Administrativia

Ranking (contest)
congrats!
Administrativia

Ranking (contest)
congrats!

PA2
don’t have a group yet? we’ll assign you one tomorrow
radiusSearch on kd-trees on your own (ask questions!)
come prepared for the interview on April 6th
15 bonus points on final if you create a client-server application using sockets!
Administrativia

Ranking (contest)
congrats!

PA2

don’t have a group yet? we’ll assign you one tomorrow
radiusSearch on kd-trees on your own (ask questions!)
come prepared for the interview on April 6th

15 bonus points on final if you create a client-server application using sockets!

Consider Linux
## Balanced BSTs

<table>
<thead>
<tr>
<th></th>
<th>sequential search (unordered sequence)</th>
<th>binary search (ordered sequence)</th>
<th>AVL</th>
</tr>
</thead>
<tbody>
<tr>
<td>search</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>insert</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>delete</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>min/max</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>floor/ceiling</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>rank</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(log n) **</td>
</tr>
</tbody>
</table>

** requires the use of ‘size’ at every node
How to remove data from an AVL tree?
Quiz

How to remove data from an AVL tree?

Can we sort using balanced trees? Cost?
Priority Queues
Queues

Enqueue

Dequeue
Priority Queues

Enqueue

Dequeue
Applications

Data Compression (huffman trees)
Process Scheduling (CPUs)
Graph Algorithms
Stream Data Algorithms
HPC Task Scheduling

Network Routing
Artificial Intelligence (search)

...
Priority Queues

Collections of <Key,Value> pairs

**keys** are objects on which an **order** is defined
Priority Queues

Collections of <Key,Value> pairs

keys are objects on which an order is defined

Every pair of keys must be comparable according to a total order:
Priority Queues

Collections of <Key,Value> pairs

**keys** are objects on which an **order** is defined

Every pair of keys must be comparable according to a **total order:**

- **Reflexive Property:** $k \leq k$
- **Antisymmetric Property:** if $k_1 \leq k_2$ and $k_2 \leq k_1$, then $k_1 = k_2$
- **Transitive Property:** if $k_1 \leq k_2$ and $k_2 \leq k_3$, then $k_1 \leq k_3$
Priority Queues

Queues

basic operations: enqueue, dequeue
always remove the item least recently added
Priority Queues

Queues
- basic operations: enqueue, dequeue
- always remove the item least recently added

Priority Queues (MaxPQ)
- basic operations: insert, removeMax
- always remove the item with highest (max) priority
Priority Queues

Queues

- basic operations: enqueue, dequeue
- always remove the item least recently added

Priority Queues (MaxPQ)

- basic operations: insert, removeMax
- always remove the item with highest (max) priority

Can also be implemented as a MinPQ
### Example (MinPQ)

<table>
<thead>
<tr>
<th>Method</th>
<th>Return Value</th>
<th>Priority Queue Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert(5, A)</td>
<td></td>
<td><code>{ (5, A) }</code></td>
</tr>
<tr>
<td>insert(9, C)</td>
<td></td>
<td><code>{ (5, A), (9, C) }</code></td>
</tr>
<tr>
<td>insert(3, B)</td>
<td>(3, B)</td>
<td><code>{ (3, B), (5, A), (9, C) }</code></td>
</tr>
<tr>
<td>min()</td>
<td>(3, B)</td>
<td><code>{ (3, B), (5, A), (9, C) }</code></td>
</tr>
<tr>
<td>removeMin()</td>
<td>(5, A)</td>
<td><code>{ (3, B), (5, A), (9, C) }</code></td>
</tr>
<tr>
<td>insert(7, D)</td>
<td>(7, D)</td>
<td><code>{ (5, A), (7, D), (9, C) }</code></td>
</tr>
<tr>
<td>removeMin()</td>
<td>(9, C)</td>
<td><code>{ (5, A), (7, D), (9, C) }</code></td>
</tr>
<tr>
<td>removeMin()</td>
<td>null</td>
<td><code>{ (7, D), (9, C) }</code></td>
</tr>
<tr>
<td>removeMin()</td>
<td>true</td>
<td><code>{ (9, C) }</code></td>
</tr>
<tr>
<td>isEmpty()</td>
<td></td>
<td><code>{ }</code></td>
</tr>
</tbody>
</table>

From Algorithm Design and Applications, Goodrich & Tamassia
<table>
<thead>
<tr>
<th></th>
<th>Sorted Array/List</th>
<th>Unsorted Array/List</th>
<th>AVL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>insert</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>removeMax</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>max</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Performance

<table>
<thead>
<tr>
<th></th>
<th>Sorted Array/List</th>
<th>Unsorted Array/List</th>
<th>AVL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>insert</strong></td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(log n)</td>
</tr>
<tr>
<td><strong>removeMax</strong></td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td><strong>max</strong></td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(log n)</td>
</tr>
</tbody>
</table>
what does this function do?

```cpp
void foo(int *array, int n) {
    std::priority_queue<int> pq;

    for (int i = 0; i < n; i++)
        pq.push(array[i]); // insert()

    while (!pq.empty()) {
        array[--n] = pq.top(); // max()
        pq.pop(); // removeMax()
    }
}
```
what does this function do?

```cpp
void foo(int *array, int n) {
    std::priority_queue<int> pq;
    for (int i = 0; i < n; i++)
        pq.push(array[i]); // insert()

    while (!pq.empty()) {
        array[--n] = pq.top(); // max()
        pq.pop(); // removeMax()
    }
}
```

Sorting!
void foo(int *array, int n) {
    std::priority_queue<int> pq;

    for (int i = 0; i < n; i++)
        pq.push(array[i]); // insert()

    while (!pq.empty()) {
        array[--n] = pq.top(); // max()
        pq.pop(); // removeMax()
    }
}
// what does this function do?

```cpp
void foo(int *array, int n) {
    std::priority_queue<int> pq;
    for (int i = 0; i < n; i++)
        pq.push(array[i]); // insert()

    while (!pq.empty()) {
        array[--n] = pq.top(); // max()
        pq.pop(); // removeMax()
    }
}
```

The running time of this sorting method depends on the priority queue implementation.
From https://xkcd.com/835/
(max) Heap

Structure Property

a heap is a complete binary tree
(max) Heap

Structure Property
a heap is a complete binary tree

Heap-Order Property
for every node $x$, $\text{key(parent}(x)) \geq \text{key}(x)$
except the root, which has no parent
Height of a heap?

What is the minimum number of nodes in a complete binary tree of height $h$?
Height of a heap?

What is the minimum number of nodes in a complete binary tree of height $h$?

\[ n \geq 2^h \]
\[ \log n \geq \log 2^h \]
\[ \log n \geq h \]
Height of a heap?

What is the minimum number of nodes in a complete binary tree of height $h$?

$n \geq 2^h$

$\log n \geq \log 2^h$

$\log n \geq h$
Height of a heap?

What is the minimum number of nodes in a complete binary tree of height $h$?

$$n \geq 2^h$$

$$\log n \geq \log 2^h$$

$$\log n \geq h$$
Example

Structure Property?

Heap-order Property?
Example

Structure Property?

Heap-order Property?
Example

Structure Property?

Heap-order Property?
Implementation
Implementation

Complete tree ...
Implementation

Complete tree ...
Implementation

Complete tree ...

<table>
<thead>
<tr>
<th></th>
<th>50</th>
<th>30</th>
<th>45</th>
<th>25</th>
<th>22</th>
<th>45</th>
<th>7</th>
<th>3</th>
<th>15</th>
<th>18</th>
<th>5</th>
<th>9</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Implementation

Complete tree ...

\[
\begin{array}{cccccccccccccccc}
50 & 30 & 45 & 25 & 22 & 45 & 7 & 3 & 15 & 18 & 5 & 9 & 31 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13
\end{array}
\]

\[\text{parent}(i)\]

\[\text{left}_\text{child}(i)\]

\[\text{right}_\text{child}(i)\]
Implementation

Complete tree ...

parent(i)  

left_child(i)  

floor(i/2)  

right_child(i)
Implementation

Complete tree ...

<table>
<thead>
<tr>
<th>50</th>
<th>30</th>
<th>45</th>
<th>25</th>
<th>22</th>
<th>45</th>
<th>7</th>
<th>3</th>
<th>15</th>
<th>18</th>
<th>5</th>
<th>9</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
</tbody>
</table>

parent(i)

floor(i/2)

left_child(i)
i*2

right_child(i)
Implementation

Complete tree ...

<table>
<thead>
<tr>
<th></th>
<th>50</th>
<th>30</th>
<th>45</th>
<th>25</th>
<th>22</th>
<th>45</th>
<th>7</th>
<th>3</th>
<th>15</th>
<th>18</th>
<th>5</th>
<th>9</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
</tbody>
</table>

parent(i)
floor(i/2)
left_child(i)
i*2
right_child(i)
i*2 + 1
insert
Insert
Insert

Append new element to the end of array
Insert

Append new element to the end of array

Check heap-order property
Insert

Append new element to the end of array

Check heap-order property

if violated, **Up-Heap** (swap with parent)
Insert

Append new element to the end of array

Check heap-order property

if violated, **Up-Heap** (swap with parent)

*repeat* until heap-order is restored
Insert

Append new element to the end of array

Check heap-order property

  if violated, **Up-Heap** (swap with parent)

    **repeat** until heap-order is restored

  if not, we are done
Insert

Append new element to the end of array

Check heap-order property
  if violated, Up-Heap (swap with parent)
    repeat until heap-order is restored
  if not, we are done

O(\log n)
insert 47
insert 47
insert 42
removeMax
removeMax
removeMax

Max element is the the first element of the array
Max element is the first element of the array the root of the heap
removeMax

Max element is the the **first** element of the array
the root of the heap

Copy last element of array to first position
removeMax

Max element is the the **first** element of the array
the root of the heap

Copy last element of array to first position
then decrement array size by 1 (removes last element)
removeMax

Max element is the first element of the array, the root of the heap.

Copy last element of array to first position, then decrement array size by 1 (removes last element).

Check heap-order property.
removeMax

Max element is the first element of the array
the root of the heap

Copy last element of array to first position
then decrement array size by 1 (removes last element)

Check heap-order property
if violated, Down-Heap (swap with larger child)
removeMax

Max element is the first element of the array
the root of the heap

Copy last element of array to first position
then decrement array size by 1 (removes last element)

Check heap-order property
if violated, Down-Heap (swap with larger child)
repeat until heap-order is restored
removeMax

Max element is the first element of the array
the root of the heap

Copy last element of array to first position
then decrement array size by 1 (removes last element)

Check heap-order property
if violated, Down-Heap (swap with larger child)
repeat until heap-order is restored
if not, we are done
removeMax

Max element is the the *first* element of the array
the root of the heap

Copy last element of array to first position
then decrement array size by 1 (removes last element)

Check heap-order property
  if violated, **Down-Heap** (swap with *larger* child)
    repeat until heap-order is restored
  if not, we are done

\[ O(\log n) \]
removeMax
removeMax
The diagram illustrates a binary max heap, after removing the maximum value. The heap is represented as follows:

```
42  30   47  25  22  45  45  3  15  18  5  9  31  7  42
```

The diagram shows the node labeled 42 being removed, leaving the heap with the maximum value 47 at the root node.
removeMax
removeMax
# Performance

<table>
<thead>
<tr>
<th></th>
<th>Sorted Array/List</th>
<th>Unsorted Array/List</th>
<th>AVL</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>insert</code></td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td></td>
</tr>
<tr>
<td><code>removeMax</code></td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
<td></td>
</tr>
<tr>
<td><code>max</code></td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
<td></td>
</tr>
<tr>
<td><code>insert N</code></td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
<td>$O(n \log n)$</td>
<td></td>
</tr>
</tbody>
</table>
## Performance

<table>
<thead>
<tr>
<th></th>
<th>Sorted Array/List</th>
<th>Unsorted Array/List</th>
<th>AVL</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>insert</strong></td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td><strong>removeMax</strong></td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td><strong>max</strong></td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(1)</td>
</tr>
<tr>
<td><strong>insert N</strong></td>
<td>O(n²)</td>
<td>O(n)</td>
<td>O(n log n)</td>
<td>O(n)**</td>
</tr>
</tbody>
</table>

(**) assuming we know the sequence in advance (`buildHeap`)
buildHeap
Problem

Build a heap by inserting a sequence of $n$ elements

‘easy’: call insert $n$ times
Problem

Build a heap by inserting a sequence of $n$ elements

‘easy’: call insert $n$ times $\mathcal{O}(n \log n)$
Problem

Build a heap by inserting a sequence of $n$ elements

‘easy’: call `insert` $n$ times $\mathcal{O}(n \log n)$

Can we do it in linear time?
buildHeap

Place \textbf{n} items into the tree (array) in \textit{any order} and it keeps the structure property.
buildHeap

Place $n$ items into the tree (array) in any order
   keeps structure property

Perform **Down-Heap** on each internal node
   from parent($n$) to 1
   keeps heap-order property
buildHeap example

input: 10 42 25 13 17 33 45 50 15 22 30 18
buildHeap example

input: 10 42 25 13 17 33 45 50 15 22 30 18

internal nodes
buildHeap example

input: 10 42 25 13 17 33 45 50 15 22 30 18
buildHeap example

input:  10  42  25  13  17  33  45  50  15  22  30  18
buildHeap example

input: 10 42 25 13 17 33 45 50 15 22 30 18
buildHeap example

input: 10 42 25 13 17 33 45 50 15 22 30 18
buildHeap example

input:  10  42  25  13  17  33  45  50  15  22  30  18
buildHeap example

input:  10  42  25  13  17  33  45  50  15  22  30  18
Analysis

Cost is **sum of the heights** of all internal nodes

assume tree is full and complete, thus $n=2^{h+1}-1$
Analysis

Cost is **sum of the heights** of all internal nodes

assume tree is full and complete, thus \( n = 2^{h+1} - 1 \)
Analysis

Cost is **sum of the heights** of all internal nodes

assume tree is full and complete, thus \( n = 2^{h+1} - 1 \)

\[
T(n) = h + 2(h - 1) + 4(h - 2) + 8(h - 3) + \cdots + 2^h(0)
= \sum_{i=0}^{h} 2^i(h - i) = h \sum_{i=0}^{h} 2^i - \sum_{i=0}^{h} i2^i
= h [2^{h+1} - 1] - [2 + (h + 1 - 2)2^{h+1}]
= 2^{h+1} - h - 2 = n - (h + 1)
= O(n)
\]
Analysis

Cost is \textbf{sum of the heights} of all internal nodes
assume tree is full and complete, thus \( n = 2^{h+1} - 1 \)

\[
T(n) = h + 2(h - 1) + 4(h - 2) + 8(h - 3) + \cdots + 2^h(0)
\]

\[
= \sum_{i=0}^{h} 2^i(h - i)
= h \sum_{i=0}^{h} 2^i - \sum_{i=0}^{h} i2^i
\]

\[
= h \left[ 2^{h+1} - 1 \right] - \left[ 2 + (h + 1 - 2)2^{h+1} \right]
\]

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

\( = O(n) \) 👍