Heaps & Priority Queues

Fawzi Emad
Chau-Wen Tseng

Department of Computer Science
University of Maryland, College Park

Overview

- Binary trees
  - Perfect
  - Complete
- Heaps
- Priority queues
Perfect Binary Tree

For binary tree with height $h$
- All nodes at levels $h-1$ or less have 2 children (full)

Complete Binary Trees

For binary tree with height $h$
- All nodes at levels $h-2$ or less have 2 children (full)
- All leaves on level $h$ are as far left as possible
Complete Binary Trees

Note: definition in book is incorrect

Two key properties
- Complete binary tree
- Value at node
  - Smaller than or equal to values in subtrees

Example heap
- \( X \leq Y \)
- \( X \leq Z \)
Heap & Non-heap Examples

Heaps

Non-heaps

Heap Properties

- **Key operations**
  - Insert (X)
  - getSmallest()

- **Key applications**
  - Heapsort
  - Priority queue
Heap Operations – Insert( X )

- **Algorithm**
  1. Add X to end of tree
  2. While (X < parent)
     - Swap X with parent // X bubbles up tree

- **Complexity**
  - # of swaps proportional to height of tree
  - $O(\log(n))$

---

Heap Insert Example

**Insert ( 20 )**

1) Insert to end of tree
2) Compare to parent, swap if parent key larger
3) Insert complete
Heap Insert Example

1) Insert to end of tree
2) Compare to parent, swap if parent key larger
3) Insert complete

Heap Operation – getSmallest()

Algorithm
1. Get smallest node at root
2. Replace root with X at end of tree
3. While ( X > child )
   Swap X with smallest child  // X drops down tree
4. Return smallest node

Complexity
- # swaps proportional to height of tree
- $O(\log(n))$
Heap GetSmallest Example

getSmallest ()

1) Replace root with end of tree
2) Compare node to children, if larger swap with smallest child
3) Repeat swap if needed
Heap Implementation

- Can implement heap as array
  - Store nodes in array elements
  - Assign location (index) for elements using formula

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>X0</td>
<td>X1</td>
<td>X2</td>
<td>X3</td>
<td>X4</td>
<td>X5</td>
</tr>
</tbody>
</table>

(a) Heap represented as a tree       (b) Heap represented as an array

Array X

Heap Implementation

- Observations
  - Compact representation
  - Edges are implicit (no storage required)
  - Works well for complete trees (no wasted space)
Heap Implementation

Calculating node locations
- Array index \( i \) starts at 0
- \( \text{Parent}(i) = \left\lfloor \frac{(i - 1)}{2} \right\rfloor \)
- \( \text{LeftChild}(i) = 2 \times i + 1 \)
- \( \text{RightChild}(i) = 2 \times i + 2 \)

Example
- \( \text{Parent}(1) = \left\lfloor \frac{(1 - 1)}{2} \right\rfloor = \left\lfloor \frac{0}{2} \right\rfloor = 0 \)
- \( \text{Parent}(2) = \left\lfloor \frac{(2 - 1)}{2} \right\rfloor = \left\lfloor \frac{1}{2} \right\rfloor = 0 \)
- \( \text{Parent}(3) = \left\lfloor \frac{(3 - 1)}{2} \right\rfloor = \left\lfloor \frac{2}{2} \right\rfloor = 1 \)
- \( \text{Parent}(4) = \left\lfloor \frac{(4 - 1)}{2} \right\rfloor = \left\lfloor \frac{3}{2} \right\rfloor = 1 \)
- \( \text{Parent}(5) = \left\lfloor \frac{(5 - 1)}{2} \right\rfloor = \left\lfloor \frac{4}{2} \right\rfloor = 2 \)
Heap Implementation

Example
- \text{LeftChild}(0) = 2 \times 0 + 1 = 1
- \text{LeftChild}(1) = 2 \times 1 + 1 = 3
- \text{LeftChild}(2) = 2 \times 2 + 1 = 5

![Heap Implementation Diagram]

Heap Implementation

Example
- \text{RightChild}(0) = 2 \times 0 + 2 = 2
- \text{RightChild}(1) = 2 \times 1 + 2 = 4

![Heap Implementation Diagram]
Heap Application – Heapsort

- Use heaps to sort values
  - Heap keeps track of smallest element in heap

- Algorithm
  1. Create heap
  2. Insert values in heap
  3. Remove values from heap (in ascending order)

- Complexity
  - $O(n \log(n))$

Heapsort Example

- Input
  - 11, 5, 13, 6, 1

- View heap during insert, removal
  - As tree
  - As array
Heapsort – Insert Values

(a) Insert 11

(b) Insert 5

(c) Rebuild heap

(d) Insert 13

(e) Insert 6

(f) Rebuild heap

(g) Insert 1

(h) Rebuild heap

Heapsort – Remove Values

(a) Print root = 1

(b) Rebuild heap

(c) Print root = 5

(d) Rebuild heap

(e) Print root = 6

(f) Rebuild heap

(g) Print root = 11

(h) Rebuild heap

(f) Print root = 13

Done
Heapsort – Insert into Array 1

Input
- 11, 5, 13, 6, 1

<table>
<thead>
<tr>
<th>Index =</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert 11</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Heapsort – Insert into Array 2

Input
- 11, 5, 13, 6, 1

<table>
<thead>
<tr>
<th>Index =</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert 5</td>
<td>11</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swap</td>
<td>5</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Heapsort – Insert in to Array 3

- **Input**
  - 11, 5, 13, 6, 1

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert 13</td>
<td>5</td>
<td>11</td>
<td>13</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Heapsort – Insert in to Array 4

- **Input**
  - 11, 5, 13, 6, 1

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert 6</td>
<td>5</td>
<td>11</td>
<td>13</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Swap</td>
<td>5</td>
<td>6</td>
<td>13</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

...
### Heapsort – Remove from Array 1

**Input**
- 11, 5, 13, 6, 1

<table>
<thead>
<tr>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Remove root</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Replace</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Swap w/ child</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

### Heapsort – Remove from Array 2

**Input**
- 11, 5, 13, 6, 1

<table>
<thead>
<tr>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Remove root</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Replace</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>13</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Swap w/ child</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>13</td>
</tr>
</tbody>
</table>
Heap Application – Priority Queue

Queue
- Linear data structure
- First-in First-out (FIFO)
- Implement as array / linked list

Priority queue
- Elements are assigned priority value
- Higher priority elements are taken out first
- Equal priority elements are taken out in FIFO order
- Implement as heap

Priority Queue

Properties
- Lower value = higher priority
- Heap keeps highest priority items in front

Complexity
- Enqueue (insert) = $O(\log(n))$
- Dequeue (remove) = $O(\log(n))$
- For any heap
Heap vs Binary Tree

**Binary tree**
- Keeps values in sorted order
- Find any value
  - $O( \log(n) )$ for balanced tree
  - $O( n )$ for degenerate tree (worst case)

**Heap**
- Keeps smaller values in front
- Find minimum value
  - $O( \log(n) )$ for any heap
- Can also organize heap to find maximum value
  - Keep largest value in front