Chapter 6: Heapsort

A complete binary tree is a binary tree in which each non-leaf has two children and all the leaves are at the same depth from the root.

A nearly complete binary tree is a binary tree constructed from a complete binary tree by eliminating a number of nodes (possibly none) from right at the leaf level.

A heap is a node-labeled, nearly complete binary tree with a special property.
Implementing Node-Labeled, Nearly Complete Binary Trees Using Arrays

The array indices start from 1. The nodes are enumerated level-wise, by going from the root to the leaf level and going from left to right within each level.

Justification for Using the Array Implementation

With this implementation, accessing the parent and the children is easy.

- For every $i$, $2 \leq i \leq n$, the parent of the $i^{th}$ node is $\lfloor i/2 \rfloor$.
- For every $i$, $1 \leq i \leq \lfloor n/2 \rfloor$, the left child of the $i^{th}$ node is $2i$.
- For every $i$, $1 \leq i \leq \lfloor (n - 1)/2 \rfloor$, the right child of the $i^{th}$ node is $2i + 1$. 
The Special Property of a Heap

A **max-heap** is a node-labeled, nearly complete binary tree, where the label is called the **key** and the keys satisfy the following **max-heap property**: 

- *For every non-leaf, its key is less than or equal to its parent’s key.*

A **min-heap** is defined with “less than or equal to” in place of “greater than or equal to.” In this case the property is called the **min-heap property**.

Here we study only max-heaps.
The Height of a Heap

What is the height of an \( n \)-node heap?

I think it’s \( \lceil \log(n + 1) \rceil \).

Close. You’re off by one. It is \( \lceil \log(n + 1) \rceil - 1 \).

... Equivalently, it is \( \lceil \log n \rceil \).
Basic Operations on Max-Heaps

Max-Heapify

An input to the procedure is a heap and a node $i$. We set $k$ to $i$, and then execute the following:

- If $k$ is a leaf, then quit the loop now.
- If the key of each child of $k$ is at most the key of $k$, then quit the loop.
- If $k$ has only one child, then set $k'$ to the unique child. Otherwise, set $k'$ to the child with the greater key than the other.
- Exchange the keys between $k$ and $k'$.
- Set $k$ to $k'$.
The Pseudo-code

Max-Heapify\((A, n, i)\)

1: \( k \leftarrow i \)
2: \( \text{while } k \leq n/2 \text{ do} \)
3: \( \{ \quad k' \leftarrow 2k \)
4: \( \triangleright \text{ Set } k' \text{ to the left child for now} \)
5: \( \quad \text{if } (k' + 1 \leq n \quad \text{and} \quad A[k'] < A[k' + 1]) \)
6: \( \quad \text{then } k' \leftarrow k' + 1 \)
7: \( \triangleright \text{ If the right child exists and it has a larger key} \)
8: \( \triangleright \text{ than the left child, } k' \text{ is the right child} \)
9: \( \quad \text{if } A[k] < A[k'] \text{ then} \)
10: \( \quad \{ \quad x \leftarrow A[k]; \quad A[k] \leftarrow A[k']; \quad A[k'] \leftarrow x \quad \} \)
11: \( \triangleright \text{ Swap } A[k] \text{ and } A[k'] \text{ if } A[k'] > A[k] \)
12: \( k \leftarrow k' \)
13: \( \triangleright \text{ Move to } k' \)
14: \( \} \)
Why is the loop terminated when $k$ exceeds $n/2$?

I don’t know...
It’s because the nodes beyond \( n/2 \) are leaves and thus are without children.

Now what’s the running time?

It’s proportional to the height of \( i \).

That’s correct.
**Build-Max-Heap**

This operation turns a given array into a max-heap.

To do this, we first turn each of the subtrees of the root into a max-heap. Then there is only one spot where violation of the max-heap property may exist, which is the root. If we execute **Max-Heapify** from the root, then such violation will be eliminated.

**Build-MaxHeap**($A, n$)

1: **Build-MaxHeap0**($A, n, 1$)

**Build-MaxHeap0**($A, n, i$)

1: if $2i \leq n$ then
2: **Build-MaxHeap0**($A, n, 2i$)
3: if $2i + 1 \leq n$ then
4: **Build-MaxHeap0**($A, n, 2i + 1$)
5: **Max-Heapify**($A, n, i$)
Unrolling the Recursion

The algorithm is a collection of calls to Max-Heapify. Since Max-Heapify($A, n, i$) does not change the keys of the nodes outside $\text{subtree}(i)$, we can reorder any way we want so long as the call for a node comes after the for its descendants. So, the following does the same:

1: \textbf{for} $i \leftarrow n$ \textbf{downto} 1 \textbf{do} \\
2: \quad \text{Max-Heapify}($A, n, i$)

Since Max-Heapify($A, n, i$) does nothing if $i > n/2$, the initial value of $i$ can be $\lceil n/2 \rceil$.

1: \textbf{for} $i \leftarrow \lceil n/2 \rceil$ \textbf{downto} 2 \textbf{do} \\
2: \quad \text{Max-Heapify}($A, n, i$)
An example
What is the running time of Build-MaxHeap($A, n$)?

The height of the heap is $O(\log n)$, so each call to Heapify costs $O(\log n)$.

Since there are $O(n)$ calls, the total cost is $O(n \log n)$.

Correct, but the analysis can be more rigorous...
For each $h$, $0 \leq h \leq \lfloor \lg n \rfloor$, the number of nodes having height $h$ is at most $\left\lfloor \frac{n}{2^{h+1}} \right\rfloor$. So, the total cost is at most

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

Note that

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

Thus, the running time is $O(n)$.

*Can you argue that the running time is actually $\Theta(n)$?*
That’s easy!

Max-Heapify is called \(\lceil n/2 \rceil\) times, so the running time is \(\Omega(n)\).

Since the running time is \(O(n)\), this implies that the running time is \(\Theta(n)\).
Heapsort ⋅ Sorting Using a Max-Heap

First, we turn the input array into a max-heap. By the max-heap property, the root has the largest key. So, we record the key as the largest key in the input array.

Now we replace the key of the root by the key of the last node and decrement the size of the node by one. This may generate violation of the max-heap property, but that can be resolved by Build-Max-Heap. Thus, we find the second largest key.

We will repeat the replacement process to find the third largest, the fourth largest, and so on.
The Pseudo-code

HeapSort($A, n$)
1: Build-MaxHeap($A, n$)
2: for $i = n$ downto 1 do
4: \[\] Identify the $i^{th}$ smallest element
6: \[\] Replace $A[1]$
7: Max-Heapify($A, i, 1$) \}
8: \[\] Heapify
9: for $i = 1$ to $n$ do $A[i] \leftarrow B[i]$

What's the running time?
Well, Max-Heapify runs in time $O(\log n)$.

There are $n$ calls to it.

So, the running time in question is $O(n \log n)$.

That's correct.
An Example

The input numbers

Heapified from the root

After Build-Max-Heap

Heapified from the root
2 has replaced 9

Heapified from the root

3 has replaced 8

Heapified from the root
A priority queue

A data structure for maintaining elements that are assigned numeric values.

Operations on Priority Queues

- **Maximum**: To obtain an element with the largest key.
- **Extract-Max**: To take out the element with the largest key.
- **Insert**: Insert an element.
- **Delete**: Delete an element.
- **Decrease-Key**: Decrease the key of an element.
- **Increase-Key**: Increase the key of an element.
Implementation of a Priority Queue Using a Max-Heap


Extract-Max($A, n$) : This is the core of HeapSort.

Insert($A, n, x$) : To add a key $x$ to $A$ whose size is $n$, first increment $n$ by one. Then insert $x$ as the $n^{th}$ element. Then resolve violation from the inserted key towards the root.
The Pseudo-code

Insert\((A, n, x)\)
1: \(n \leftarrow n + 1; \ A[n] \leftarrow x\)
2: Upward-Fix\((A, n, n)\)

Upward-Fix\((A, k)\)
1: \(i \leftarrow k\)
2: while \(i \geq 2\) do
3: \(j \leftarrow \lfloor i/2 \rfloor\)
4: if \(A[j] > A[i]\) then
5: \(\{ \ y \leftarrow A[i]; \ A[i] \leftarrow A[j]; \ A[j] \leftarrow y \ \}\)
7: \(i \leftarrow j\)
8: ▶ Move to \(j\)

The running time is \(O(\lg n)\).
Delete($A, i$) : Exercise 6.5-7.

Increase-Key($A, i, x$) : What needs to be done is to replace $A[i]$ by $x$ if $x > A[i]$. To do the replacement, set $A[i]$ to $x$ and call Upward-Fix($A, i$).

Decrease-Key($A, i$) : Use Heapify.