Priority Queues, Heaps, and Heapsort

CSE 2320 – Algorithms and Data Structures
Alexandra Stefan
(includes slides from Vassilis Athitsos)
University of Texas at Arlington

Last modified: 2/27/2017
Overview

• Priority queue
  – A data structure that on delete/remove removes the item with the HIGHEST priority
  – Implementation (supporting data structures)
    • list/array (sorted/unsorted)
    • heap

• Heap
  – Definition, properties, operations (insert, delete, create)
  – Building a heap (bottom-up and top-down)
  – Heapsort
  – Finding top k
  – Index items – the heap has the index of the element

• Heapsort
Priority Queues

• Goal – to support (efficiently):
  – Insertion of a new element.
  – Deletion of the max element.
  – Initialization (organizing an initial set of data).

• Useful for **online** processing
  – When we do not have all the data at once (the data keeps coming or changing).

(So far we have seen sorting methods that work in **batch mode**:
  • They are given all the items at once, they sort the items, Done!)

• Applications:
  – Scheduling:
    • flights take-off and landing, programs executed (CPU), database queries
  – Waitlists:
    • patients in a hospital, student admission
  – Graph algorithms (part of MST)
Arrays and Lists as Priority Queues

Arrays and lists (sorted or unsorted) can be used as priority queues, but they require $O(N)$ for either insert or delete max.

<table>
<thead>
<tr>
<th>Data structure</th>
<th>Insert</th>
<th>Delete max</th>
<th>Create from batch of N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array unsorted</td>
<td>$\Theta(1)$</td>
<td>$\Theta(N)$</td>
<td>$\Theta(N)$</td>
</tr>
<tr>
<td>List unsorted</td>
<td>$\Theta(1)$</td>
<td>$\Theta(N)$</td>
<td>$\Theta(N)$</td>
</tr>
<tr>
<td>Array sorted</td>
<td>$O(N)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(NlgN)$ (e.g. mergesort)</td>
</tr>
<tr>
<td>List sorted</td>
<td>$O(N)$ (find correct position)</td>
<td>$\Theta(1)$</td>
<td>$\Theta(NlgN)$ (e.g. mergesort)</td>
</tr>
</tbody>
</table>
Priority Queues and Sorting

• Sorting with a heap:
  – Given items to sort:
  – Create a priority queue that contains those items.
  – Initialize result to empty list.
  – While the priority queue is not empty:
    • Remove max element from queue and add it to beginning of result.

• Heapsort – $\Theta(N \lg N)$
  – builds the heap in $O(N)$. 
Heap Operations

• Initialization:
  – Given N-size array, heapify it.
  – Time: $\Theta(N)$. Good!

• Insertion of a new item:
  – Requires rearranging items, to maintain the heap property.
  – Time: $O(\lg N)$. Good!

• Deletion/removal of the largest element (max-heap):
  – Requires rearranging items, to maintain the heap property.
  – Time: $O(\lg N)$. Good!

• Min-heap is similar.
Heap

• Intuition
  – Lists and arrays: not fast enough => Try a tree (‘fast’ if ‘balanced’).
  – Want to remove the max fast => keep it in the root
  – Keep the tree balanced after insert and delete (it does not degenerate to a list)

• Specific heap properties:
  – Every node, N, is larger than or equal to any of his children (their keys).
    • => root has the largest key
  – Complete tree:
    • All levels are full except for possibly the last one
    • If the last level is not full, all nodes are leftmost (no ‘holes’).

• This tree can be represented by an array, A.
  – Root stored at index 1,
  – Node at index i has left child at 2i, right child at 2i+1 and parent at $\left\lfloor i/2 \right\rfloor$
A valid heap will always have these properties. (Also called invariants.) They will be preserved even after delete and insert.

**P1: Order:** Every node, N, is larger than or equal to any of its children.
- => Max is in the root.
- => Any path from root to a node (and leaf) will go through nodes that have decreasing value/priority. E.g.: 9,7,5,1 (blue path), or 9,5,4,4

**P2: Shape** (complete tree)
- All levels are full except for possibly the last level.
  - => Heap height = \( \lfloor \log N \rfloor \)
  - => If height h => \( 2^h \leq N \leq 2^{h+1} - 1 \)
- On last level all nodes are rightmost:
Binary Heap: Array Representation

Practice:
- Tree -> Array
- Array -> Tree

We can represent the tree as an array:

<table>
<thead>
<tr>
<th>index</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>9</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Root is at index 1. (At index 0: no data or put the heap size there.)

Node at index i

<table>
<thead>
<tr>
<th>Children</th>
<th>Parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left 2i</td>
<td>[i/2]</td>
</tr>
<tr>
<td>Right 2i+1</td>
<td></td>
</tr>
</tbody>
</table>
Heap – Shape Property

• A binary tree representing a heap has to be **complete**:
  – All levels are full, except possibly for the last level.
  – At the last level:
    • Nodes are placed on the left.
    • Empty positions are placed on the right.
  – There is “no hole”
Heap Practice

For each tree, say if it is a max heap or not.

E2

E3

E4

E5
Answers

For each tree, say if it is a max heap or not.

E1

E2

E3

E4

E5
Examples and Exercises

• Invalid heaps
  – Order property violated
  – Shape property violated (‘tree with holes’)

• Valid heaps (that may seem strange)
  – Same key in node and one or both children
  – ‘Extreme’ heaps (all nodes in the left child are smaller than any node in the right child or vice versa)
    – **Min-heaps**

• Where can these elements be found?
  – Largest element?
  – 2-nd largest?
  – 3-rd largest?
Heap Operations

- Max-Heapify(A, i) (fix-up/swim-up)
- Build-Max_Heap(A) (build heap bottom-up)
- Heapsort(A)
- Heap-based Priority Queues
- Top-k
- Handles
Max-Heapify(A,i)  
(fix-up/swim-up)

• USED
  – When the key value decreased.
  – To make tree rooted at i be a heap.
    • Assumes the subtrees rooted at Left(i) and Right(i) are heaps.

• How:
  – Repeatedly exchange items as needed, between a node and his largest child, starting at i.

• Example:
  – X was a B (or decreased to B).
Max-Heapify

• B will move down until in a good position.

• Exchange B and T.
Max-Heapify

• B will move down until in a good position.

• Exchange B and T.

• Exchange B and S.
Max-Heapify

- B will move down until in a good position.
- Exchange B and T.
- Exchange B and S.
- Exchange B and R.
Max-Heapify(A,i)

Left(i)-left child of node i
Right(i)-right child of node i

Assumes Left(i) and Right(i) are heaps, but A[i] may be smaller than its children (thus not a heap).

Max-Heapify(A, i)

1. \( l = \text{Left}(i) \)
2. \( r = \text{Right}(i) \)
3. if \( l \leq A.\text{heap-size} \) and \( A[l] > A[i] \)
   4. \( \text{largest} = l \)
5. else \( \text{largest} = i \)
6. if \( r \leq A.\text{heap-size} \) and \( A[r] > A[\text{largest}] \)
   7. \( \text{largest} = r \)
8. if \( \text{largest} \neq i \)
   9. exchange \( A[i] \) with \( A[\text{largest}] \)
10. \( \text{Max-Heapify}(A, \text{largest}) \)
Batch Initialization

• Batch initialization of a heap
  – The process of converting an unsorted array of data into a heap.
  – We will see 2 methods:
    • top-down and
    • bottom-up.

<table>
<thead>
<tr>
<th>Batch Initialization Method</th>
<th>Time</th>
<th>Extra space (in addition to the space of the input array)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top-down (insert in new array one by one)</td>
<td>$O(N \lg N)$</td>
<td>$\Theta(N)$</td>
</tr>
<tr>
<td>Bottom-up (fix the given array)</td>
<td>$O(N)$</td>
<td>$\Theta(1)$</td>
</tr>
</tbody>
</table>
Bottom-Up Batch Initialization

Turns array A into a heap in O(N).
(N = number of elements of A)

\[
\text{BUILD-MAX-HEAP}(A) \\
1 \quad A.\text{heap-size} = A.\text{length} \\
2 \quad \text{for } i = \lceil A.\text{length}/2 \rceil \text{ downto 1} \\
3 \quad \text{MAX-HEAPIFY}(A, i)
\]

- See animation: https://www.cs.usfca.edu/~galles/visualization/HeapSort.html
  - Note that they do not highlight the node being processed, but directly the children of it as they are compared to find the larger one of them.
Bottom-Up - Example

• Convert the given array to a heap using bottom-up:
  (must work in place):
  5, 3, 20, 15, 7, 12, 9, 14, 8, 11.
Running Time

• How can we analyze the running time?

• To simplify, suppose that last level if complete: => \( N = 2^n - 1 \) (=> last level is \((n-1)\) => heap height is \((n-1) = \log N\) ) (see next slide)

• Counter \( i \) starts at value \( 2^{n-1} - 1 \).
  – That gives the last node on level \( n-2 \).
  – At that point, we call fixDown on a heap of height 1.
  – For all the \( 2^{n-2} \) nodes at this level, we call fixDown on a heap of height 1 (nodes at this level are at indexes \( i \) s.t. \( 2^{n-1}-1 \geq i \geq 2^{n-2} \)).

…..

• When \( i \) is 1 (=\( 2^0 \)) we call Max-Heapify on a heap of height \( n-1 \).

```
BUILD-MAX-HEAP(A)
1   A.heap-size = A.length
2   for i = [A.length/2] downto 1
3       MAX-HEAPIFY(A, i)
```
A perfect binary tree with \( N \) nodes has:

- \( \lceil \lg N \rceil + 1 \) levels
- Height \( \lceil \lg N \rceil \)
- \( \lceil N/2 \rceil \) leaves (half the nodes are on the last level)
- \( \lceil N/2 \rceil \) internal nodes (half the nodes are internal)

\[
\sum_{k=0}^{n-1} 2^k = 2^n - 1
\]

<table>
<thead>
<tr>
<th>Level</th>
<th>Nodes per level</th>
<th>Sum of nodes from root up to this level</th>
<th>Heap height</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( 2^0 ) (=1)</td>
<td>( 2^1 - 1 ) (=1)</td>
<td>n-1</td>
</tr>
<tr>
<td>1</td>
<td>( 2^1 ) (=2)</td>
<td>( 2^2 - 1 ) (=3)</td>
<td>n-2</td>
</tr>
<tr>
<td>2</td>
<td>( 2^2 ) (=4)</td>
<td>( 2^3 - 1 ) (=7)</td>
<td>n-3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>( 2^i )</td>
<td>( 2^{i+1} - 1 )</td>
<td>n-1-i</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n-2</td>
<td>( 2^{n-2} )</td>
<td>( 2^{n-1} - 1 )</td>
<td>1</td>
</tr>
<tr>
<td>n-1</td>
<td>( 2^{n-1} )</td>
<td>( 2^n - 1 )</td>
<td>0</td>
</tr>
</tbody>
</table>
Running Time: $O(N)$

<table>
<thead>
<tr>
<th>Counter from:</th>
<th>Counter to:</th>
<th>Level</th>
<th>Nodes per level</th>
<th>Height of heaps rooted at these nodes</th>
<th>Time per node (fixDown)</th>
<th>Time for fixing all nodes at this level</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{n-2}$</td>
<td>$2^{n-1} - 1$</td>
<td>n-2</td>
<td>$2^{n-2}$</td>
<td>1</td>
<td>$O(1)$</td>
<td>$O(2^{n-2} * 1)$</td>
</tr>
<tr>
<td>$2^{n-3}$</td>
<td>$2^{n-2} - 1$</td>
<td>n-3</td>
<td>$2^{n-3}$</td>
<td>2</td>
<td>$O(2)$</td>
<td>$O(2^{n-3} * 2)$</td>
</tr>
<tr>
<td>$2^{n-4}$</td>
<td>$2^{n-3} - 1$</td>
<td>n-4</td>
<td>$2^{n-4}$</td>
<td>3</td>
<td>$O(3)$</td>
<td>$O(2^{n-4} * 3)$</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$2^0 = 1$</td>
<td>$2^1 - 1 = 1$</td>
<td>0</td>
<td>$2^0 = 1$</td>
<td>n – 1</td>
<td>$O(n-1)$</td>
<td>$O(2^0 * (n-1))$</td>
</tr>
</tbody>
</table>

- To simplify, assume: $N = 2^n - 1$.
- The analysis is a bit complicated. Pull out $2^{n-1}$ gives: $2^{n-1} \sum_{k=1}^{n-1} kx^k \leq \sum_{k=1}^{\infty} kx^k \rightarrow 2^{n-1} \frac{x}{(1-x)^2}$ for $x = \frac{1}{2}$ because $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$, for $|x| < 1$.
- Total time: sum over the rightmost column: $O(2^{n-1}) \Rightarrow O(N)$ (linear!)
Heapsort

In the pseudocode below:

• Where is the sorted data?
• How does the heap get shorter?

HEAPSORT(A)
1 BUILD-MAX-HEAP(A)
2 for i = A.length down to 2
4 A.heap-size = A.heap-size - 1
5 MAX-HEAPIFY(A, 1)

Give an example that takes $\Theta(N \lg N)$. Give an example that takes $\Theta(N)$ (extreme case: all equal).
Is Heapsort stable?

• For quick reference:
  – If both children are the same value, and the parent value (9) needs to move down, the value from which child will be promoted (8a or 8b)?)

```
MAX-HEAPIFY(A, i)
1  l = LEFT(i)
2  r = RIGHT(i)
4    largest = l
5  else largest = i
7    largest = r
8  if largest ≠ i
9    exchange A[i] with A[largest]
10   MAX-HEAPIFY(A, largest)
```

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
4  A.heap-size = A.heap-size – 1
5  MAX-HEAPIFY(A, 1)
```
Is Heapsort stable? - NO

- Both of these operations are unstable:
  - Max-Heapify
  - Going from the built heap to the sorted array (remove max and put at the end)

- For quick reference:
  - If both children are the same value, and the parent value (9) needs to move down, the value from which child will be promoted (8a or 8b)? Ans: left child

```
MAX-HEAPIFY(A, i)
1  l = LEFT(i)
2  r = RIGHT(i)
4    largest = l
5  else largest = i
7    largest = r
8  if largest != i
9    exchange A[i] with A[largest]
10   MAX-HEAPIFY(A, largest)
```

HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
4  A.heap-size = A.heap-size - 1
5  MAX-HEAPIFY(A, 1)
Is Heapsort Stable? - No

- No. The sorted array is built from the end => items get reversed.
- Example (data comes out of the heap in the right order, it still gets flipped due to moving max to end (see the array above the heap drawing)):
  - Input array: A = [9, 8a, 8b, 5] (8a and 8b are both 8. a and b are used to indicate their order in the input array).
  - Sorted array: A = [5, 8b, 8a, 9].

Note: in this example, even if the array was a heap to start with, the sorting part (removing max and putting it at the end) causes the sorting to not be stable.
Is Heapsort Stable? - No

• **Max-Heapify is not stable.**
  
  – When a node is swapped with his child, they jump all the nodes in between them (in the array).

• **Example:**
  
  – Input array: \( A = [9, 6, 8a, 5, 8b] \) (8a and 8b are both 8. a and b are used to indicate their order in the input array).
Heap-Based Priority Queues
Heap-Based Priority Queues

• Insert(S,x) – inserts x in S.
• Maximum(S) – returns the element of S with the largest key.
• Extract-Max(S) – removes and returns the element of S with the largest key.
• Increase-Key(S,x,k) – Changes x’s key to be k. Assumes x’s key was initially lower than k.
• Decrease-Key(S,x,k) – Changes x’s key to be k. Assumes x’s key was initially higher than k.
  – Decrease the priority and apply Max-Heapify.
Increasing a Key

• Also called “increasing the priority” of an item.
• Such an operation can lead to violation of the heap property.
• Easy to fix:
  – Exchange items as needed, between node and parent, starting at the node that changed key.

```
T
/  \\
S   R
/    \
G     A
/  \\
E   I
/
A
```

```
Increasing a Key

• Also called “increasing the priority” of an item.
• Such an operation can lead to violation of the heap property.
• Easy to fix:
  – Exchange items as needed, between node and parent, starting at the node that changed key.
• Example:
  – An E changes to a V.
Increasing a Key

• Also called “increasing the priority” of an item.
• Such an operation can lead to violation of the heap property.
• Easy to fix:
  – Exchange items as needed, between node and parent, starting at the node that changed key.
• Example:
  – An E changes to a V.
  – Exchange V and G. Done?

![Heapsort example diagram](image)

- X
- O
- T
- S
- M
- N
- A
- R
- I
- V
- G
Increasing a Key

• Also called “increasing the priority” of an item.
• Such an operation can lead to violation of the heap property.
• Easy to fix:
  – Exchange items as needed, between node and parent, starting at the node that changed key.
• Example:
  – An E changes to a V.
  – Exchange V and G.
  – Exchange V and T. Done?
Increasing a Key

• Also called “increasing the priority” of an item.
• Such an operation can lead to violation of the heap property.
• Easy to fix:
  – Exchange items as needed, between node and parent, starting at the node that changed key.
• Example:
  – An E changes to a V.
  – Exchange V and G.
  – Exchange V and T. Done.
Increasing a Key (fix-up)

**Heap-Increase-Key** \((A, i, key)\)

1. **if** \(key < A[i]\)
2. **error** “new key is smaller than current key”
3. \(A[i] = key\)
4. **while** \(i > 1\) and \(A[\text{Parent}(i)] < A[i]\)
5. exchange \(A[i]\) with \(A[\text{Parent}(i)]\)
6. \(i = \text{Parent}(i)\)
Inserting a New Record

- This is a heap with 12 items.
- How will a heap with 13 items look?
  - Where can the new node be? (do not worry about the data in the nodes now)

<table>
<thead>
<tr>
<th>position</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>T</td>
<td>S</td>
<td>O</td>
<td>G</td>
<td>R</td>
<td>M</td>
<td>N</td>
<td>A</td>
<td>E</td>
<td>B</td>
<td>A</td>
<td>I</td>
</tr>
</tbody>
</table>

- **Table:**

- **Diagram:**

![Diagram](image-url)
Inserting a New Record

- Let’s insert V

<table>
<thead>
<tr>
<th>position</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>T</td>
<td>S</td>
<td>O</td>
<td>G</td>
<td>R</td>
<td>M</td>
<td>N</td>
<td>A</td>
<td>E</td>
<td>B</td>
<td>A</td>
<td>I</td>
<td></td>
</tr>
</tbody>
</table>

```
Let's insert V
```
## Inserting a New Record

Let’s insert V

- Put V in the last position and fix up.

<table>
<thead>
<tr>
<th>position</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>T</td>
<td>S</td>
<td>O</td>
<td>G</td>
<td>R</td>
<td>M</td>
<td>N</td>
<td>A</td>
<td>E</td>
<td>B</td>
<td>A</td>
<td>I</td>
<td>V</td>
</tr>
</tbody>
</table>

![Tree Diagram]
Inserting a New Record

<table>
<thead>
<tr>
<th>position</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>V</td>
<td>S</td>
<td>T</td>
<td>G</td>
<td>R</td>
<td>O</td>
<td>N</td>
<td>A</td>
<td>E</td>
<td>B</td>
<td>A</td>
<td>I</td>
<td>M</td>
</tr>
</tbody>
</table>

- Fixed heap

\[\text{MAX-HEAP-INSERT} (A, key)\]
1. \(A.\text{heap-size} = A.\text{heap-size} + 1\)
2. \(A[A.\text{heap-size}] = -\infty\)
3. \(\text{HEAP-INCREASE-KEY} (A, A.\text{heap-size}.key)\)
Inserting a New Record

• Insertion: Put in the last position and fix up.

• Discuss the runtime for insertion.
  – Let N be the number of nodes in the heap.

<table>
<thead>
<tr>
<th>position</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>T</td>
<td>S</td>
<td>O</td>
<td>G</td>
<td>R</td>
<td>M</td>
<td>N</td>
<td>A</td>
<td>E</td>
<td>B</td>
<td>A</td>
<td>I</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Case</th>
<th>Discussion</th>
<th>Notation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Worst</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>General</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Inserting a New Record

- Insertion: Put in the last position and fix up.

- Discuss the runtime for insertion.
  - Let N be the number of nodes in the heap.

<table>
<thead>
<tr>
<th>position</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>T</td>
<td>S</td>
<td>O</td>
<td>G</td>
<td>R</td>
<td>M</td>
<td>N</td>
<td>A</td>
<td>E</td>
<td>B</td>
<td>A</td>
<td>I</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Case</th>
<th>Discussion</th>
<th>Notation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best</td>
<td>1</td>
<td>$\Theta(1)$</td>
<td>V was M</td>
</tr>
<tr>
<td>Worst</td>
<td>Height of heap</td>
<td>$\Theta(\lg N)$</td>
<td>Shown here</td>
</tr>
<tr>
<td>General</td>
<td></td>
<td>$O(\lg N)$</td>
<td></td>
</tr>
</tbody>
</table>

![Heap Diagram]

44
Remove the Maximum

- This is a heap with 12 items.

- How will a heap with 11 items look like?
  - What node will disappear?
    (Again think about the ‘circles’, not the data in them)

- Where is the record with the highest key?
Remove the Maximum

- Removing the maximum
  - Save the record on top of the heap (e.g. as ret)
  - Copy the last record on the first position (on top)
  - Decrement heap size by 1
  - **Fix-down (Max-Heapify) the top (invalid heap now)**
  - Return the original top record, ret.

- ‘No holes’
  - Because a valid item is placed in root.

<table>
<thead>
<tr>
<th>position</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>I</td>
<td>S</td>
<td>O</td>
<td>G</td>
<td>R</td>
<td>M</td>
<td>N</td>
<td>A</td>
<td>E</td>
<td>B</td>
<td>A</td>
</tr>
</tbody>
</table>

![Tree Diagram]

\[ \text{len} \]
Remove the Maximum

<table>
<thead>
<tr>
<th>position</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>S</td>
<td>R</td>
<td>O</td>
<td>G</td>
<td>I</td>
<td>M</td>
<td>N</td>
<td>A</td>
<td>E</td>
<td>B</td>
<td>A</td>
</tr>
</tbody>
</table>

- **Fixed heap:**
  - Programming trick (alternative):
    - Swap the top with the last
    - ...
    - Return from index len+1

```java
HEAP-EXTRACT-MAX(A)
1 if A.heap-size < 1
2 error "heap underflow"
3 max = A[1]
5 A.heap-size = A.heap-size - 1
6 MAX-HEAPIFY(A, 1)
7 return max
```
Remove the Maximum

- Removing the maximum
  - Save the record on top of the heap (e.g. as max)
  - Copy the last record on the first position (on top)
  - Decrement heap size by 1
  - Fix-down the top
  - Return the original top record, max.

- Discuss the runtime for remove max.

<table>
<thead>
<tr>
<th>Case</th>
<th>Discussion</th>
<th>Notation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best</td>
<td>1</td>
<td>Θ(1)</td>
<td></td>
</tr>
<tr>
<td>Worst</td>
<td>Height of heap</td>
<td>Θ(lgN)</td>
<td>Last node was A, not I</td>
</tr>
<tr>
<td>General</td>
<td>1=&lt;...=&lt;lgN</td>
<td>O(lgN)</td>
<td></td>
</tr>
</tbody>
</table>
Removal of a Non-Max Node

• Replace it with the last node in the heap.

• To restore the heap property: fix up (new priority is larger) or fix down (new priority is smaller).
  – Check which one it is, and fix the heap accordingly.

• Give an example where the new priority is decreased.
Insertions and Deletions - Summary

• Insertion:
  – Insert the item to the end of the heap.
  – Fix up to restore the heap property.
  – Time = $O(lg N)$

• Deletion:
  – Will always delete the maximum element. This element is always at the top of the heap (the first element of the heap).
  – Deletion of the maximum element:
    • Exchange the first and last elements of the heap.
    • Delete the last element (which is the maximum element).
    • Fix down to restore the heap property.
    • Time = $O(lg N)$
Top-Down Batch Initialization

• Build a heap of size N by repeated insertions in an originally empty heap.
  – E.g. build a max-heap from: 5, 3, 20, 15, 7, 12, 9, 14, 8, 11.

• Time complexity? $O(N \lg N)$
  – N insertions performed.
  – Each insertion takes $O(\lg X)$ time.
    • X- current size of heap.
    • X goes from 1 to N.
  – In total, worst (and average) case: $\Theta(N \lg N)$
    • $T(N) = \Theta(N \lg N)$.
      – The last $N/2$ nodes are inserted in a heap of size ($\lg N$)-1. => $T(N) = \Omega(N \lg N)$.
        » Example that results in $\Theta(N)$?
      – Each of the N insertions takes at most $\lg N$. => $T(N) = O(N \lg N)$
Index Heap, Handles

• So far:
  – We assumed that the actual data is stored in the heap.
  – We can increase/decrease priority of any specific node and restore the heap.
• In a real application we need to be able to do more
  – Find a particular record in a heap
    • John Doe got better and leaves. Find the record for John in the heap.
    • (This operation will be needed when we use a heap later for MST.)
  – You cannot put the actual data in the heap
    • You do not have access to the data (e.g. for protection)
    • To avoid replication of the data. For example you also need to frequently search in that data so you also need to organize it for efficient search by a different criteria (e.g. ID number).
# Index Heap Example - Workout

1. Show the heap with this data (fill in the figure on the right based on the **HA array**).  
   1. For each heap node show the corresponding array index as well.

<table>
<thead>
<tr>
<th>Index</th>
<th>HA (H-&gt;A)</th>
<th>AH (A-&gt;H)</th>
<th>Name</th>
<th>Priority</th>
<th>Other data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>2</td>
<td>Aidan</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>4</td>
<td>Alice</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>Cam</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>3</td>
<td>Joe</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>1</td>
<td>Kate</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>Mary</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>7</td>
<td>Sam</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

**HA** – Heap to Array  
**AH** – Array to Heap
Index Heap Example - Solution

HA – Heap to Array
AH – Array to Heap

<table>
<thead>
<tr>
<th>Index</th>
<th>HA (H-&gt;A)</th>
<th>AH (A-&gt;H)</th>
<th>Name</th>
<th>Priority</th>
<th>Other data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>2</td>
<td>Aidan</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>4</td>
<td>Alice</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>Cam</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>3</td>
<td>Joe</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>1</td>
<td>Kate</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>Mary</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>7</td>
<td>Sam</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

Property:
HA(AH(j)) = j  e.g. HA(AH(5)) = 5
AH(HA(j)) = j  e.g. AH(HA(1)) = 1

Decrease Kate’s priority to 1. Update the heap.
To swap nodes \(p_1\) and \(p_2\) in the heap: \(HA[p_1] \leftrightarrow HA[p_2]\), and \(AH[HA[p_1]] \leftrightarrow AH[HA[p_2]]\).
Index Heap Example

Decrease Key – (Kate 20 -> Kate 1)

HA – Heap to Array
AH – Array to Heap

<table>
<thead>
<tr>
<th>Index</th>
<th>HA (H-&gt;A)</th>
<th>AH (A-&gt;H)</th>
<th>Name</th>
<th>Priority</th>
<th>Other data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5-4</td>
<td>2</td>
<td>Aidan</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>4</td>
<td>Alice</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4-5</td>
<td>5</td>
<td>Cam</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>3-1</td>
<td>Joe</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>1-3</td>
<td>Kate</td>
<td>20,1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>Mary</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>7</td>
<td>Sam</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

Property:
HA(AH(j)) = j e.g. HA(AH(5)) = 5
AH(HA(j)) = j e.g. AH(HA(1)) = 1

Decrease Kate’s priority to 1. Update the heap.
To swap nodes 1 and 3 in the heap: HA[1]<>HA[3], and AH[HA[1]] <> AH[HA[3]].
Index Heap Example

Decrease Key - cont

HA – Heap to Array
AH – Array to Heap

<table>
<thead>
<tr>
<th>Index</th>
<th>HA (H-&gt;A)</th>
<th>AH (A-&gt;H)</th>
<th>Name</th>
<th>Priority</th>
<th>Other data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>Aidan</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>Alice</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>Cam</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>Joe</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>1 3 7</td>
<td>5</td>
<td>Kate</td>
<td>20 1</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>Mary</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>Sam</td>
<td>6</td>
</tr>
</tbody>
</table>

Property:
HA(AH(j)) = j  e.g. HA(AH(5)) = 5
AH(HA(j)) = j  e.g. AH(HA(1)) = 1

Continue to fix down 1. Update the heap.
To swap nodes 3 and 7 in the heap:  HA[3]<->HA[7], and AH[HA[3]] <-> AH[HA[7]].
Finding the Top k Largest Elements
Finding the Top $k$ Largest Elements

• Using a max-heap
• Using a min-heap
Finding the Top k Largest Elements

• Assume N elements

• Using a **max-heap**
  – Build max-heap of size N from all elements, then
    • May require extra space if cannot modify the array (build the heap in place and remove k)
  – remove k times
  – Time: \( \Theta(N + k \times \lg N) \)
    • (build heap: \( \Theta(N) \), k delete ops: \( \Theta(k \times \lg N) \) )

• Using a **min-heap**
  – Build a min-heap, H, of size k (from the first k elements).
  – (N-k) times perform both: *insert* and then *delete* in H.
  – After that, all N elements went through this min-heap and k are left so they **must be** the k largest ones.
  – advantage: less space (constant space: k)
  – Version 1: Time: \( \Theta(k + (N - k) \times \lg k) \) (build heap + (N-k) insert & delete)
  – Version 2 (get the top k sorted): Time: \( \Theta(k + N \times \lg k) \) (build heap + (N-k) insert & delete + k delete)
Top k Largest with Max-Heap

• N = 10, k = 3, Input: 5, 3, 12, 15, 7, 34, 9, 14, 8, 11.
  (Find the top 3 largest elements.)

• Method:
  – Build a max heap using bottom-up
  – Delete/remove 3 (=k) times from that heap
    • What numbers will come out?

• Show all the steps (even those for bottom-up build heap).
  Draw the heap as a tree.
Top k Largest with Min-Heap

Workout Sheet

• N = 10, k = 3, Input: 5, 3, 12, 15, 7, 34, 9, 14, 8, 11.
  (Find the top 3 largest elements.)

• Method:
  – Build a min heap using bottom-up from the first 3 (=k) elements: 5, 3, 12
  – Repeat 7 (=N-k) times: one insert (of the next number) and one remove.
Top k Largest with Min-Heap

Answers

• What is left in the min heap are the top 3 largest numbers.
  – If you need them in order of largest to smallest, do 3 remove operations.

• Intuition:
  – the MIN-heap acts as a ‘sieve’ that keeps the largest elements going through it.
Show the actual heaps and all the steps (insert, delete, and steps for bottom-up heap build). Draw the heaps as a tree.

- N = 10, k = 3, Input: 5, 3, 12, 15, 7, 34, 9, 14, 8, 11.
  (Find the top 3 largest elements.)
- Method:
  - Build a min heap using bottom-up from the first 3 (=k) elements: 5, 3, 12
  - Repeat 7 (=N-k) times: one insert (of the next number) and one remove.
Top largest k with MIN-Heap: Show the actual heaps and all the steps (for insert, remove, and even those for bottom-up build heap). Draw the heaps as a tree.

5, 3, 12
Other Types of Problems

- Is this (array or tree) a heap?
- Tree representation vs array implementation:
  - Draw the tree-like picture of the heap given by the array ...
  - Given tree-like picture, give the array
- Perform a sequence of remove/insert on this heap.
- Decrement priority of node $x$ to $k$
- Increment priority of node $x$ to $k$
- Remove a specific node (not the max)