CMSC 132: Object-Oriented Programming II

Heaps & Priority Queues

Department of Computer Science
University of Maryland, College Park

Overview

- Binary trees
  - Full, Perfect, Complete
- Heaps
  - Insert
  - getSmallest
- Heap applications
  - Heapsort
  - Priority queues
Full Binary Tree
- Binary tree where all nodes have 0 or 2 children

Perfect Binary Tree
- A full binary tree where
  - All leaves at level $h$ for tree of height $h$
Complete Binary Trees

An binary tree (height h) where
- Perfect tree to level h-1
- Leaves at level h are as far left as possible

h = 1

h = 2

h = 3

Complete Binary Trees

Basic complete tree shape

Not Allowed
Heaps

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

- Example heap

```
 X
 / \ 
Y   Z
```

X ≤ Y
X ≤ Z

Heap & Non-heap Examples

Heaps

- 5
- 5
- 45
- 6
- 22
- 25
- 8
- 45
- 22
- 25

Non-heaps

- 5
- 8
- 45
- 6
- 22
- 25
- 8
- 45
- 22
- 25
Heap Properties

- Heaps are balanced trees
  - Height = \( \log_2(n) = O(\log(n)) \)

- Can find smallest element easily
  - Always at top of heap!

- Can organize heap to find maximum value
  - Value at node larger than values in subtrees
  - Heap can track either min or max, but not both

Heap

- 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

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

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

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($i$) = ⌊$(i - 1) / 2$⌋
- LeftChild($i$) = $2 \times i + 1$
- RightChild($i$) = $2 \times i + 2$

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
- LeftChild(0) = 2 \times 0 + 1 = 1
- LeftChild(1) = 2 \times 1 + 1 = 3
- LeftChild(2) = 2 \times 2 + 1 = 5

Heap Implementation

Example
- RightChild(0) = 2 \times 0 + 2 = 2
- RightChild(1) = 2 \times 1 + 2 = 4
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
11
(b) Insert 5
5
(c) Rebuild heap
11
5

(d) Insert 13
5
11 13
(e) Insert 6
5
11 13
(f) Rebuild heap
6
11 13

(g) Insert 1
5
6 13
11 1
(h) Rebuild heap
5
1 13
11 6

Heapsort – Remove Values

(a) Print root = 1
1
5 13
11 6
(b) Rebuild heap
6
11 11
5 13 6
(c) Print root = 5
5
11 11
5 13

(d) Rebuild heap
6
11 11
5 13
(e) Print root = 6
6
11 13
11 13
(f) Rebuild heap
13
11 13

(g) Print root = 11
11
13
13
(h) Rebuild heap
13
(i) Print root = 13
13
Done
**Heapsort – Insert in to 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 in to 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

**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 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

**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 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

Index = 0 1 2 3 4

Remove root 1 5 13 11 6

Replace 6 5 13 11

Swap w/ child 5 6 13 11

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 FIFO order
- Implement as heap
  - Enqueue ⇒ insert( )
  - Dequeue ⇒ getSmallest( )
Priority Queue

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

Complexity
- Enqueue ⇒ insert( ) = O( log(n) )
- Dequeue ⇒ getSmallest( ) = 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