Heaps & Priority Queues

Nelson Padua-Perez
Bill Pugh
Department of Computer Science
University of Maryland, College Park
Overview

- Heaps
- Priority queues
Heaps

Two key properties
- Heap shape
- 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**
- Add X to end of tree
- 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

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

- Insert (8)

1) Insert to end of tree
2) Compare to parent, swap if parent key larger
3) Insert complete
Heap Operation – getSmallest()

Algorithm

- Get smallest node at root
- Replace root with X at end of tree
- While ( X > child )
  - Swap X with smallest child  // X drops down tree
- 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 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
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
- Parent($i$) = $\lfloor (i - 1) / 2 \rfloor$
- LeftChild($i$) = $2 \times i + 1$
- RightChild($i$) = $2 \times i + 2$

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

- Parent(1) = ⌊(1 - 1) / 2⌋ = ⌊0 / 2⌋ = 0
- Parent(2) = ⌊(2 - 1) / 2⌋ = ⌊1 / 2⌋ = 0
- Parent(3) = ⌊(3 - 1) / 2⌋ = ⌊2 / 2⌋ = 1
- Parent(4) = ⌊(4 - 1) / 2⌋ = ⌊3 / 2⌋ = 1
- Parent(5) = ⌊(5 - 1) / 2⌋ = ⌊4 / 2⌋ = 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

Example

- RightChild(0) = 2 × 0 + 2 = 2
- RightChild(1) = 2 × 1 + 2 = 4
Heap Application – Heapsort

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

- Algorithm
  - Create heap
  - Insert values in heap
  - 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 in to Array 1

Input
11, 5, 13, 6, 1

Index = 0 1 2 3 4
Insert 11: 11 11
Heapsort – Insert in to Array 2

Input
11, 5, 13, 6, 1

Index =

<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>

---

**Note:** The diagram shows the insertion of 13 into the array, maintaining the heap property. The table illustrates the index positions of the elements after the insertion.
Heapsort – Insert in to Array 4

Input
11, 5, 13, 6, 1

Index = 0 1 2 3 4

Insert 6
5 11 13 6

Swap
5 6 13 11
**Heapsort – Remove from Array 1**

**Input**

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>Remove root</td>
<td>1</td>
<td>5</td>
<td>13</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>Replace</td>
<td>6</td>
<td>5</td>
<td>13</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>Swap w/ child</td>
<td>5</td>
<td>6</td>
<td>13</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>
Heapsort – Remove from Array 2

Input

11, 5, 13, 6, 1

Index = 0 1 2 3 4

Remove root

5 6 13 11

Replace

11 6 13

Swap w/ child

6 11 13
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 arbitrary 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 Search Tree

- **Binary search 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