CMPSC 250
Analysis of algorithms

Dr. Aravind Mohan
Allegheny College
February 21, 2018
Lecture 13 - Heap Sort
Heap

- **Binary Heap** is a binary tree with two special properties:?
  1. **Order property**: The value in node \( n \) is \( \geq \) the values in its children, for every node \( n \) (MAX heap). How about MIN heap?
  2. **Shape property**: All leaves are either at depth \( d \) or \( d - 1 \) for some \( d \) (a) There is at most 1 node with just 1 child. (b) That child is the left child of its parent, and (c) it is the rightmost leaf at depth \( d \).
Heap

- **Binary Heap** is a binary tree with two special properties

Heap properties:

1. **Order** property: The value in node \( n \) is \( \geq \) the values in its children, for every node \( n \) (MAX heap). How about MIN heap?
Binary Heap is a binary tree with two special properties

Heap properties:

1. **Order** property: The value in node $n$ is $\geq$ the values in its children, for every node $n$ (MAX heap). How about MIN heap?

2. **Shape** property:
   1. All leaves are either at depth $d$ or $d - 1$ for some $d$.
   2. All of the leaves at depth $d - 1$ are to the right of the leaves at depth $d$.
   3. (a) There is at most 1 node with just 1 child. (b) That child is the left child of its parent, and (c) it is the rightmost leaf at depth $d$. 

Dr. Aravind Mohan (Allegheny College)
Implementing Priority Queues using Heaps

- Use array (or ArrayList) starting at position 1 (not 0)
- Each item in the array corresponds to one node in the heap
  - The root of the heap is always in array[1]
  - Its left child is in array[2]
  - Its right child is in array[3]
Implementing Priority Queues using Heaps

- Use array (or ArrayList) starting at position 1 (not 0)
- Each item in the array corresponds to one node in the heap
  - The root of the heap is always in array[1]
  - Its left child is in array[2]
  - Its right child is in array[3]
  - Generally, if a node is in array[k], then its left child is in array[k * 2], and its right child is in array[k * 2 + 1]
  - If a node is in array[k], then its parent is in array[k / 2]
Heap Sort

- **Reverse the heap property**: child nodes should always be less than the parent nodes
- **Phase 1**: convert the array into an $n$-element heap
Heap Sort

- **Reverse the heap property**: child nodes should always be less than the parent nodes
- **Phase 1**: convert the array into an \( n \)-element heap
- **Phase 2**: repeatedly remove maximum element from the heap, and place that element in its proper position in the array
  - swap element at 0th position with element at \( (n - 1) \)th position and then “reheapify” considering only the first \( n - 1 \) elements
  - repeat this process until heap size is reduced to 1 (minimum element remains, at 0th position)
Heap Sort

6  10  5  12  3  9  20  2  15  8  18

Diagram of a heap: 6 at the root, with 10 and 5 as its children, 12 and 3 as children of 10, and 9 and 20 as children of 5, with 2, 15, 8, and 18 as children of 12 and 3 respectively.
Heap Sort: Phase 1 - build the heap

for $i = 1$ to $n - 1$ do
    insert element $s[i]$ into the heap consisting of the elements $s[0]...s[i - 1]$

Once the heap is built, $s[0]$ will contain the maximum element
Phase 1 - build the Heap

Diagram of a binary heap with values:
- Root: 6
- Level 1: 10 (right), 5 (left)
- Level 2: 12 (left), 3 (right), 9 (left), 20 (right)
- Level 3: 2, 15, 8, 18
Phase 1 - build the Heap
Phase 1 - build the Heap

Diagram of a binary heap with the following structure:
- Top node: 10
- Left child of top node: 6
  - Left child of 6: 12
    - Left child of 12: 2
    - Right child of 12: 15
  - Right child of 6: 3
    - Left child of 3: 8
    - Right child of 3: 18
- Right child of top node: 5
  - Left child of 5: 9
  - Right child of 5: 20
Phase 1 - build the Heap
Phase 1 - build the Heap

```
   12
  /   \
10    5
 / \   / \
6   3 9  20
 / \ / \ / \
2  15 8  18```

Phase 1 - build the Heap

```
          12
         /   \
       10    9
      /   \   /  \
    6     3  5    20
   / \   /\  /   /  \
  2   15 8 18
```
Phase 1 - build the Heap

```
     20
   /   \
  10   12
 / \\ / \ \
6 10 3 12
/ \ / \ / \ \
2 15 8 5 9
```

Dr. Aravind Mohan (Allegheny College)
Phase 1 - build the Heap
Phase 1 - build the Heap
Phase 1 - build the Heap
Phase 1 - build the Heap

```
    20
   /   
 18    12
 / 
10   15
 /   / 
2   6  3  8
 /       
5   9
```

**Maximum element**
Phase 2 - repeatedly select the max

for $i = n - 1$ to 1 do
  swap $s[0]$ and $s[i]$
  demote $s[0]$ to its proper place
in the heap consisting of the elements $s[0]...s[i-1]$
Phase 2 - select max
Phase 2 - select max
Phase 2 - select max

Diagram of a tree with nodes labeled with numbers: 18, 15, 12, 10, 8, 5, 9, 2, 6, 3, 20.
Phase 2 - select max
Phase 2 - select max

```
   3
  / \  /  \
15 12 /   /
/    /     /     \
10   8  5  9
/  / \
2  6 18 20
```
Phase 2 - select max

![Binary tree with numbers: 15, 10, 6, 8, 2, 3, 18, 20, 12, 5, 9.]

- 15
  - 10
    - 6
      - 2
      - 3
    - 8
      - 18
      - 20
  - 12
    - 5
    - 9
Phase 2 - select max

Diagram of a tree with numbers at each node.
Phase 2 - select max
Phase 2 - select max

Diagram of a tree with nodes labeled from 2 to 20.
Phase 2 - select max
Phase 2 - select max
Phase 2 - select max
Phase 2 - select max

```
  3
 / \
8   9
/ \ / \   
6 8 2 5 10
/ \ /   /   
12 15 18 20
  
```
Phase 2 - select max

![Binary Tree Diagram]

- 9
- 8
  - 6
    - 12
    - 15
  - 2
    - 18
    - 20
- 5
  - 3
  - 10
Phase 2 - select max

```
        3
       /\  \
      8  5
     / \ / \  \
   6  2 9 10
  / \  /  /  \
12 15 18 20
```
Phase 2 - select max

```
8
/  \
6   5
/  \
3   2
/  \
12  15

18  20

9   10
```
Phase 2 - select max
Phase 2 - select max

![Binary Tree Diagram]

- Root Node: 6
- Left Child: 3
  - Left Child: 2
    - Left Child: 12
    - Right Child: 15
  - Right Child: 8
    - Left Child: 18
    - Right Child: 20
- Right Child: 5
  - Left Child: 9
  - Right Child: 10
Phase 2 - select max

```
2
/   \
| 3   |
|    / |
|   6  |
|  12  |
|  15  |
|  18  |
|  20  |
```

```
5
/   \
|  9 |
|  10 |
```
Phase 2 - select max

```
    5
   / \  /
  3   2
 / \  / \  /
6   8 9 10
/ \ / \  / \
12 15 18 20
```
Phase 2 - select max
Phase 2 - select max
Phase 2 - select max

```
    2
   / \
  3   5
 / \  /  \
6   8 9   10
/ \ / \ /   /  \
12 15 18 20
```
Heap Sort Completed
Heap Sort Complexity

for \(i \leftarrow 1\) to \(n-1\) do

insert element \(s[i]\) into the heap consisting of the elements \(s[0]...s[i-1]\)

\(O(n \log n)\) operations

for \(i \leftarrow n-1\) down to \(1\) do

swap \(s[0]\) and \(s[i]\)

“demote” \(s[0]\) to its proper place in the heap consisting of the elements \(s[0]...s[i-1]\)

\(O(n \log n)\)
Heap Sort

Note that heap sort is just a more clever version of selection sort since a maximum is repeatedly selected and placed in its proper position.
## Comparision

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection-sort</td>
<td>$O(n^2)$</td>
<td>slow, in-place, for small data sets (&lt; 1K)</td>
</tr>
<tr>
<td>insertion-sort</td>
<td>$O(n^2)$</td>
<td>slow, in-place, for small data sets (&lt; 1K)</td>
</tr>
<tr>
<td>heap-sort</td>
<td>$O(n \log n)$</td>
<td>fast, in-place, for large data sets (1K — 1M)</td>
</tr>
<tr>
<td>merge-sort</td>
<td>$O(n \log n)$</td>
<td>fast, sequential data access, for huge data sets (&gt; 1M)</td>
</tr>
</tbody>
</table>