to a binary tree.

In other words, we will use a binary tree (trees and other CS terms are discussed below) and make sure that the following holds:

- For each subtree in the tree, the root of the subtree has the smallest value of all values stored in the subtree.

A binary tree with this property is said to have the **min-heap property**.

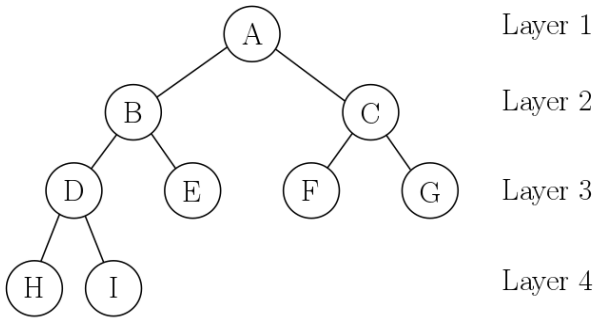
There are multiple ways to make sure that the min-heap property is maintained by `pop_min` and `add`. The general idea is that the straightforward implementation of both of these operations will temporarily disrupt the min-heap property and then we need to work some more to restore it. This is done by traversing the tree from root to leaf (after min_extract) or leaf to root (after add) to restore the min-heap property ("trickling" down, or up a value that is "out of place"). Now, what will be running of these operations? In the worst case, the running time can be proportional to the length of the longest path from the root to some of the leaf nodes (the length of the longest path from root to leaf is called the **height** of the tree). So we should keep the tree height small. For this reason we require the following property:

- The tree is most compact, i.e., the height of the tree cannot be reduced while keeping the same number of nodes.

We will call trees with this property **near complete binary trees** and the property will be called the "**shape property**" when discussing minimum heaps. You can prove that a binary tree is near complete if and only if every leaf node is at distance *h* or *h-1* from the root where *h* is the height of the tree. We need extra care when implementing our two operations to make sure that both the shape and the min-heap properties are maintained.

# Implementation of Minimum Heaps

How to implement minimum heaps, i.e., how to store the data? One idea is to use dictionaries. One idea is to use the graph class. Both are overkill. A better idea is to introduce a new class to represent a node which points to its parent and children (if any). However, we can also simply use an array, exploiting that we only need to deal with near complete binary trees. The idea is shown on the next figure:




| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Node | A | B | C | D | E | F | G | H | I | | | | | | |
| Layer | 1 | 2 | | 3 | | | | 4 | | | | | | | |

The array (middle in the bottom) stores the nodes consecutively in a left-to-right order: That is, for this implementation we impose an ordering on the children of any node.

# Graph Theory Concepts

We learned about the following graph theory concepts:

- Tree: Undirected graph where for any two distinct nodes of the graph there is a unique path between them. Empty graph is also a tree. (Recall the notion of path, walk, loop, cycle.)
- Forest: Undirected graph where where for any two distinct nodes of the graph there is at most one path connecting them. (Recall connected components.)
- Rooted tree: Tree with a special node, designated as the so-called "root". The height of a node in a rooted tree is its distance to the root. A node is a child of another node if its a neighbor of the node and its height is one plus the height of the other node. The nodes having the same height form a layer or level. If *c* is the child of *n*, we also say that *n* is the parent of *c*. Any node can have multiple children, but only one parent. We can also talk about extended family relationships, like grandparents or grandchildren of a node, etc. A node that does not have any children is called a leaf. A subtree, informally, is a node and all its descendants. More formally, a subtree in a tree is the tree composed of a node and all its descendants together with the edges between them.

Last modified: Thursday, 28 January 2016, 8:05 AM