• This weekend (Nov 11-12), YHack will be hosting a 24-hour hackathon

• Registration link: tiny.cc/yminihack

• Registration deadline: Nov 9

• Additional information: yhackmini.org
Priority Queues and Heaps
Revisiting Queues

- How do people board a plane?
  - Mainly a queue but “not everyone is the same”
  - First class, frequent flyers, families, …
Airplane boarding example

- Passengers join the “queue”

  ```c
  void enqueue(key, person)
  ```

- Need to know first in line

- How to compare?

  Priority, arrival time, time, ...

Complicated formula!
On priorities

- Combination of traits
  - Fare
  - Arrival time
  - Frequent flyer status, etc...

- Need to “compare” people
Three fundamental operations

- \text{insert}(\text{key, element}) = \text{ENQUEUE}
- \text{min\_element()}
- \text{remove\_min() = DEQUEUE}
### Example

<table>
<thead>
<tr>
<th>Operation</th>
<th>Output</th>
<th>Priority Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert(5, A)</td>
<td>-</td>
<td>{(5, A)}</td>
</tr>
<tr>
<td>insert(9, C)</td>
<td>-</td>
<td>{(5, A), (9, C)}</td>
</tr>
<tr>
<td>insert(3, B)</td>
<td>-</td>
<td>{(5, A), (9, C), (3, B)}</td>
</tr>
<tr>
<td>insert(7, D)</td>
<td>-</td>
<td>{(5, A), (9, C), (3, B), (7, D)}</td>
</tr>
<tr>
<td>min_element()</td>
<td>B</td>
<td>{(5, A), (9, C), (3, B), (7, D)}</td>
</tr>
<tr>
<td>min_key()</td>
<td>3</td>
<td>{(5, A), (9, C), (3, B), (7, D)}</td>
</tr>
</tbody>
</table>
Inserting in an unsorted container

Insert?

- Array List/Linked list insertion
  - $O(1)$ unless we need to expand

min_element?min_key?remove_min()? 

- Search through whole list and find smallest
  - $O(n)$ always
Introducing Heaps

- Tree based data structure
  - Root, parent, child, leaf
  - Binary Tree
    - At most two children
Shape invariant

Nodes filled from **left** to **right** and **top** to **bottom**

Not a heap!
Value invariant

*Option 1: Parent is smaller than both children (minHeap)*

*Option 2: Parent is larger than both children (maxHeap)*
Usage?

- Super easy to find minimum in minHeap (maximum in maxHeap)
- Opposite is hard
- Simple DS, should be easy to update
Rules:
- binary
- max
- complete
Rules:

- **binary**: internal nodes have 1 or 2 children
- **max**
- **complete**
Rules:

- **binary**: internal nodes have 1 or 2 children
- **max**: parent \( \geq \) child
- **complete**
Rules:

• **binary**: internal nodes have 1 or 2 children

• **max**: parent ≥ child

• **complete**: all levels filled (lowest can be partial, left to right)
Notice every subtree is also a heap

Rules:

- **binary**: internal nodes have 1 or 2 children
- **max**: parent ≥ child
- **complete**: all levels filled (lowest can be partial, left to right)
How can we identify the indices of the children of a given node?
How can we identify the indices of the children of a given node?

\[
\text{left-child}(i) = 2i \\
\text{right-child}(i) = 2i + 1
\]
How can we identify the indices of the children of a given node?

\[
\text{left-child}(i) = 2i \\
\text{right-child}(i) = 2i + 1 \\
\text{parent}(i) = ?
\]
How can we identify the indices of the children of a given node?

\[
\text{left-child}(i) = 2i \\
\text{right-child}(i) = 2i + 1 \\
\text{parent}(i) = \left\lfloor \frac{i}{2} \right\rfloor
\]
How can we identify the indices of the children of a given node?

- left-child(i) = 2i
- right-child(i) = 2i + 1
- parent(i) = \lfloor i/2 \rfloor

Use array to store heap (avoid wasting space with pointers)

```
16 14 10 8 7 9 3 2 4 1 ...
```

minElement()

- Just return top of heap

Just an array!
Return h[1]
Insert

- The shape invariant tells us where to insert
- How to preserve the value invariant?
- Need to **float** the number
Example: insert(9)

- The shape invariant tells us the location
- How to preserve the value invariant?
- Need to `float` the number
Example: insert(9)

- The shape invariant tells us the location
- How to preserve the value invariant?
- Need to float the number
Example: insert(9)

- The shape invariant tells us the location
- How to preserve the value invariant?
- Need to **float** the number
Example: insert(9)

- The shape invariant tells us the location
- How to preserve the value invariant?
- Need to float the number
Example: insert(9)

- The shape invariant tells us the location
- How to preserve the value invariant?
- Need to **float** the number

**Done!**
Implementing Insertions

\[ h[\text{numElem}] = \text{newNumber} \]
\[ \text{float}(\text{numElem}) \]
\[ \text{numElem}++; \]
Float operation

float(index)

If index==1 return

If h[parent(index)] > h[index]

Swap index and parent(index)

float(parent(index))

runtime? O(h), where h is height of tree
What about remove_min?

Preserve the shape invariant!
Can only remove last leaf
What about remove_min?

Move last to top
What about removal?

Need to sink 13. Where?
Oops!

Sibling is not happy
Swap 13 with SMALLEST child

Recursively sink down
Swap 13 with SMALLEST child
Final step: decrease numElem
Heap building: the FORWARD METHOD
Heap building: the FORWARD METHOD (left to right)
Heap building: the FORWARD METHOD (left to right)
Heap building: the FORWARD METHOD (left to right)
Heap building: the **FORWARD METHOD** (left to right)

\[ 9 \ 3 \ 7 \ 10 \ 8 \ 4 \ 14 \ 2 \ 16 \ 1 \]

\[ \text{Diagram:} \]

- Node 9 is the root.
- Node 3 is the left child of node 9.
- Arrows indicate the parent-child relationship.
Heap building: the **FORWARD METHOD** (left to right)

```
9 3 7 10 8 4 14 2 16 1
```

Diagram:

![Diagram of a heap with numbers and arrows indicating the forward method of heap building.](image-url)
Heap building: the FORWARD METHOD (left to right)
Heap building: the FORWARD METHOD (left to right)
Heap building: the FORWARD METHOD (left to right)
Heap building: the FORWARD METHOD (left to right)

```
1 2 3 4 5 6 7 8 9 10
10 9 7 3 8 4 14 2 16 1
```

```
10
  / \  \
 /   \ 
9     7
  / \  \
  /   \ 
 3   2   4
```

time?
Heap building: the *FORWARD METHOD* (left to right)

```
1 2 3 4 5 6 7 8 9 10
10 9 7 3 8 4 14 2 16 1
```

`10` becomes the root of the heap.

```
10
/   \
9    11
/ \
2   3
/ \
3   7
```

*Time = O(n log n)*

*O(log n) per insertion*
Heap building: the *FORWARD METHOD* (left to right)

1 2 3 4 5 6 7 8 9 10

10 9 7 3 8 4 14 2 16 1

time = $O(n \log n)$

$O(\log n)$ per insertion

Works for streaming data
Heap building: the REVERSE METHOD (right to left)
Heap building: the REVERSE METHOD (right to left)

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>9</td>
<td>7</td>
<td>3</td>
<td>8</td>
<td>4</td>
<td>14</td>
<td>2</td>
<td>16</td>
<td>1</td>
</tr>
</tbody>
</table>

Diagram of a binary heap:

```
  10
 /   \
9    7
 / \
3 8 4
 / \  /
2 16 1
```
Heap building: the REVERSE METHOD (right to left)

10 9 7 3 8 4 14 2 16 1

already heaps
Heap building: the REVERSE METHOD (right to left)

already heaps
Heap building: the REVERSE METHOD (right to left)

10 9 7 3 8 4 14 2 16 1

heapify next

already heaps
Heap building: the REVERSE METHOD (right to left)

| 10 | 9 | 7 | 3 | 8 | 4 | 14 | 2 | 16 | 1 |

heapify next

already heaps
Heap building: the REVERSE METHOD (right to left)

1 2 3 4 5 6 7 8 9 10

10 9 7 3 8 4 14 2 16 1

already heaps
Heap building: the REVERSE METHOD (right to left)

1 2 3 4 5 6 7 8 9 10

10 9 7 3 8 4 14 2 16 1

already heaps
Heap building: the REVERSE METHOD (right to left)

1 2 3 4 5 6 7 8 9 10

10 9 7 3 8 4 14 2 16 1

heapify next

already heaps
Heap building: the **REVERSE METHOD** (right to left)

```
1 2 3 4 5 6 7 8 9 10
10 9 7 3 8 4 14 2 16 1
```

- `10`: already heap
- `9`: `3` is smaller than `9`, so swap
  - `3`: already heap
- `7`: `8` is smaller than `7`, so swap
  - `8`: already heap
- `14`: already heap

heapify next
Heap building: the REVERSE METHOD (right to left)

1 2 3 4 5 6 7 8 9 10

10 9 7 3 8 4 14 2 16 1

heapify next

already heaps
Heap building: the REVERSE METHOD (right to left)

| 10 | 9 | 7 | 16 | 8 | 4 | 14 | 2 | 3 | 1 |

heapify next

already heaps
Heap building: the REVERSE METHOD (right to left)

already heaps
Heap building: the REVERSE METHOD (right to left)

already heaps
Heap building: the REVERSE METHOD (right to left)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>9</td>
<td>7</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>14</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

---

10

9

16

8

4

14

3

2

1

heapify next

already heaps
Heap building: the REVERSE METHOD (right to left)

1 2 3 4 5 6 7 8 9 10

10 9 7 16 8 4 14 2 3 1

heapify next

already heaps
Heap building: the REVERSE METHOD (right to left)

1 2 3 4 5 ? 6 7 8 9 10

10 9 7 16 8 4 14 2 3 1

heapify next

already heaps
Heap building: the REVERSE METHOD (right to left)

1 2 3 4 5 6 7 8 9 10

10 9 14 16 8 4 7 2 3 1

10

9

16

8

2

3

14

4

7

already heaps

heapify next
Heap building: the REVERSE METHOD (right to left)

already heaps
Heap building: the REVERSE METHOD (right to left)

already heaps
Heap building: the REVERSE METHOD (right to left)

Already heaps

Heapsify next
Heap building: the REVERSE METHOD (right to left)

Heapify next

already heaps
Heap building: the REVERSE METHOD (right to left)

1 2 3 4 5 6 7 8 9 10

10 9 14 16 8 4 7 2 3 1

heapify next

already heaps
Heap building: the REVERSE METHOD (right to left)

heapify next

already heaps
Heap building: the **REVERSE METHOD** (right to left)

```
1 2 3 4 5 6 7 8 9 10
10 16 14 9 8 4 7 2 3 1
```

heapify next

already heaps
Heap building: the REVERSE METHOD (right to left)

Heapify next

already heaps
Heap building: the REVERSE METHOD (right to left)

1 2 3 4 5 6 7 8 9 10

10 16 14 9 8 4 7 2 3 1

already heaps
Heap building: the **REVERSE METHOD** (right to left)

1 2 3 4 5 6 7 8 9 10

10 16 14 9 8 4 7 2 3 1

![Binary Heap Diagram]

already heaps
Heap building: the REVERSE METHOD (right to left)

heapify next

already heaps
Heap building: the REVERSE METHOD (right to left)

heapify next

already heaps
Heap building: the REVERSE METHOD (right to left)

12345678910

1610149847231

heapify next

16

10

9>

8

231

14

47

already heaps
Heap building: the REVERSE METHOD (right to left)

16 10 14 9 8 4 7 2 3 1

already heaps
Heap building: the REVERSE METHOD (right to left)

1 2 3 4 5 6 7 8 9 10

16 10 14 9 8 4 7 2 3 1

Time?
Heap building: the REVERSE METHOD (right to left)
Heap building: the REVERSE METHOD (right to left)

1 2 3 4 5 6 7 8 9 10

16 10 14 9 8 4 7 2 3 1

Time?

$\text{heapify}(x) = O(\text{height}(x))$

$\sum_{\text{all } x} \text{height}(x) = O(n \log n)$
better calculation

\[ \sum \text{height}(x) \]

all \( x \)

time? \( O(n) \)
\[ \sum \text{height}(x) \leq \frac{n}{2} \cdot 1 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3 + \cdots + 2 \cdot ((\log n) - 1) + 1 \cdot \log n \]

\#nodes \quad \text{height}

lowest level

\#nodes \quad \text{height}

root level
## Summary

<table>
<thead>
<tr>
<th>Sequence Type</th>
<th>(\text{insert}(k,e))</th>
<th>(\text{min_element}())</th>
<th>(\text{remove_min}())</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unordered List</td>
<td>(O(1))</td>
<td>(O(n))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>Ordered List</td>
<td>(O(n))</td>
<td>(O(1))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>Heap</td>
<td>(O(\log n))</td>
<td>(O(1))</td>
<td>(O(\log n))</td>
</tr>
</tbody>
</table>

Can we do better?
### Summary

<table>
<thead>
<tr>
<th>Sequence Type</th>
<th>insert((k,e))</th>
<th>min_element()</th>
<th>remove_min()</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unordered List</strong></td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td><strong>Ordered List</strong></td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td><strong>Heap</strong></td>
<td>O(log(n))</td>
<td>O(1)</td>
<td>O(log(n))</td>
</tr>
<tr>
<td><strong>Fibonacci Heaps</strong></td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>