COMP251: Heaps & Heapsort

Roman Sarrazin-Gendron & Jérôme Waldispühl
School of Computer Science
McGill University

From (Cormen et al., 2002)

Based on slides from D. Plaisted (UNC)
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.
Heaps – Example

Max-heap as a binary tree.

Last row filled from left to right.
Heaps as arrays

Notice the first index is 1

Max-heap as an array.

Map from array elements to tree nodes:
- Root : $A[1]$
- Left[$i$] : $A[2i]$
- Right[$i$] : $A[2i+1]$
- Parent[$i$] : $A[\lfloor i/2 \rfloor]$
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

![Diagram of a heap](image)
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 (recall COMP250)
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

MaxHeapify(A, 2)

Node n=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

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

Root : \(A[1]\)
Left\([i]\) : \(A[2i]\)
Right\([i]\) : \(A[2i+1]\)
Parent\([i]\) : \(A[\lfloor i/2 \rfloor]\)
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 \( l \leq n \) and \( A[l] > A[i] \)
5. then \( \text{largest} \leftarrow l \)
6. else \( \text{largest} \leftarrow i \)
7. if \( r \leq n \) and \( A[r] > A[\text{largest}] \)
8. then \( \text{largest} \leftarrow r \)
9. if \( \text{largest} \neq i \)
10. then \( \text{exchange } A[i] \leftarrow A[\text{largest}] \)
11. \( \text{MaxHeapify}(A, \text{largest}) \)

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

Compare the value of the root and its left and right children

Time to fix the subtree rooted at one of i’s children is \( O(\text{size of subtree}) \)

If the root is not the largest, then we swap and maxHeapify child
Worst case running time of MaxHeapify($A, n$)

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

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

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

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

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

\[ \Omega\left(\log(n)\right) \leq \frac{2 \cdot n}{3} \leq \frac{n}{3} \]
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.

\begin{itemize}
  \item \textbf{BuildMaxHeap}(A)
  \begin{enumerate}
    \item \( n \leftarrow \text{length}[A] \)
    \item \textbf{for} \( i \leftarrow \lfloor \text{length}[A]/2 \rfloor \) \textbf{downto} 1
    \item \textbf{do} \text{MaxHeapify}(A, i, n)
  \end{enumerate}
\end{itemize}

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(10/2 = 5)
BuildMaxHeap – Example

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

MaxHeapify(\(\lfloor 10/2 \rfloor = 5 \))

MaxHeapify(4)
BuildMaxHeap – Example

\[
\begin{align*}
\text{MaxHeapify} \left( \left\lfloor 10/2 \right\rfloor = 5 \right) \\
\text{MaxHeapify}(4) \\
\text{MaxHeapify}(3)
\end{align*}
\]
BuildMaxHeap – Example

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

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

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

MaxHeapify(\lfloor 10/2 \rfloor = 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

We did a $O(\log n)$ operation $O(n)$ times.

MaxHeapify(\lfloor 10/2 \rfloor = 5)
MaxHeapify(4)
MaxHeapify(3)
MaxHeapify(2)
MaxHeapify(1)
Correctness of **BuildMaxHeap**

- **Loop Invariant property (LI):** At the start of each iteration of the `for` loop, each node \(i+1, i+2, \ldots, n\) is the root of a max-heap.
- **Initialization:**
  - Before first iteration \(i = \lfloor n/2 \rfloor\)
  - Nodes \(\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n\) are leaves, thus max-heaps.
- **Maintenance:**
  - By LI, subtrees at children of node \(i\) are max heaps.
  - Hence, `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 `MaxHeapify`
Running Time of \textit{BuildMaxHeap}

- **Loose upper bound:**
  - Cost of a \textit{MaxHeapify} call \times \# calls to \textit{MaxHeapify}
  - $O(\lg n) \times O(n) = O(n \lg n)$
    - But we’re not really doing $O(n)$ work at each step since the heaps get smaller

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

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

$$
\sum_{h=0}^{\left\lceil \lg n \right\rceil} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\left\lceil \lg n \right\rceil} \frac{h}{2^h}\right) = O(n)
$$

\textbf{Running time of BuildMaxHeap is } O(n)
\[ \left\lfloor \frac{n}{2^{h+1}} \right\rfloor \]

Let \( n = 15 \):

- **h=3**: \( 1 \leq \left\lfloor \frac{15}{2^{3+1}} \right\rfloor = 1 \) nodes with \( h=3 \)
- **h=2**: \( 2 \leq \left\lfloor \frac{15}{2^{2+1}} \right\rfloor = 2 \) nodes with \( h=2 \)
- **h=1**: \( 4 \leq \left\lfloor \frac{15}{2^{1+1}} \right\rfloor = 4 \) nodes with \( h=1 \)
- **h=0**: \( 8 \leq \left\lfloor \frac{15}{2} \right\rfloor = 8 \) nodes with \( h=1 \)

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

• **Loose upper bound:**
  – Cost of a MaxHeapify call × # calls to MaxHeapify
  – $O(\lg n) \times O(n) = O(n \lg n)$

• **Tighter bound:**
  – Cost of MaxHeapify is $O(h)$.
  – $\leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes with height $h$.
  – Height of heap is $\left\lceil \frac{\lg n}{2} \right\rceil$

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

The second summation is a constant!

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

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 $n$.

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\((A)\)

\[
\text{HeapSort}(A)
\]

1. Build-Max-Heap\((A)\)
2. for \(i \leftarrow \text{length}[A]\) downto 2
3. do exchange \(A[1] \leftrightarrow A[i]\)
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 |

```
    7
   / 
  4   3
 / \
1   2
```
Heapsort – Example

2  4  3  1  7  

4  2  3  1  7  

Heapify

Heapsort algorithm example.
Heapsort – Example

Heapify

1  2  3  4  7

3  2  1  4  7

Heapify

1  2  3

3  2  1

7  4
Heapsort – Example
Heapsort – Example

1 2 3 4 7

1 2 3 4 7

1

7 4 3 2
Heap Procedures for Sorting

- **BuildMaxHeap**  $O(n)$  Happens only once!
- for loop $n-1$ times (i.e. $O(n)$)
  - exchange elements $O(1)$
  - MaxHeapify $O(lg \, n)$

=> HeapSort  $O(n \, lg \, n)$
Office hours

• They started yesterday!
• The calendar is here:  
• The Zoom links are on MyCourses (content)
• There is a waiting room, don’t worry if you are not immediately let in.
• Monitor the chat!