Heaps Part 02
Definition: A data structure that can be defined recursively as a collection of nodes, where each node is a data structure consisting of a value, together with a list of references (edges) to nodes, with the constraints that no reference is duplicated, and none points to the root.
• Trees Have
  – Nodes
  – Edges

• Trees CANNOT
  – Contain Self-Referencing Edges
  – Have Cycles
  – Be Disjointed
• Binary Tree Structure
• Node’s data must be comparable
• Node’s have at most two children
  – Left Child
  – Right Child
• Max Heap: Children must be less than or equal to the parent
• Min Heap: Children must be greater than or equal to the parent
• Assume Leaves are NULL references
• Array Heap
• Assume Root is at Index 0
• Left Child Index = Parent Index * 2 + 1
• Right Child Index = Parent Index * 2 + 2
• Parent Index = (Child Index-1)/2
• Add
  – Replace the first leaf in breadth order with the new data
  – From that node “bubble up” the data if necessary

• Bubble Up
  – If the child’s data is smaller than the parent then swap that information
  – Continue swapping child data with parent data until the parent is smaller than the child or we reach the root index
• **Add**
  – Replace the first leaf in breadth order with the new data
  – From that node “bubble up” the data if necessary

• **Bubble Up**
  – If the child’s data is smaller than the parent then swap that information
  – Continue swapping child data with parent data until the parent is smaller than the child or we reach the root index
• **Add**
  – Replace the first leaf in breadth order with the new data
  – From that node “bubble up” the data if necessary

• **Bubble Up**
  – If the child’s data is smaller than the parent then swap that information
  – Continue swapping child data with parent data until the parent is smaller than the child or we reach the root index

---

**Example Adding “0”**

- Add 0 to the tree.
- Replace the first leaf in breadth order with the new data.
- From that node “bubble up” the data if necessary.

- Bubble Up:
  - If the child’s data is smaller than the parent, then swap.
  - Continue swapping child data with parent data until the parent is smaller than the child or we reach the root index.

---

**Diagram**

- Tree structure with nodes showing the process of adding “0” to the tree.
• **Add**
  – Replace the first leaf in breadth order with the new data
  – From that node “bubble up” the data if necessary

• **Bubble Up**
  – If the child’s data is smaller than the parent then swap that information
  – Continue swapping child data with parent data until the parent is smaller than the child or we reach the root index

**Example Adding “0”**
• **Add**
  – Replace the first leaf in breadth order with the new data
  – From that node “bubble up” the data if necessary

• **Bubble Up**
  – If the child’s data is smaller than the parent then swap that information
  – Continue swapping child data with parent data until the parent is smaller than the child or we reach the root index

---

**Example Adding “0”**

```plaintext
[0] 1
[1] 3
[2] 0
[3] 5
[4] 6
[5] 4
[6] 2
[7] 5
[8] 6
[9] 4
[10] 2
[11] 0
[12] null
null
null
null
null
null
null
```
Add
- Replace the first leaf in breadth order with the new data
- From that node “bubble up” the data if necessary

Bubble Up
- If the child’s data is smaller than the parent then swap that information
- Continue swapping child data with parent data until the parent is smaller than the child or we reach the root index

Example Adding “0”
• Add
  – Replace the first leaf in breadth order with the new data
  – From that node “bubble up” the data if necessary

• Bubble Up
  – If the child’s data is smaller than the parent then swap that information
  – Continue swapping child data with parent data until the parent is smaller than the child or we reach the root index

Example Adding “0”
• Add
  – Replace the first leaf in breadth order with the new data
  – From that node “bubble up” the data if necessary

• Bubble Up
  – If the child’s data is smaller than the parent then swap that information
  – Continue swapping child data with parent data until the parent is smaller than the child or we reach the root index

Example Adding “0”
• Remove
  – Store the data at the Root
  – Replace the Root data with the Data in the last node in Breadth Order
  – Starting from the root, “Bubble Down” that information
  – Return the stored value, previously at the root

• Bubble Down
  – Pick the smaller of the 2 children
  – If its value is smaller than the parent, then swap those values
  – Continue this until the parent’s value is smaller or we reach the tree’s bounds
• Remove
  – Store the data at the Root
  – Replace the Root data with the Data in the last node in Breadth Order
  – Starting from the root, “Bubble Down” that information
  – Return the stored value, previously at the root

• Bubble Down
  – Pick the smaller of the 2 children
  – If its value is smaller than the parent, then swap those values
  – Continue this until the parent’s value is smaller or we reach the tree’s bounds

Example Remove

```
Example Remove
[0]
[1]
[2]
```

```

diagram

[0]
  [1]
    [3]       [2]
      [5]     [4]
        [6]
```

```
[4]
```
• Remove
  – Store the data at the Root
  – Replace the Root data with the Data in the last node in Breadth Order
  – Starting from the root, “Bubble Down” that information
  – Return the stored value, previously at the root

• Bubble Down
  – Pick the smaller of the 2 children
  – If its value is smaller than the parent, then swap those values
  – Continue this until the parent’s value is smaller or we reach the tree’s bounds
Heaps

- **Remove**
  - Store the data at the Root
  - Replace the Root data with the Data in the last node in Breadth Order
  - Starting from the root, “Bubble Down” that information
  - Return the stored value, previously at the root

- **Bubble Down**
  - Pick the smaller of the 2 children
  - If its value is smaller than the parent, then swap those values
  - Continue this until the parent’s value is smaller or we reach the tree’s bounds

**Example Remove**

```
    1          2
     / \
   [3]  [4]

Return Value: 1
```
Heaps

- **Remove**
  - Store the data at the Root
  - Replace the Root data with the Data in the last node in Breadth Order
  - Starting from the root, “Bubble Down” that information
  - Return the stored value, previously at the root

- **Bubble Down**
  - Pick the smaller of the 2 children
  - If its value is smaller than the parent, then swap those values
  - Continue this until the parent’s value is smaller or we reach the tree’s bounds

Example Remove

```
  4
 /  \
3   5
 /     \
2   6
```

Return Value: 1
• Remove
  – Store the data at the Root
  – Replace the Root data with the Data in the last node in Breadth Order
  – Starting from the root, “Bubble Down” that information
  – Return the stored value, previously at the root

• Bubble Down
  – Pick the smaller of the 2 children
  – If its value is smaller than the parent, then swap those values
  – Continue this until the parent’s value is smaller or we reach the tree’s bounds

Example Remove

Return Value: 1
Heaps

• Remove
  – Store the data at the Root
  – Replace the Root data with the Data in the last node in Breadth Order
  – Starting from the root, “Bubble Down” that information
  – Return the stored value, previously at the root

• Bubble Down
  – Pick the smaller of the 2 children
  – If its value is smaller than the parent, then swap those values
  – Continue this until the parent’s value is smaller or we reach the tree’s bounds

Example Remove

Return Value: 1

```
Remove
- Store the data at the Root
- Replace the Root data with the Data in the last node in Breadth Order
- Starting from the root, “Bubble Down” that information
- Return the stored value, previously at the root

Bubble Down
- Pick the smaller of the 2 children
- If its value is smaller than the parent, then swap those values
- Continue this until the parent’s value is smaller or we reach the tree’s bounds
```
**Heaps**

- **Remove**
  - Store the data at the Root
  - Replace the Root data with the Data in the last node in Breadth Order
  - Starting from the root, “Bubble Down” that information
  - Return the stored value, previously at the root

- **Bubble Down**
  - Pick the smaller of the 2 children
  - If its value is smaller than the parent, then swap those values
  - Continue this until the parent’s value is smaller or we reach the tree’s bounds

---

**Example Remove**

```
[0]
[1] [2]
[7] [8] [9] [10]
```

Return Value: 1
• Remove
  – Store the data at the Root
  – Replace the Root data with the Data in the last node in Breadth Order
  – Starting from the root, “Bubble Down” that information
  – Return the stored value, previously at the root

• Bubble Down
  – Pick the smaller of the 2 children
  – If its value is smaller than the parent, then swap those values
  – Continue this until the parent’s value is smaller or we reach the tree’s bounds
• Remove
  – Store the data at the Root
  – Replace the Root data with the Data in the last node in Breadth Order
  – Starting from the root, “Bubble Down” that information
  – Return the stored value, previously at the root

• Bubble Down
  – Pick the smaller of the 2 children
  – If its value is smaller than the parent, then swap those values
  – Continue this until the parent’s value is smaller or we reach the tree’s bounds
• Remove
  – Store the data at the Root
  – Replace the Root data with the Data in the last node in Breadth Order
  – Starting from the root, “Bubble Down” that information
  – Return the stored value, previously at the root

• Bubble Down
  – Pick the smaller of the 2 children
  – If its value is smaller than the parent, then swap those values
  – Continue this until the parent’s value is smaller or we reach the tree’s bounds

Example Remove Again

Return Value: 2
• **Remove**
  - Store the data at the Root
  - Replace the Root data with the Data in the last node in Breadth Order
  - Starting from the root, “Bubble Down” that information
  - Return the stored value, previously at the root

• **Bubble Down**
  - Pick the smaller of the 2 children
  - If its value is smaller than the parent, then swap those values
  - Continue this until the parent’s value is smaller or we reach the tree’s bounds

---

**Example Remove Again**

```
Return Value: 2
```

---

```
Example Remove Again

```

```
[0]

```

```
[1] [2]

```

```

```

```
[7] [8] [9] [10]

```

```
null null null null

```

```
null

```

```
null

```

```
null

```
• **Remove**
  – Store the data at the Root
  – Replace the Root data with the Data in the last node in Breadth Order
  – Starting from the root, “Bubble Down” that information
  – Return the stored value, previously at the root

• **Bubble Down**
  – Pick the smaller of the 2 children
  – If its value is smaller than the parent, then swap those values
  – Continue this until the parent’s value is smaller or we reach the tree’s bounds

**Example Remove Again**

![Example Tree]

Return Value: 2
• **Remove**
  – Store the data at the Root
  – Replace the Root data with the Data in the last node in Breadth Order
  – Starting from the root, “Bubble Down” that information
  – Return the stored value, previously at the root

• **Bubble Down**
  – Pick the smaller of the 2 children
  – If its value is smaller than the parent, then swap those values
  – Continue this until the parent’s value is smaller or we reach the tree’s bounds

---

**Example Remove Again**

```
[0]
[1]  [2]
[7]  [8]
```

Return Value: 2
• **Remove**
  – Store the data at the Root
  – Replace the Root data with the Data in the last node in Breadth Order
  – Starting from the root, “Bubble Down” that information
  – Return the stored value, previously at the root

• **Bubble Down**
  – Pick the smaller of the 2 children
  – If its value is smaller than the parent, then swap those values
  – Continue this until the parent’s value is smaller or we reach the tree’s bounds

**Example Remove Again**

Return Value: 2
- **Remove**
  - Store the data at the Root
  - Replace the Root data with the Data in the last node in Breadth Order
  - Starting from the root, “Bubble Down” that information
  - Return the stored value, previously at the root

- **Bubble Down**
  - Pick the smaller of the 2 children
  - If its value is smaller than the parent, then swap those values
  - Continue this until the parent’s value is smaller or we reach the tree’s bounds

Example Remove Again

Return Value: 2
• Remove
  – Store the data at the Root
  – Replace the Root data with the Data in the last node in Breadth Order
  – Starting from the root, “Bubble Down” that information
  – Return the stored value, previously at the root

• Bubble Down
  – Pick the smaller of the 2 children
  – If its value is smaller than the parent, then swap those values
  – Continue this until the parent’s value is smaller or we reach the tree’s bounds

Example Remove Again

```
[0] 3
[1] 5
[2] 4
[3] 6
[4] null
[5] null
[6] null
[7] null
[8] null
```

Return Value: 2
Heaps

• Remove
  – Store the data at the Root
  – Replace the Root data with the Data in the last node in Breadth Order
  – Starting from the root, “Bubble Down” that information
  – Return the stored value, previously at the root

• Bubble Down
  – Pick the smaller of the 2 children
  – If its value is smaller than the parent, then swap those values
  – Continue this until the parent’s value is smaller or we reach the tree’s bounds

Example Remove Again

Return Value: 2
• Remove
  – Store the data at the Root
  – Replace the Root data with the Data in the last node in Breadth Order
  – Starting from the root, “Bubble Down” that information
  – Return the stored value, previously at the root

• Bubble Down
  – Pick the smaller of the 2 children
  – If its value is smaller than the parent, then swap those values
  – Continue this until the parent’s value is smaller or we reach the tree’s bounds
• Heap Sort

**Min Heap**

```
1
   /   
  3   2
 /     |
5     6
```
Heap Sort
1. Add all Values to the Heap
2. Remove All Values from the Heap
3. DONE!
What?
• Heap Sort
1. Add all Values to the Heap
2. Remove All Values from the Heap
3. DONE!

Min Heap

Diagram of a Min Heap with values 1, 2, 3, 4, 5, 6.
• Heap Sort
  1. Add all Values to the Heap
  2. Remove All Values from the Heap
  3. DONE!

Heap Sort Values \{5,1,4,3,6,2\}
• Heap Sort
  1. Add all Values to the Heap
  2. Remove All Values from the Heap
  3. DONE!

Heap Sort Values \{5,1,4,3,6,2\}
• Heap Sort
  1. Add all Values to the Heap
  2. Remove All Values from the Heap
  3. DONE!

Heap Sort Values {5, 1, 4, 3, 6, 2}
- Heap Sort
  1. Add all Values to the Heap
  2. Remove All Values from the Heap
  3. DONE!

Heap Sort Values \{5, 1, 4, 3, 6, 2\}
• Heap Sort
  1. Add all Values to the Heap
  2. Remove All Values from the Heap
  3. DONE!

Heap Sort Values \{5,1,4,3,6,2\}
• Heap Sort
  1. Add all Values to the Heap
  2. Remove All Values from the Heap
  3. DONE!

Heap Sort Values \(\{5,1,4,3,6,2\}\)
• Heap Sort
  1. Add all Values to the Heap
  2. Remove All Values from the Heap
  3. DONE!
Heap Sort
1. Add all Values to the Heap
2. Remove All Values from the Heap
3. DONE!
• Heap Sort
  1. Add all Values to the Heap
  2. Remove All Values from the Heap
  3. DONE!

Heap Sort Values \{5,1,4,3,6,2\}
• Heap Sort
1. Add all Values to the Heap
2. Remove All Values from the Heap
3. DONE!
• Heap Sort
1. Add all Values to the Heap
2. Remove All Values from the Heap
3. DONE!

Heap Sort Values \{5,1,4,3,6,2\}

Returned Values:
Heap Sort

1. Add all Values to the Heap
2. Remove All Values from the Heap
3. DONE!

Heap Sort Values \(\{5,1,4,3,6,2\}\)

Returned Values: 1,
• Heap Sort
  1. Add all Values to the Heap
  2. Remove All Values from the Heap
  3. DONE!

Heap Sort Values \{5,1,4,3,6,2\}

Returned Values: 1,
• Heap Sort
  1. Add all Values to the Heap
  2. Remove All Values from the Heap
  3. DONE!

Heap Sort Values \{5, 1, 4, 3, 6, 2\}

Returned Values: 1, 2
• Heap Sort
1. Add all Values to the Heap
2. Remove All Values from the Heap
3. DONE!

Heap Sort Values \{5,1,4,3,6,2\}

Returned Values: 1, 2
Heap Sort
1. Add all Values to the Heap
2. Remove All Values from the Heap
3. DONE!

Heap Sort Values \{5,1,4,3,6,2\}

Returned Values: 1, 2
• Heap Sort
  1. Add all Values to the Heap
  2. Remove All Values from the Heap
  3. DONE!

Heap Sort Values \{5, 1, 4, 3, 6, 2\}

Returned Values: 1, 2, 3
Heap Sort
1. Add all Values to the Heap
2. Remove All Values from the Heap
3. DONE!

Heap Sort Values \{5, 1, 4, 3, 6, 2\}

Returned Values: 1, 2, 3
• Heap Sort
  1. Add all Values to the Heap
  2. Remove All Values from the Heap
  3. DONE!

Heap Sort Values \{5,1,4,3,6,2\}

Returned Values: 1, 2, 3, 4
• Heap Sort
1. Add all Values to the Heap
2. Remove All Values from the Heap
3. DONE!

Heap Sort Values \{5,1,4,3,6,2\}

Returned Values: 1, 2, 3, 4
Heap Sort
1. Add all Values to the Heap
2. Remove All Values from the Heap
3. DONE!

Heap Sort Values \{5,1,4,3,6,2\}

Returned Values: 1, 2, 3, 4, 5
Heap Sort

1. Add all Values to the Heap
2. Remove All Values from the Heap
3. DONE!

Heap Sort Values \{5, 1, 4, 3, 6, 2\}

Returned Values: 1, 2, 3, 4, 5, 6
Heap Sort

1. Add all Values to the Heap
2. Remove All Values from the Heap
3. DONE!

Heap Sort Values \{5,1,4,3,6,2\}

DONE!

Returned Values: 1, 2, 3, 4, 5, 6
### Heap Sort Complexity

<table>
<thead>
<tr>
<th>Worst Case</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorted in Descending Order</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operations</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Add all $n$ values</td>
<td></td>
</tr>
<tr>
<td>Remove all $n$ values</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Special Consideration</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Heaps are always balanced trees</td>
<td></td>
</tr>
<tr>
<td>Worst Case</td>
<td>Complexity</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>Sorted in Descending Order</td>
<td>$O(n\log n)$</td>
</tr>
<tr>
<td>Operations</td>
<td></td>
</tr>
<tr>
<td>Add all $n$ values</td>
<td></td>
</tr>
<tr>
<td>Remove all $n$ values</td>
<td></td>
</tr>
<tr>
<td>Special Consideration</td>
<td></td>
</tr>
<tr>
<td>Heaps are always balanced trees</td>
<td></td>
</tr>
</tbody>
</table>