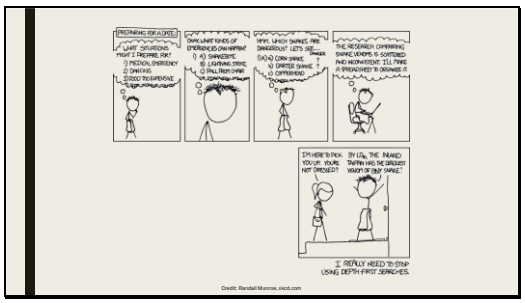
Slide 1



Slide 2

CS 2112

Recitation 10
Heaps + Dijkstra's Algorithm

November 5 / 6, 2024

Slide 3

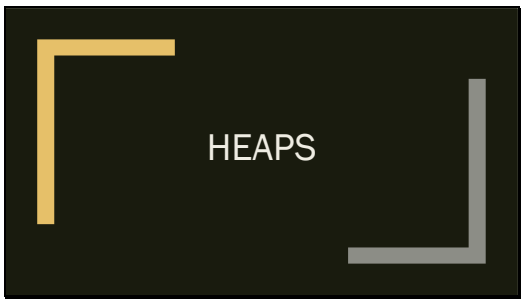
Agenda

- Heaps Review
- Dijkstra's Algorithm

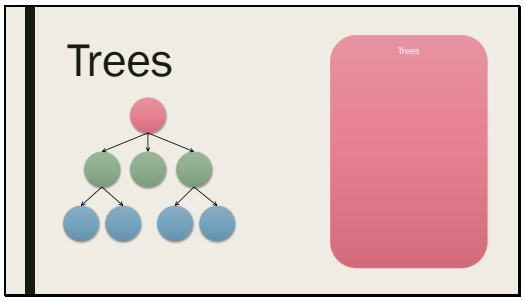
Reminders

- A5 Design Doc Due Wednesday

Slide 4

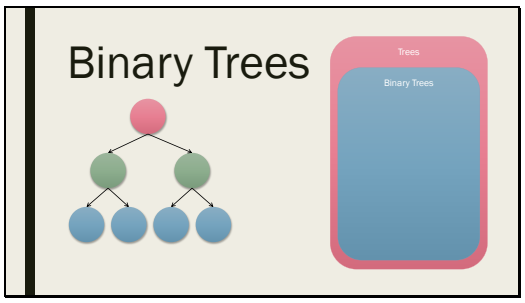


Slide 5



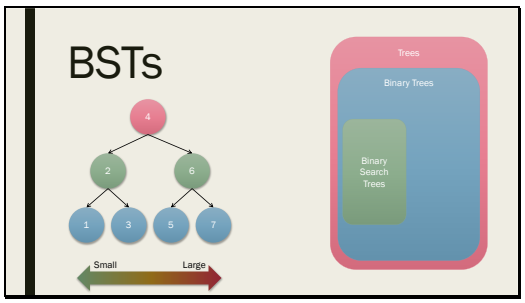
As a quick review, trees are data structures consisting of data in nodes and pointers to more nodes, called “children”

Slide 6



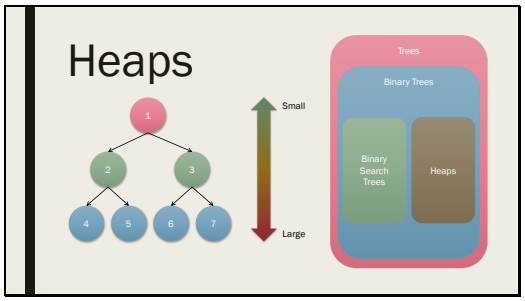
A binary tree is a specific type of tree where each node only has at most 2 children

Slide 7



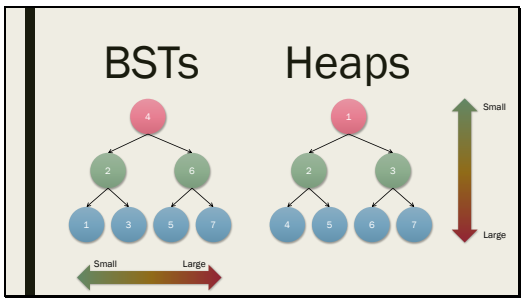
A binary **search** tree is a further specialization that enforces the invariant that all children to the left of a node are smaller, and all children to the right are bigger (according to some sort order)

Slide 8



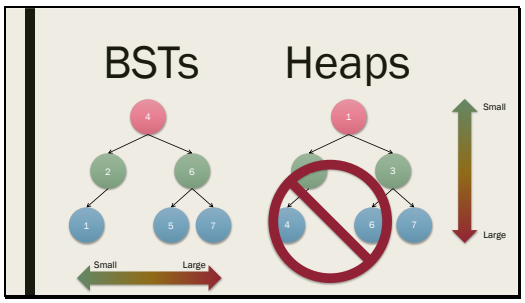
A heap, on the other hand, is a different type of binary tree, and is not a search tree. The heap invariant requires the a node be smaller than all its children (for a min-heap; a max-heap is the exact opposite).

Slide 9



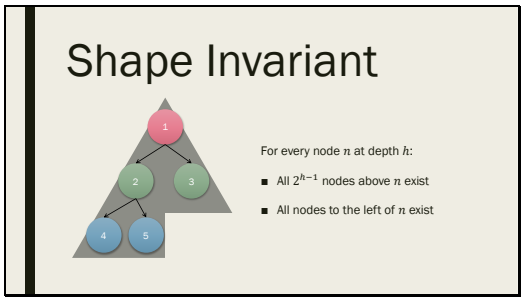
Note that heaps have this up-down relationship as opposed to a BST's left-right ordering

Slide 10



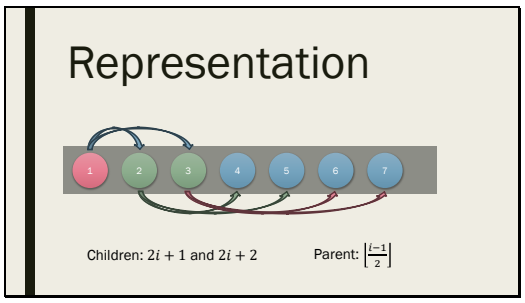
Also unlike a BST, a heap cannot be missing arbitrary children

Slide 11



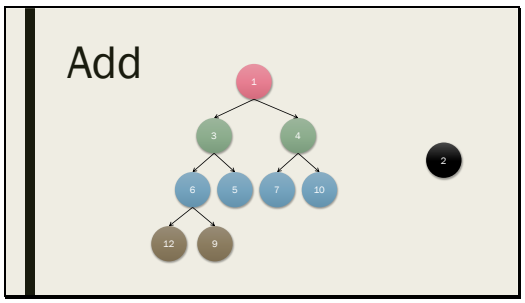
A heap must fulfill this shape invariant, which bounds which children can be missing

Slide 12



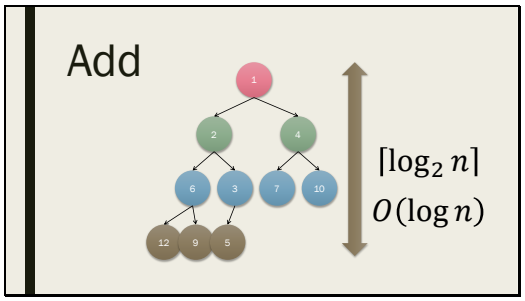
As a result, a heap can be compactly stored in an array instead of with pointers to arbitrary objects in memory.
Note that all we have changed is where the nodes are stored in memory (consecutively, instead of in random places). We still are modelling this as a tree, with parent and children nodes.
The convenient part is that we no longer need to explicitly store the pointers, because we can just compute the index of where the children and parents are.

Slide 13



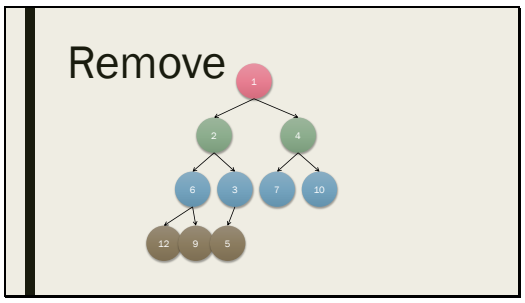
Practice: How would you add this 2 node to the heap?

Slide 14



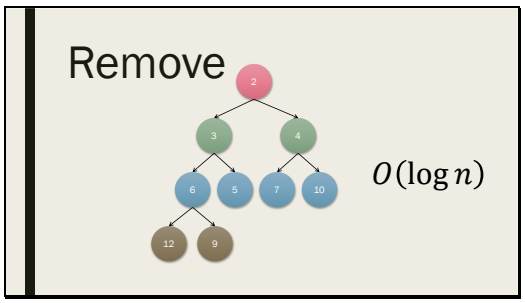
The time complexity of add is O(log n) because it's proportional to the height of the tree.

Slide 15



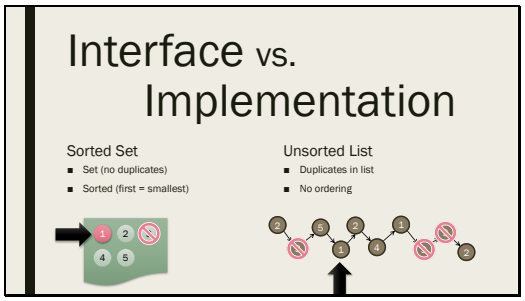
Practice: Now how would you remove the smallest node from this heap?

Slide 16



Like add, remove is also O(log n)

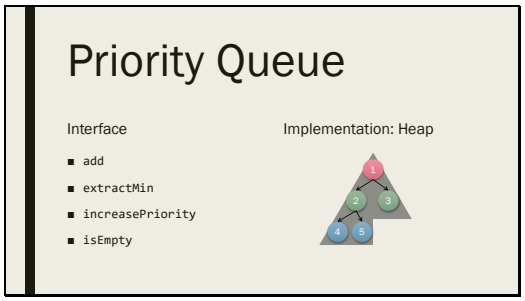
Slide 17



Switching gears a bit, remember there is a difference between an interface, which is how users will interact with your code, and the implementation, which is how your code works behind the scenes.

To take A3's example, a sorted set as an interface gives users the illusion that there's no duplicates and the first element must be the smallest one, but the implementation might be an unordered linked list, where you have to scan the whole list to find the minimum and have to remember to remove all duplicates when removing an element.

Slide 18

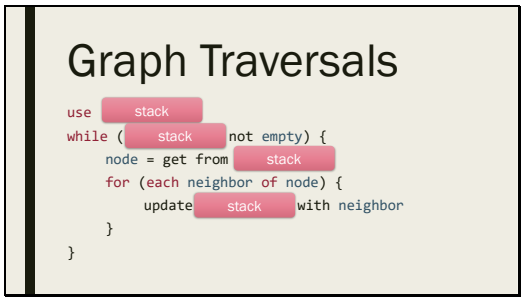


In that vein, a priority queue is an *interface* that supports adding and removing elements based on some priority, as well as perhaps changing the priority.

A heap is a particularly efficient *implementation* of that interface. (Note a BST has the same asymptotic complexity; the heap is preferable only because of better constant factors)

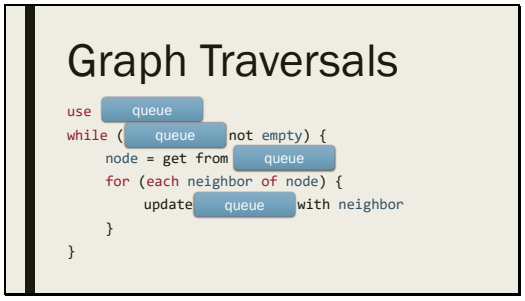
Slide 19

Slide 20



You've probably seen this graph traversal algorithm, where you push neighbors to a stack and pop from the stack each cycle.
This is a DFS (depth first search).

Slide 21



Notice that if you replace the stack with a queue, you end up with a BFS (breadth first search), but the algorithm itself is the same.

Slide 22

Graph Traversals

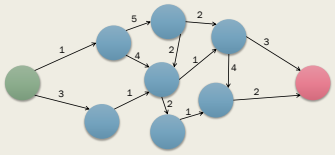
```
use priority queue*
while (priority queue not empty) {
    node = get from priority queue
    for (each neighbor of node) {
        update priority queue with neighbor
    }
}
```

* using shortest found distance to node as priority

The core insight is that if we replace the data structure instead with a priority queue (using the shortest known path to a node as its priority), then we get Dijkstra's algorithm. That's it.

Slide 23

Shortest Path

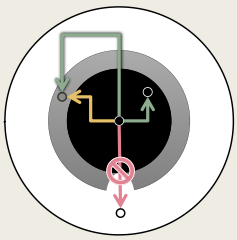


Dijkstra's algorithm, to remind you, finds the shortest path between two nodes in a directed graph with non-negative edges.
(technically Dijkstra's can find the distance to *every* other node, not just the target one).

Slide 24

Invariant

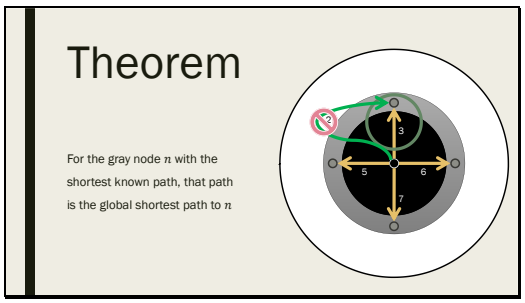
1. For any black node n, the shortest path to n is known and only uses black nodes
2. For any gray node n, the shortest path to n through black nodes is known.
3. All edges from black nodes go to black or gray nodes.



This loop invariant formalization and proof for Dijkstra's algorithm comes from Prof. David Gries of CS 2110

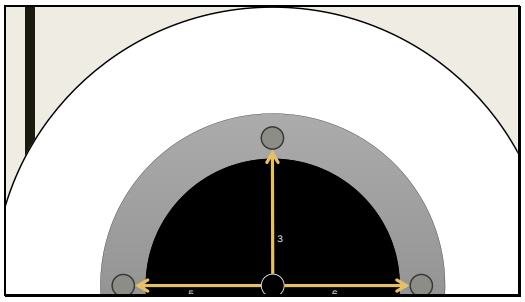
Here is a loop invariant for Dijkstra's algorithm. Note this particular formalization comes from Prof. David Gries and CS 2110, and differs slightly from the one shown in class. I find this one more intuitive so I wanted you to see it.
Note that on point 2, it states we know the shortest path through other black nodes to any gray node n, but there may be even shorter paths outside the black nodes we haven't seen.

Slide 25



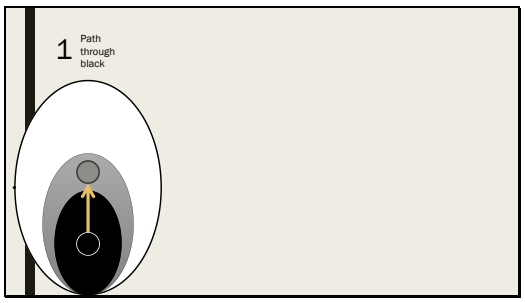
The key to understanding why the algorithm works is this theorem.

Slide 26

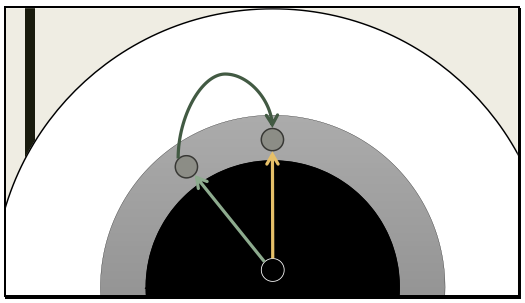


Let's imagine what the shortest path to a gray node might look like. In the first scenario, the shortest path just goes through all the black nodes we've seen.

Slide 27

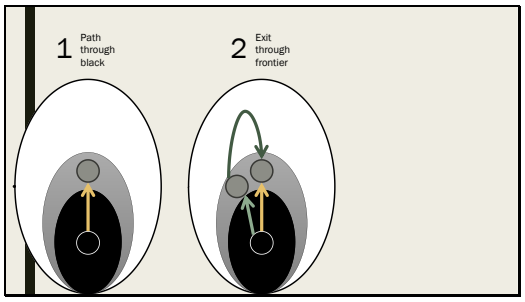


Slide 28

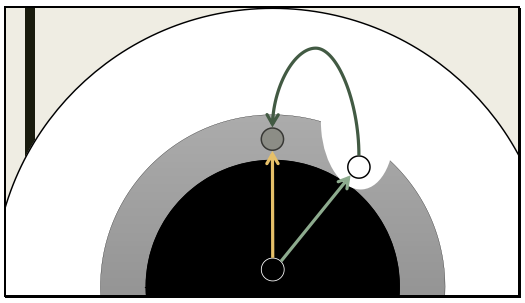


In the second scenario, the shortest path travels through a different gray node first, and then perhaps takes some shortcut outside to the target node.

Slide 29

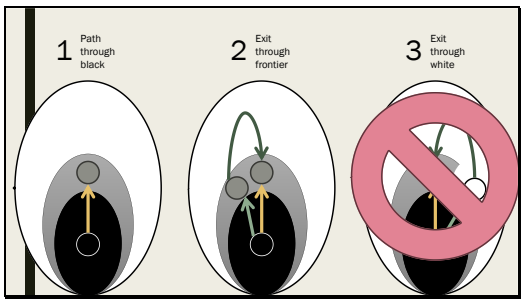


Slide 30



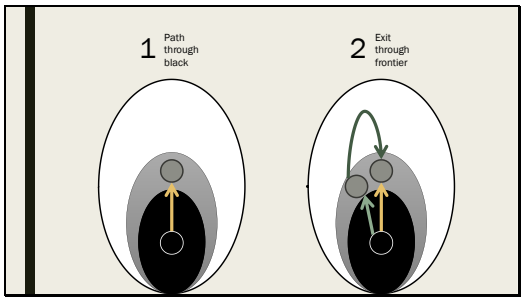
In the third scenario, the shortest path doesn't touch another gray node at all and takes a shortcut outside directly.

Slide 31



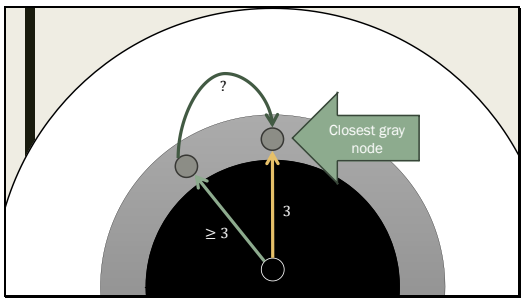
However, the third point of the invariant guarantees that scenario 3 is not possible, since any edge leaving a black node cannot go straight to a white node.

Slide 32



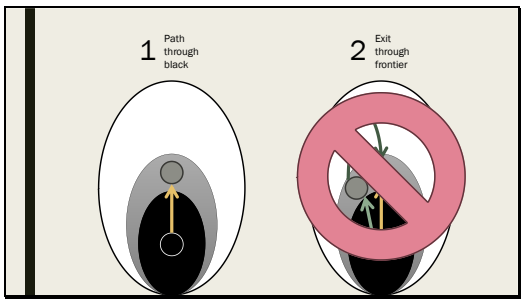
So there's only two possibilities.

Slide 33



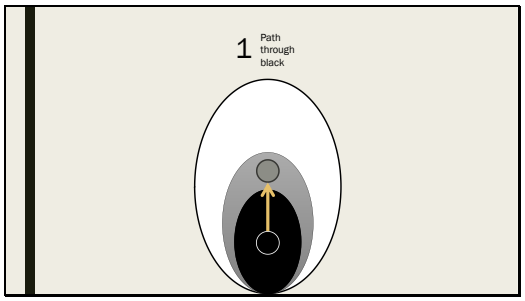
Zooming back in on scenario 2, notice that the theorem explicitly looks at the short path to any gray node. Which means that a path to a different gray node must be **longer** than the original path to the target gray node. So it doesn't matter how fast the shortcut outside the frontier is, because the path to get to the shortcut must be longer.

Slide 34



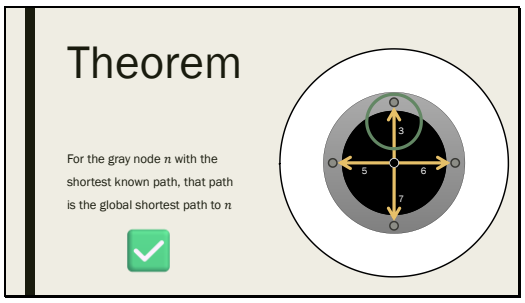
Therefore scenario 2 is not possible.

Slide 35



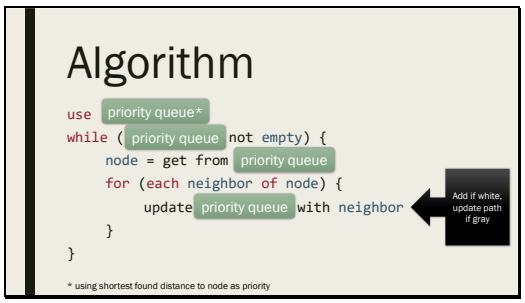
This means the shortest path to this gray node must be through the black nodes. But the invariant guarantees we know the shortest path through the black nodes to any gray node.

Slide 36



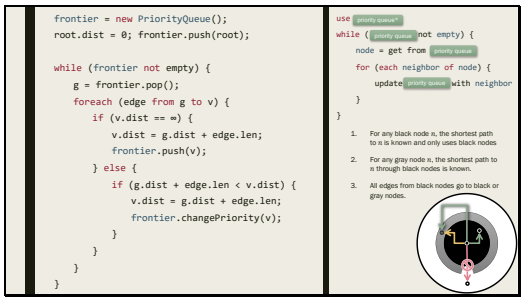
Therefore, we know the shortest path to this node.

Slide 37



It's for this reason that, if our priority queue models our frontier, it is safe to pop the node with the shortest known path off the frontier each step.

Slide 38



Using our invariant, we can construct the full logic of the algorithm. Any node that has infinite distance is white, anything in the priority queue is gray, and any node removed from the queue is black.

For initialization, the only node whose distance we know is the starting node, so we set its distance to zero and add it to the frontier.

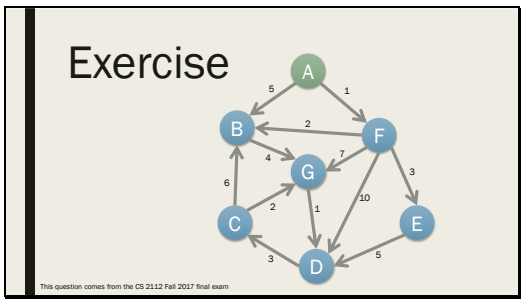
The postcondition is that we've found the shortest path to every node, which happens when all reachable nodes turn black – thus our loop continues while there's still nodes to process.

The theorem tells us we can make progress towards termination by popping the shortest distance node in the frontier.

To preserve the invariant, we must color any white neighbors of the newly blackened node to be gray (by computing its only known path distance and adding them to the frontier). Also, if we find a new shorter path to a neighbor, we have to update it too.

And that's it!

Slide 39



Exercise: compute the state of the priority queue at each iteration of Dijkstra's algorithm starting at node A. This was a question from my final exam.

Slide 40