Lab 6. Level count

Each tree object can count nodes at all levels

levelCount(0) returns 1
Lab 6. Level count

levelCount(1) returns:
levelCount(0) of leftChild + levelCount(0) of right child
Lab 6. Level count

levelCount(2) returns:
levelCount(1) of leftChild + levelCount(1) of right child
Is this tree balanced?

A. Yes
B. No
We insert node 5

Is the tree still balanced?

A. Yes
B. No
Lab 7. AVL trees

Which node is unbalanced (needs rotation)?

A. Node 6
B. Node 4
C. Node 2
D. Node 7
E. None of the above
The rebalancing algorithm in the lab is similar to the algorithm we learned in class:

Going up from the point of insertion, you find the first unbalanced node and from it you collect its child and its grandchild, moving always into the child with the larger height:

$4 \rightarrow 6 \rightarrow 5$

$x \rightarrow y \rightarrow z$
Lab 7. AVL trees

4 → 6 → 5
x → y → z

Now you need to consider 4 different cases:

Case 1: right-left heavy
Determine order for this case:
a=x, b=z, c=y, a<b<c

Collect child subtrees to be reattached:
t_0 = x.left, t_1= z.left, t_2=z.right, t_3=y.right
Restructuring is implemented: you only need to consider all the cases and identify 3 nodes and 4 children

\[ a=x, \quad b=z, \quad c=y, \quad a<b<c \]

\[ t_0 = x.\text{left}, \quad t_1 = z.\text{left}, \quad t_2 = z.\text{right}, \quad t_3 = y.\text{right} \]
Lab 7. AVL trees

Tree after restructuring:
A. 
B. 
C. None of the above
Priority Queue ADT
Binary heaps

Lecture 22
by Marina Barsky
Priority Queue ADT

➢ A **Priority Queue** is a generalization of a *Queue* where each element is assigned a *priority* and elements come out in order of priority

➢ If the priority is the earliest time they were added to the queue then Priority Queue becomes a regular FIFO Queue

➢ We are interested in a case when priority of each element is not related to the time when the element was added to the queue
Priority Queue ADT

Specification

**Priority Queue** is an Abstract Data Type supporting the following main operations:

- `top()` - get an element with the highest priority
- `enqueue(e, p)`* - adds a new element `e` with priority `p`
- `dequeue()` - removes and returns the element with the highest priority

*To simplify the discussion we use `enqueue(p)`, where `p` is a number which reflects the priority*
## Priority Queue: possible Data Structures

<table>
<thead>
<tr>
<th></th>
<th>enqueue</th>
<th>dequeue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted array/list</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Sorted array/list</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
Binary max-heap

Definition

Binary max-heap is a binary tree where the value of each node is at least (≥) the values of its children.

https://visualgo.net/en/heap?slide=1
Heap?
Heap? Yes
Heap?

42

29
14
19

18
7
12
25
7
Heap? No
Heap operations: \textit{top}

return the root value

Run-time: O(1)
Heap operations: **enqueue(e)**

attach a new node to any leaf
Heap operations: enqueue (e)

the heap property may become violated
Heap operations: enqueue \((e)\)

to fix that we let the new node \(sift\ up\)
Heap operations: \textit{sift\_up}(e)

if current element is bigger than the parent: 
\textit{swap}
Heap operations: \texttt{sift\_up(e)}

if current element is bigger than the parent:  
\texttt{swap}
Heap operations: \textit{sift\_up(e)}

if current element is bigger than the parent: \textit{swap}
Heap operations: \( \text{sift\_up}(e) \)

if current element is bigger than the parent: 
\( \text{swap} \)
Heap operations: \textit{sift}\_up(e)

this works because the heap property is violated only on a single edge at a time
Heap operations: \texttt{sift\_up(e)}

if current element is bigger than the parent: 
\textit{swap}
Heap operations: \textit{sift\_up(e)}

if current element is bigger than the parent: \textit{swap}
Heap operations: \textit{sift\_up}(e)

heap property is restored
Heap operations: \textit{enqueue} \((e)\)

running time of \textit{enqueue} depends on how many times we need to \textit{swap}
Heap operations: \textit{enqueue (e)}

with each swap, the problematic node moves one node closer to the root

running time: $O(\text{tree height})$
Heap operations: dequeue

remove and return the root value
Heap operations: dequeue

*remove* the root value
Heap operations: dequeue

replace the empty node value with any leaf node value and remove the leaf
Heap operations: dequeue

replace the empty node value with any leaf node value and remove the leaf
Heap operations: dequeue

again, this may violate the heap property
Heap operations: dequeue

to fix it we let the problematic node *sift down*
Heap operations: \( \text{sift\_down}(e) \)

if current node is smaller than one of its children, swap it with the largest child
Heap operations: \textit{sift\_down}(e)

swapping with the largest child automatically restores both broken edges
Heap operations: *sift_down*(e)

swapping with the largest child automatically restores both broken edges
Heap operations: *sift_down(e)*

if current node is smaller than one of its children, swap it with the largest child
Heap operations: $\textit{sift\_down}(e)$

if current node is smaller than one of its children, swap it with the largest child
Heap operations: \texttt{sift\_down(e)}

the heap property is restored
Suppose you have a Binary Search Tree. Is it also a heap?

A. Yes

B. No

C. Sometimes
Suppose you have a binary heap. Is it also a binary search tree?

A. Yes
B. No
C. Sometimes
Heap operations: enqueue and dequeue

Running time depends on how many times the swap is performed to restore the heap

running time: $O(\text{tree height})$
We want a tree with the min possible height

How to Keep a Tree Shallow?

**Definition**

A binary tree is *complete* if all its levels are at full capacity except possibly the last one which is filled from left to right.
Example: complete binary tree
Complete binary tree ?
Complete binary tree?
Complete binary tree?
Complete binary tree?
Advantage of Complete Binary Trees:
low height

**Theorem**

A complete binary tree with \( n \) total nodes has height at most \( O(\log n) \).
Proof

- Complete the last level of the tree if it is not full to get a **full** binary tree.

- This full tree has \( n' \geq n \) nodes and the same height \( h \).

- At level 0 we have \( 2^0 = 1 \) node, at the first level: \( 2^1 = 2 \) nodes, at level \( k \): \( 2^k \) nodes, and the total number of levels is \( h-1 \). Then the total number of nodes:

\[
n' = 1 + 2^1 + 2^2 + \ldots + 2^{h-1} = \frac{2^{(h-1)+1} - 1}{2-1} = 2^h - 1
\]

(sum of geom. series)

- Note that \( n' \leq 2n \), because the actual total number of nodes \( n \) is between \( 2^{h-2+1} - 1 + 1 = 2^{h-1} \) and \( 2^h - 1 \)

- Then \( n' = 2^h - 1 \) and hence:

\[
h = \log_2(n' + 1) \leq \log_2(2n + 1) = O(\log n).
\]
If we store Heap as a Complete Binary Tree:

- *Top* in time $O(1)$
- *Dequeue* in time $O(\log n)$
- *Enqueue* in time $O(\log n)$

As long as we keep the tree complete
How many of these structures represent a complete binary tree?

A. 0-1  
B. 2  
C. 3  
D. 4  
E. 5-8
A Complete Binary Tree can be stored in an Array
A Complete Binary Tree can be stored in an Array
A Complete Binary Tree can be stored in an Array
Tree operations in a heap array

parent(A[i]) = A[⌊(i-1)/2⌋]
Tree operations in a heap array

left_child(A[i]) = A[2i + 1]
Tree operations in a heap array

right_child(A[i]) = A[2i + 2]
Heap array: **enqueue (33)**

to add an element, insert it as a leaf in the **rightmost vacant position** in the last level (the last position of the array) and let it **sift up**
Heap array: \textit{enqueue} (33)

\texttt{enqueue(33)}

\texttt{parent(9) = 4}
\texttt{swap(A[9],A[4])}

\texttt{parent(i) = ⌊(i-1)/2⌋}
Heap array: \textit{enqueue} (33)

\begin{align*}
  \text{parent}(9) &= 4 \\
  \text{swap}(A[9], A[4]) \\
  \text{parent}(4) &= 1 \\
  \text{swap}(A[4], A[1])
\end{align*}

\[
\text{parent}(i) = \left\lfloor \frac{(i-1)}{2} \right\rfloor
\]
Heap array: **enqueue (33)**

parent(9) = 4
swap(A[9],A[4])

parent(4) = 1
swap(A[4],A[1])

parent(1) = 0  **OK**

stop
Heap array: dequeue()

Similarly, to extract the maximum value, replace the root by the last leaf and let it sift down
Binary **Min-Heap**

**Definition**

Binary **min**-heap is a binary tree where the value of each node is **at most** the values of its children.

Can be implemented similarly to max-heap
How many swaps will we do after we call dequeue() on this min-heap?

A. 0
B. 1
C. 2
D. 3
E. None of the above
If we insert 7 into this binary **min**-heap, how many swaps will we need to do?

A. 0  
B. 1  
C. 2  
D. 3  
E. None of the above
What is the array representation of the following min-heap tree?

A. \([8, 4, 9, 2, 5, 1, 6, 3, 7]\)
B. \([1, 2, 3, 4, 5, 6, 7, 8, 9]\)
C. \([1, 2, 4, 8, 9, 5, 3, 6, 7]\)
D. \([8, 9, 4, 5, 2, 6, 7, 3, 1]\)
E. Something else
Priority Queue ADT: possible Data Structures

<table>
<thead>
<tr>
<th></th>
<th>top</th>
<th>enqueue</th>
<th>dequeue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unssorted array/list</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Sorted array/list</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Balanced BST</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>Binary heap</td>
<td>O(1)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
</tbody>
</table>
Priority Queue with binary heap: notes

- Binary heap can be used to implement *Priority Queue* ADT

- Heap implementation is very efficient: all required update operations work in time $O(\log n)$

- Heap implementation as an array is also *space efficient*: we only store an array of priorities. Parent-child relationships are not stored, but are implied by the positions in the array
Common implementations of Priority Queues using Heaps

- C++: `priority_queue` in `std` library
- Java: `PriorityQueue` in `java.util` package
- Python: `heapq` (separate module)

Underneath is a **Dynamic Array**