Heaps II

Lecture 23

Prof. Eric Polizzi
Summary previous lecture

- **Priority queue**
  - **Problem**: using an ordered array, dequeue (removal) of priority item (for example: largest or smallest item) is $O(1)$ but enqueue (insertion) of items is $O(N)$
  - **Solution**: use a heap implementation that offers $O(\log N)$ for both
  - Heap is a ADT (Heap == CBT+heap condition)
  - **heap condition**: *every node's key is larger than (or equal to) the keys of its children.*
  - Priority queue ← Heap == (CBT + heap cond.) ← array implementation
Summary previous lecture

- Removal of priority item

Step 1: remove root

Step 2: last node into root

Step 3: trickle down root

```java
public Node remove()
{
    Node root = heapArray[0];
    heapArray[0] = heapArray[--N];
    trickleDown(0);
    return root;
}
```
Insertion of new item

- Step 1: add new node (first open position)

```
public boolean insert(int key) {
    if (N==maxSize) return false;
    Node newNode=new Node(key);
    heapArray[N] = newNode;
    trickleUp(N++);
    return true;
}
```

- Step 2: trickle up new node

```
private void trickleUp(int i) {
    // reach top of heap - done
    if (i==0) return;
    // check if parent is smaller
    int parent=(i-1)/2;
    if (heapArray[i]>heapArray[parent]) {
        swap(i,parent);
        //swap
        trickleUp(parent); //recursion
    }
}
```
How to Heapify an array?

1st approach - Using successive insertions

O(NlogN) 
N trickleup
How to Heapify an array?

2nd approach – Trickledown in place

- At first all the items can be placed in random locations in the array
- They can then be rearranged into a heap using N/2 applications of trickleDown... some speed advantage compared to 1st approach

- No need to Trickledown last row
- Start at node (N/2-1) and works backward

```java
for (int i=N/2-1; i>=0; i--)
    trickleDown(i);
```
How to Heapify an array?

2nd approach – Trickledown in place

O(NlogN)
N/2 trickledown
Heapify - using the same array

8 20 33 10 7 2 6 15 19 27

8 20 33 10 7 2 6 15 19 27

8 20 33 10 7 2 6 15 19 27

8 20 33 10 7 2 6 15 19 27

8 27 33 19 20 2 6 15 10 7

33 27 8 19 20 2 6 15 10 7
Heapsort

- **Review: linked-list insertion sort**
  - Insert items from a random array to a sorted list, and copy back (easy concept)
  - Cost $O(N^2)$: $O(N^2)$ comparisons but only $2N$ copies, does not work in place (requires x2 memory)

- **Motivations for Heapsort:**
  - It is not difficult to Heapify a random array $\rightarrow$ cost is $O(N\log N)$
  - Removing one priority item is $O(\log N)$
    
    **Remark:** once root item is replaced by last node, only one trickledown is enough to heapify the new data
  - Removing $N$ times $\rightarrow O(N\log N)$ sorting algorithms
  - Can work in place (no additional memory needed)
  - Can be a bit slower than QuickSort but it runs in $O(N\log N)$ no matter how the data is distributed
Heapsort - successive removals
Heapsort - using the same array

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>27</td>
<td>8</td>
<td>19</td>
<td>20</td>
<td>2</td>
<td>6</td>
<td>15</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>27</td>
<td>20</td>
<td>8</td>
<td>19</td>
<td>7</td>
<td>2</td>
<td>6</td>
<td>15</td>
<td>10</td>
<td>33</td>
</tr>
<tr>
<td>20</td>
<td>19</td>
<td>8</td>
<td>15</td>
<td>7</td>
<td>2</td>
<td>6</td>
<td>10</td>
<td>27</td>
<td>33</td>
</tr>
<tr>
<td>19</td>
<td>15</td>
<td>8</td>
<td>10</td>
<td>7</td>
<td>2</td>
<td>6</td>
<td>20</td>
<td>27</td>
<td>33</td>
</tr>
<tr>
<td>15</td>
<td>10</td>
<td>8</td>
<td>6</td>
<td>7</td>
<td>2</td>
<td>19</td>
<td>20</td>
<td>27</td>
<td>33</td>
</tr>
<tr>
<td>10</td>
<td>7</td>
<td>8</td>
<td>6</td>
<td>2</td>
<td>15</td>
<td>19</td>
<td>20</td>
<td>27</td>
<td>33</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>2</td>
<td>6</td>
<td>10</td>
<td>15</td>
<td>19</td>
<td>20</td>
<td>27</td>
<td>33</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>2</td>
<td>8</td>
<td>10</td>
<td>15</td>
<td>19</td>
<td>20</td>
<td>27</td>
<td>33</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>7</td>
<td>8</td>
<td>10</td>
<td>15</td>
<td>19</td>
<td>20</td>
<td>27</td>
<td>33</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>10</td>
<td>15</td>
<td>19</td>
<td>20</td>
<td>27</td>
<td>33</td>
</tr>
</tbody>
</table>
for (int j=N-1; j>=0; j--)
{
    Node priorityNode=remove();
    heapArray[j] = priorityNode;
}