COMP251: Heaps & Heapsort

Giulia Alberini & Jérôme Waldispühl
School of Computer Science
McGill University

From (Cormen et al., 2002)

Based on slides from D. Plaisted (UNC)
Priority Queue

Assume a set of comparable elements or “keys”.

Like a queue, but now we have a more general definition of which element to remove next, namely the one with highest priority.

e.g. hospital emergency room
Priority Queue ADT

- add(key)
- removeMin()
  - “highest” priority = “number 1” priority
- peek()
- contains(element)
- remove(element)
Complete Binary Tree (definition)

Binary tree of height \( h \) such that every level less than \( h \) is full, and all nodes at level \( h \) are as far to the left as possible.
Heap data structure

• Tree-based data structure (here, binary tree, but we can also use k-ary trees)

• Max-Heap
  – Largest element is stored at the root.
  – for all nodes $i$, excluding the root, $A[\text{PARENT}(i)] \geq A[i]$.

• Min-Heap
  – Smallest element is stored at the root.
  – for all nodes $i$, excluding the root, excluding the root, $A[\text{PARENT}(i)] \leq A[i]$.

• Tree is filled top-down from left to right
  → Complete tree
Heaps – Example

Max-heap as a binary tree.

Last row filled from left to right.
Heap (array implementation)
Heap (array implementation)
parent = child / 2
left   = 2*parent
right  = 2*parent + 1
Heap index relations

parent = child / 2
left = 2*parent
right = 2*parent + 1
Heap index relations

parent = child / 2
left = 2*parent
right = 2*parent + 1
Heap index relations

\[
\text{parent} = \text{child} / 2
\]
\[
\text{left} = 2 \times \text{parent}
\]
\[
\text{right} = 2 \times \text{parent} + 1
\]
Height

- **Height of a node in a tree**: the number of edges on the longest simple path down from the node to a leaf.
- **Height of a heap = height of the root** = $\Theta(lg n)$.
- Most Basic operations on a heap run in $O(lg n)$ time
- Shape of a heap
Sorting with Heaps

• Use max-heaps for sorting.
• The array representation of max-heap is not sorted.
• Steps in sorting
  1. Convert the given array of size $n$ to a max-heap ($BuildMaxHeap$)
  2. Swap the first and last elements of the array.
      • Now, the largest element is in the last position – where it belongs.
      • That leaves $n - 1$ elements to be placed in their appropriate locations.
      • However, the array of first $n - 1$ elements is no longer a max-heap.
      • Float the element at the root down one of its subtrees so that the array remains a max-heap ($MaxHeapify$)
      • Repeat step 2 until the array is sorted.
Heapsort

• Combines the better attributes of merge sort and insertion sort.
  – Like merge sort, worst-case running time is $O(n \lg n)$.
  – Like insertion sort, sorts in place.

• Introduces an algorithm design technique
  – Create data structure (*heap*) to manage information during the execution of an algorithm.

• The *heap* has other applications beside sorting.
  – Priority Queues
Maintaining the heap property

• Suppose two sub-trees are max-heaps, but the root violates the max-heap property.

• Fix the offending node by exchanging the value at the node with the larger of the values at its children.
  – The resulting tree may have a sub-tree that is not a heap.

• Recursively fix the children until all of them satisfy the max-heap property.
MaxHeapify – Example

Node n=2

MaxHeapify(A, 2)
MaxHeapify – Example

MaxHeapify(A, 2)
MaxHeapify – Example

MaxHeapify(A, 2)
MaxHeapify(A, 4)
MaxHeapify – Example

MaxHeapify(A, 2)
MaxHeapify(A, 4)
MaxHeapify – Example

MaxHeapify(A, 2)
MaxHeapify(A, 4)
MaxHeapify(A, 9)
MaxHeapify – Example

- Root: A[1]
- Left[i]: A[2i]
- Right[i]: A[2i+1]
- Parent[i]: A[\lfloor i/2 \rfloor]

MaxHeapify(A, 2)
MaxHeapify(A, 4)
MaxHeapify(A, 9)
Procedure MaxHeapify

**Assumption:** Left\((i)\) and Right\((i)\) are max-heaps.

\(n\) is the size of the heap.

\[
\text{MaxHeapify}(A, \ i) \quad \#\text{use heap properties to find the children of the root in the array}
\]

1. \(l \leftarrow \text{leftNode}(i)\)
2. \(r \leftarrow \text{rightNode}(i)\)
3. \(n \leftarrow \text{HeapSize}(i)\)
Procedure MaxHeapify

**Assumption:** Left($i$) and Right($i$) are max-heaps.
 n is the size of the heap.

```
MaxHeapify(A, i)

1.  l ← leftNode(i)          #use heap properties to
2.  r ← rightNode(i)         find the children of the root
3.  n ← HeapSize(i)          in the array
5.      then largest ← l     value of the
6.  else largest ← i          root and its
8.      then largest ← r     children
```
Procedure MaxHeapify

Assumption: Left($i$) and Right($i$) are max-heaps. 

$n$ is the size of the heap.

```plaintext
MaxHeapify(A, i)  
1. $l \leftarrow$ leftNode($i$)  
2. $r \leftarrow$ rightNode($i$)  
3. $n \leftarrow$ HeapSize($i$)  
4. if $l \leq n$ and $A[l] > A[i]$  
   then $largest \leftarrow l$  
5. else $largest \leftarrow i$  
6. if $r \leq n$ and $A[r] > A[largest]$  
   then $largest \leftarrow r$  
7. if $largest \neq i$  
   then exchange $A[i] \leftrightarrow A[largest]$  
8. $MaxHeapify(A, largest)$  
```

If the root is not the largest, then we swap and maxHeapify child
Procedure MaxHeapify

**Assumption:** Left($i$) and Right($i$) are max-heaps.  
$n$ is the size of the heap.

```plaintext
MaxHeapify(A, i)
1.  l ← leftNode(i)  #use heap properties to
2.  r ← rightNode(i)  find the children of the root
3.  n ← HeapSize(i)  in the array
4.  if $l \leq n$ and $A[l] > A[i]$
5.      then largest ← l  Compare the
6.     else largest ← i  value of the
8.      then largest ← r  left and right
9.    if largest ≠ i  children
10.   then exchange $A[i] \leftrightarrow A[largest]$  
11.   MaxHeapify(A, largest)
```

If the root is not the largest, then we swap and maxHeapify child

Time to determine if there is a conflict and find the largest children is $\Theta(1)$
Procedure MaxHeapify

**Assumption:** Left\((i)\) and Right\((i)\) are max-heaps. 
n is the size of the heap.

```
MaxHeapify\((A, i)\)
1. \(l \leftarrow \text{leftNode}(i)\)  # use heap properties to find the children of the root in the array
2. \(r \leftarrow \text{rightNode}(i)\)
3. \(n \leftarrow \text{HeapSize}(i)\)
4. if \(l \leq n \text{ and } A[l] > A[i]\)
5. then \(\text{largest} \leftarrow l\)
6. else \(\text{largest} \leftarrow i\)
7. if \(r \leq n \text{ and } A[r] > A[\text{largest}]\)
8. then \(\text{largest} \leftarrow r\)
9. if \(\text{largest} \neq i\)
10. then exchange \(A[i] \leftrightarrow A[\text{largest}]\)
11. \(\text{MaxHeapify}(A, \text{largest})\)
```

If the root is not the largest, then we swap and maxHeapify child

Time to determine if there is a conflict and find the largest children is \(\Theta(1)\)

Time to fix the subtree rooted at one of \(i\)’s children is \(O(\text{size of subtree})\)
Worst case running time of MaxHeapify(A, 0)

- **Size of a tree** = number of nodes in this tree
- **T(n)**: time used for an input of size n (a tree with n nodes)
- **T(n) = T(size of the largest subtree) + Θ(1)**
- **Size of the largest subtree ≤ 2n/3** (worst case occurs when the last row of tree is exactly half full)

\[ T(n) \leq T(2n/3) + \Theta(1) \Rightarrow T(n) = O(lg n) \]

Alternately, MaxHeapify takes \( O(h) \) where \( h \) is the height of the node where MaxHeapify is applied
Height vs. Depth

Maximum capacity of a heap

Max # nodes / level

2^0

2^1

2^2

2^3

Maximum capacity of a binary tree of height \( h = 2^{h+1} - 1 \)
Heap of height \( h+1 \) has at least \( (2^{h+1}-1) + 1 \) nodes

\( \Rightarrow n_h \geq 2^h \Rightarrow \log_2 n_h \geq h \Rightarrow h = O(\log n) \)
Worst case running time of MaxHeapify

Note: Valid iff $h \geq 1$
The last row is double the size of the previous row.

Note: Valid iff $h \geq 1$
Worst case running time of MaxHeapify

The last row is double the size of the previous row.

Total in heap (n): \( n = 3 \cdot 2^h - 1 \)
Total left subtree \( n_{left} \leq 2^{h+1} - 1 = \frac{3}{3} \cdot 2 \cdot (2^h - \frac{1}{2}) = \frac{2}{3} \cdot (3 \cdot 2^h - \frac{3}{2}) \leq \frac{2}{3} \cdot n \)

Note: Valid iff \( h \geq 1 \)
Worst case running time of MaxHeapify

\[ O(\log(n)) \leq \frac{2 \cdot n}{3} \geq \frac{n}{3} \]
Building a heap

- Use \textit{BuildMaxHeap} to convert an array \( A \) into a max-heap.
- Call MaxHeapify on each element in a bottom-up manner.

\textbf{BuildMaxHeap}(A)

1. \( n \leftarrow \text{length}[A] \)
2. \textbf{for} \( i \leftarrow \left\lfloor \text{length}[A]/2 \right\rfloor \) \textbf{downto} 1
3. \textbf{do} MaxHeapify(A, \( i \), \( n \))

Length(a)/2 is the midpoint. At the right, everything is a child.
**BuildMaxHeap – Example**

Input Array:

```
24  21  23  22  36  29  30  34  28  27
```

Starting tree *(not max-heap)*
BuildMaxHeap – Example

MaxHeapify(\lceil 10/2 \rceil = 5)
BuildMaxHeap – Example

MaxHeapify(\left\lfloor 10/2 \right\rfloor = 5)
MaxHeapify(4)
BuildMaxHeap – Example

MaxHeapify(\lfloor 10/2 \rfloor = 5)
MaxHeapify(4)
BuildMaxHeap – Example

MaxHeapify(\[10/2\] = 5)
MaxHeapify(4)
MaxHeapify(3)
BuildMaxHeap – Example

MaxHeapify(\lfloor 10/2 \rfloor = 5)
MaxHeapify(4)
MaxHeapify(3)
BuildMaxHeap – Example

MaxHeapify(\lfloor 10/2 \rfloor = 5)
MaxHeapify(4)
MaxHeapify(3)
MaxHeapify(2)
BuildMaxHeap – Example

MaxHeapify(⌈10/2⌉ = 5)
MaxHeapify(4)
MaxHeapify(3)
MaxHeapify(2)
BuildMaxHeap – Example

MaxHeapify(\lceil 10/2 \rceil = 5)
MaxHeapify(4)
MaxHeapify(3)
MaxHeapify(2)
BuildMaxHeap – Example

MaxHeapify(\lfloor 10/2 \rfloor = 5)
MaxHeapify(4)
MaxHeapify(3)
MaxHeapify(2)
MaxHeapify(1)
BuildMaxHeap – Example

MaxHeapify(\lfloor 10/2 \rfloor = 5)
MaxHeapify(4)
MaxHeapify(3)
MaxHeapify(2)
MaxHeapify(1)
BuildMaxHeap – Example

MaxHeapify(\lfloor 10/2 \rfloor = 5)
MaxHeapify(4)
MaxHeapify(3)
MaxHeapify(2)
MaxHeapify(1)
BuildMaxHeap – Example

MaxHeapify(\lfloor 10/2 \rfloor = 5)
MaxHeapify(4)
MaxHeapify(3)
MaxHeapify(2)
MaxHeapify(1)

We did a O(log n) operation O(n) times.
Correctness of \textit{BuildMaxHeap}

• **Loop Invariant property (LI):** At the start of each iteration of the \texttt{for} loop, each node \(i+1, i+2, \ldots, n\) is the root of a max-heap.

• **Initialization:**
  – Before first iteration \(i = \left\lfloor n/2 \right\rfloor\)
  – Nodes \(\left\lfloor n/2 \right\rfloor+1, \left\lfloor n/2 \right\rfloor+2, \ldots, n\) are leaves, thus max-heaps.

• **Maintenance:**
  – By LI, subtrees at children of node \(i\) are max heaps.
  – Hence, \texttt{MaxHeapify}(\(i\)) renders node \(i\) a max heap root (while preserving the max heap root property of higher-numbered nodes).
  – Decrementing \(i\) reestablishes the loop invariant for the next iteration.

• **Stop:** bounded number of calls to \texttt{MaxHeapify}
Running Time of *BuildMaxHeap*

**Loose upper bound:**
- Cost of a `MaxHeapify` call $\times$ number of calls to `MaxHeapify`
- $O(\lg n) \times O(n) = O(n \lg n)$

But we’re not really doing $O(n)$ work at each step since the heaps get smaller.

**Tighter bound:**
- Cost of `MaxHeapify` is $O(h)$.
- Height of heap is $\lfloor \lg n \rfloor$
- $\leq \lceil n/2^{h+1} \rceil$ nodes with height $h$. 
\[ \left\lfloor \frac{n}{2^{h+1}} \right\rfloor \]

\( n = 15 \)

\( \leq \left\lfloor \frac{n}{2^{h+1}} \right\rfloor \) nodes with height \( h \).

- \( h = 3 \)
  - \( 1 \leq \left\lfloor \frac{15}{2^{3+1}} \right\rfloor = 1 \) nodes w. \( h=3 \)

- \( h = 2 \)
  - \( 2 \leq \left\lfloor \frac{15}{2^{2+1}} \right\rfloor = 2 \) nodes w. \( h=2 \)

- \( h = 1 \)
  - \( 4 \leq \left\lfloor \frac{15}{2^{1+1}} \right\rfloor = 4 \) nodes w. \( h=1 \)

- \( h = 0 \)
  - \( 8 \leq \left\lfloor \frac{15}{2} \right\rfloor = 8 \) nodes w. \( h=1 \)

\( h + 1 \) is the nth row, so we know the max number of nodes per row.
Running Time of \textit{BuildMaxHeap}

- **Loose upper bound:**
  - Cost of a \textit{MaxHeapify} call $\times$ \# calls to \textit{MaxHeapify}
  - $O(lg\ n) \times O(n) = O(n\ lg\ n)$

- **Tighter bound:**
  - Cost of \textit{MaxHeapify} is $O(h)$.
  - Height of heap is $[lg\ n]$
  - $\leq [n/2^{h+1}]$ nodes with height $h$.

\[
\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1 - 1/2)^2} = 2
\]

There are log $n$ rows. Total work of a row is a fraction of $n$. When we combine all the rows, the total work done grows linearly with $n$.

\[
\sum_{h=0}^{\lfloor lg\ n \rfloor} \left\lfloor \frac{n}{2^{h+1}} \right\rfloor O(h) = O\left(n \sum_{h=0}^{\lfloor lg\ n \rfloor} \frac{h}{2^h}\right) = O(n)
\]

**Running time of BuildMaxHeap is $O(n)$**
Heapsort

1. Builds a max-heap from the array.

2. Put the maximum element (i.e. the root) at the correct place in the array by swapping it with the element in the last position in the array.

3. “Discard” this last node (knowing that it is in its correct place) by decreasing the heap size, and call MAX-HEAPIFY on the new root.

4. Repeat this process (goto 2) until only one node remains.
Heapsort(A)

HeapSort(A)
1. Build-Max-Heap(A)
2. for $i \leftarrow \text{length}[A]$ downto 2
4. MaxHeapify(A, 1, i-1)

Remember insertion sort! We are progressively sorting a sub-array, this time at the end. We make a heap, take the first element (the max), swap it to the end of the list, repeat with a shorter list
Heapsort – Example

7 4 3 1 2
Heapsort – Example

2 4 3 1 7

4 2 3 1 7

Heapify

1

7

1

2

4

3

2

3

4

5

6

7
Heapsort – Example

1 2 3 4 7

3 2 1 4 7

Heapify

1

2 3

3

2 1

7 4
Heapsort – Example

1 2 3 4 7

2 1 3 4 7

Heapify

1 2

2 1

7 4 3
Heapsort – Example

1 2 3 4 7

1 2 3 4 7

1 7 4 3 2
Heap Procedures for Sorting

• BuildMaxHeap \( O(n) \)
• for loop \( n-1 \) times (i.e. \( O(n) \))
  – exchange elements \( O(1) \)
  – MaxHeapify \( O(lg \ n) \)

=> HeapSort \( O(n \ lg \ n) \)

Happens only once!

Because we built the max heap, at each step, the array is only one set of swaps away from being a max heap