CMSC 132: OBJECT-ORIENTED PROGRAMMING II

Heaps & Priority Queues

Department of Computer Science
University of Maryland, College Park
Overview

• Binary trees
  • Complete
• Heaps
  • Insert
  • getSmallest
• Heap applications
  • Heapsort
  • Priority queues
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
  - \( X \leq Y \)
  - \( X \leq Z \)
Heap & Non-heap Examples

Heaps

Non-heaps
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
  - 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

- Insert (20)

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

(a) Heap represented as a tree
(b) Heap represented as an array
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
  - $\text{Parent}(i) = \left\lfloor \frac{(i - 1)}{2} \right\rfloor$
  - $\text{LeftChild}(i) = 2 \times i + 1$
  - $\text{RightChild}(i) = 2 \times i + 2$

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

- 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 × 0 +1 = 1
  - LeftChild(1) = 2 × 1 +1 = 3
  - LeftChild(2) = 2 × 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
  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
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 = 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>
</tr>
</tbody>
</table>
Heapsort – Insert in to Array 2

• Input
  • 11, 5, 13, 6, 1

Index = 0 1 2 3 4

Insert 5 11 5

Swap 5 11
Heapsort – Insert into 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

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

<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>5</td>
<td>6</td>
<td>13</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>Replace</td>
<td></td>
<td>11</td>
<td>6</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>Swap w/ child</td>
<td></td>
<td>6</td>
<td>11</td>
<td>13</td>
<td></td>
</tr>
</tbody>
</table>
Heap Application – Priority Queue

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

- Priority queue
  - Elements are assigned priority value
  - Higher priority elements are taken out first
  - Implement as heap
    - Enqueue ⇒ \texttt{insert}()
    - Dequeue ⇒ \texttt{getSmallest}()
Priority Queue

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

• Complexity
  • Enqueue $\Rightarrow \text{insert}( ) = O(\log(n))$
  • Dequeue $\Rightarrow \text{getSmallest}( ) = O(\log(n))$
  • 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