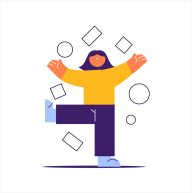
Lec 27: Heaps and Priority Queues

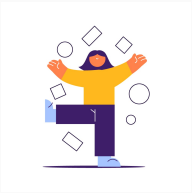
April 29, 2026

Admin



- Lab today! (Take-home lab.)
- Please come to lab even if you've finished to check in.
- Any questions?

Info for Final



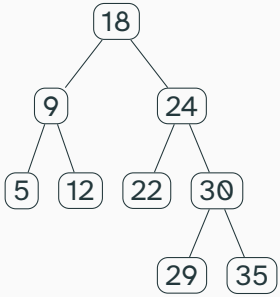
- Final: Monday May 25th at 9:30AM
- Alternate time: Tuesday May 19th at 9AM
- You *must* attend one of these times!
- If it is possible that you will miss May 25th, you have to take it on May 19th
- Please email me in the next week if you are taking the final on May 19th.

Implementing Tree Iterators

Implementing Tree Iterators

- Goal: implement the traversals we mentioned as an iterator
- Can do `next()` and `hasNext()` on demand
- Problem: want to get values on demand (should be updated as the tree is updated)
  - Don't want to traverse the tree, store all tree values, and then dispense them one by one (you may do that in the lab however)
  - Instead: each call to `next()` should go to the next node in the tree we want to output
- Challenge: implementing a recursive traversal piece-by-piece
- To think about: what data structure helps with recursion?

Pre-order traversal

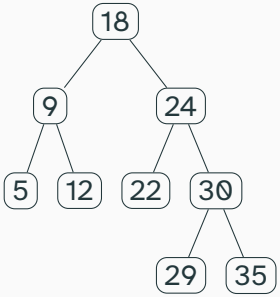


- Visits the node, then recursively traverses the left child, then the right child
- Keep track of the current node we're traversing
- What happens when we hit a leaf?
- Could backtrack by following pointers; might get confusing
- Instead: maintain nodes to visit on a stack!

Pre-order traversal

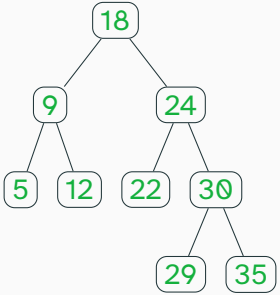
- Stack maintains the non-empty `BinaryTree<E>` objects that we *still need to traverse*
- So `next()`:
  - pops the top item off the stack
  - Stores its value to be returned
  - Pushes its right child onto the stack if nonempty
  - Pushes its left child onto the stack if nonempty
- `hasNext()`?
  - Just returns if the stack is empty
- Let's look at the code

Post-order and In-order traversal



- We won’t go over these in this class. In short: a stack works, but you need to use it a little differently.

Level-order Traversal



Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.

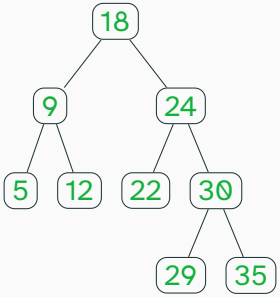
Level-order traversal

- Level-order traversal is not recursive!
- How do we keep track of what nodes to visit next?
- Key insight: the order we visit nodes at a given “level” is the same order we visited their parents
- So the *first* parents to be visited have the *first* children that are visited
- …Can we use a queue?

Level-order iterator

- To begin: push root onto the queue
- `next()`:
  - Dequeue node off the queue; store its value to be returned
  - Enqueue its non-empty children onto the queue
- `hasNext()`: return if queue is empty
- Let’s look at the code

Level-order Traversal



Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Queue is labelled with the *values* of the nodes, but in reality the objects stored are of type `BinaryTree`

Queue: 9 24 Queue: 24 5 12 Queue: 5 12 22 30 Queue: 12 22 30 Queue: 22 30 Queue: 30 Queue: 29 35 Queue: 35 Queue: Queue:

Priority Queues

Priority Queues

- **Priority queue:** A data structure that can store a set of items, each with some *priority*, with the following operations:
  - **Insert**(i, p)**:** Insert a new item i with priority p into the priority queue
  - **RemoveMin**$()$**:** Remove (and return) the item i with the smallest priority in the queue
- How long do these operations take in an AVL tree? $O(\log n)$
- Today, we'll see a different, much simpler way to obtain $O(\log n)$ worst case. It's also faster in practice

Motivating Priority Queues

- Handle a queue with “priorities:” first in is not necessarily first out
    - Real-life examples: airplane boarding; hospital triage
- Scheduling different processes on a CPU
- Many algorithmic applications
    - How can we sort in $O(n \log n)$ time if **Insert()** and **RemoveMin()** are $O(\log n)$ time?

Heap

- Let's define a data structure using a tree
- Then: we'll significantly simplify it into an array data structure
- This data structure is called a *heap*
- Invariant: Each element in the heap is smaller than its children
  - This implies: each element in the heap is smaller than all of its descendants
- Invariant 2: The heap is a *complete* binary tree: all levels but the last are "full"; last is filled left to right
- Let's draw a heap [Blackboard]
- Does *not* necessarily satisfy Binary Search Tree property (left child is less than right child)

Inserting into a Heap

In pairs: How can we insert into a heap to maintain these invariants?

- Where does the new item go?
  - Last level must be filled in left to right; let's put it there
- How can we ensure that this satisfies the heap property?
  - Swap with parent until the heap property is satisfied
  - Called "sift up"
- Why does this work? [Blackboard]

Inserting into a Heap: Analysis

- What is the height of a complete binary tree with n nodes?
  - $O(\log n)$
- Each swap takes $O(1)$ time, so insert takes $O(\log n)$ time

Removing the Minimum Element of a Heap

- How can we remove the minimum element?
  - It would be a lot nicer to remove the last element of the last level
  - Idea: Swap the root and the last element. Then we can safely remove it. Then, we'll "sift down" to preserve the heap property [Blackboard]
- To sift down: swap the element with its *smaller* child (why?). Repeat until heap property is maintained
- Also $O(\log n)$

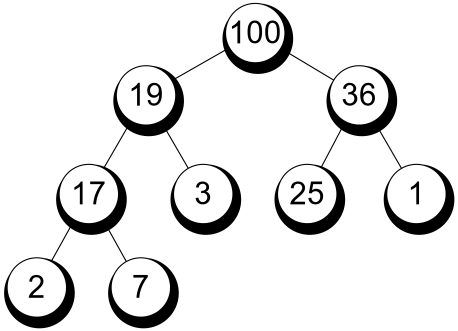
Heap as array

- Observation: The shape of our tree is super restricted. Can we store it in an easier way?
- Let's number the nodes starting at 0, in level order
- If a node has number i, what numbers are its children?
  - $2i + 1$ and $2i + 2$
- If a node has number i, what number is its parent?
  - $\lfloor (i - 1)/2 \rfloor$
- (Can prove by induction; just take my word for it for now.)
- Let's throw out the tree and do a heap operation using just the array! [Blackboard]

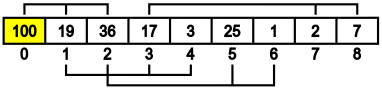
A heap is *only an ArrayList.* It has no nodes, no pointers, no extra information.

Priority Queue

Tree representation



Array representation



- Insert a new item (Insert)
- Remove minimum weight item (ExtractMin)
- Done using a heap
- *Extremely* efficient; used extensively in practice
- $O(\log n)$ time to insert or remove minimum item

Heaps (Reference)

- Heap property: each item in the tree is smaller than either of its children
- Tree has minimum height; filled in left to right (“full” tree)
- Maintain implicitly in an array, *not* as a tree (do not need pointers!)
- Extract min, or insert a new item, in $O(\log n)$ time

Heapify

How do we build a heap?

- Seems easy: call **Insert()** on each item one at a time, takes $O(n \log n)$ total time
    - This was the method used when heaps were originally invented
- But: it's possible to build a heap in $O(n)$ total time!
- This subroutine is called Heapfiy

Heapify: Idea

- Idea: go *bottom-up* through the heap, making sure that taller and taller subtrees satisfy the heap property
- How do we make sure the leaves satisfy the heap property (they are smaller than their children)?
    - Don’t need to do anything
- What about the parents of the leaves?
    - Sift down

Heapify

- Go through all items of the heap bottom up (from back to front in the array). Call sift down on all of them.
- Running time?
- The time to call sift-down on a node is the height of its subtree (the longest path to any leaf)
  - $n/2$ leaves have 0 sifts
  - $n/4$ nodes have ≤ 1 sift
  - $n/8$ nodes have ≤ 2 sifts
  - $n/16$ nodes have ≤ 3 sifts
- With a little math, the total sifts is $< 2n$.