Chapter 6
Heap and Its Application

We have already discussed two sorting algorithms: Insertion sort and Merge sort; and also witnessed both Bubble sort and Selection sort in a project.

Insertion sort takes $\Theta(n^2)$ time in the worst case, but it is a fast in-place algorithm for small input sizes. An algorithm is in-place if only a constant number of elements of the input are stored outside. Merge sort runs in $\Theta(n \log n)$, but it is not an in-place algorithm (null).

We will discuss two more sorting algorithms: heapsort in this chapter and quicksort in the next one.

Heapsort runs in $\Theta(n \log n)$, based on an interesting data structure of heap; while quicksort has an average-case running time of $\Theta(n \log n)$ and a worst-case running time of $\Theta(n^2)$, but often outperforms heapsort in practice.
What is a Heap?

The *binary heap* data structure is an array object that can be viewed as a *nearly complete binary tree*.

Such a tree is completely filled at all the levels, except possibly the bottom one, which is filled from left up to a point. Each node of the tree corresponds to an element of the array that stores the value kept in the node.

Besides array $A$, a heap contains two additional pieces of data, $\text{length}[A]$ containing the maximum number of elements that can be put in $A$, and $\text{heap-size}[A]$ containing the number of elements currently stored in the heap supported by $A$. 
Why a heap?

When representing a binary tree with A, the root of the tree is kept in A[1]. In general, given the index \( i \) of a node in the tree, the indices of its parent in the tree(\( \text{Parent}(i) \)), its left child(\( \text{Left}(i) \)), and its right child(\( \text{Right}(i) \)), can be computed, respectively, as \( \left\lfloor \frac{i}{2} \right\rfloor \), \( 2i \), and \( 2i + 1 \).

On the other hand, given a binary tree, we can also map all the nodes in the tree into the elements of an array A by labeling the root to A[1], and once a node is mapped to an index \( i \), its left child, and right child are mapped to \( 2i \) and \( 2i + 1 \).

Hence, there is a 1-1 correspondence between a binary tree and an array. Such a relationship is particularly appropriate for a nearly complete binary tree (?).
Two kinds of heaps

A *max-heap* satisfies the following additional property of $A[\text{Parent}(i)] \geq A[i]$, namely, the value of every node, except the root, is no more than that of its parent; while a *min-heap* satisfies the property of $A[\text{Parent}(i)] \leq A[i]$, that is, for all the nodes, except the root, its value is no less than that of its parent.

Hence, the root of a max-heap keeps the largest value, while the root of a min-heap keeps the smallest.

We will use the max-heap for the heapsort algorithm; while the min-heap is often used for the *priority queue* data structure and scheduling problems, which we will discuss in CS4310.

**Homework:** Exercises 6.1-6, 6.1-3(*), 6.1-4 and 6.1-5(*).
A few notions

When regarding the heap as a tree, we define the height of a node in a heap to be the number of edges on the longest simple path from this node to a leaf, and define the height of a heap to be the height of its root.

We also define the level of a node in a tree to be the length of the path from this node to the root. Thus, the level of the root is just 0, and the maximum level of a tree equals its height.

In a complete binary tree, every node, except a leaf, has exactly two children. Thus, the root, at level 0, has two children, each of them, at level 1, has two children, thus four nodes at level 2.

In general, at level $i \in [0, H]$, there are exactly $2^i$ nodes. So what?
The height of a tree

Thus, for a complete binary tree with \( n \) nodes,
\[
\sum_{h=0}^{H} 2^i = 2^{H+1} - 1 = n,
\]
i.e., \( H = \log_2(n + 1) - 1 = \Theta(\log(n)) \).

For a nearly complete binary tree, by the same token,
\[
\log_2(n + 1) - 1 \leq H \leq \log_2(n).
\]
Hence, the height of heap, a nearly complete binary tree, is in \( \Theta(\log n) \).

As we will see, most of the operations for heaps run in time proportional to the height of the heap, thus, taking \( \Theta(\log n) \) time.

**Homework:** Exercises 6.1-1(*) and 6.1-2.
Basic heap procedures

We now present the details of several basic procedures, and show how to apply them in the heapsort algorithm:

The Max-Heapify procedure maintains the heap property, running in $\Theta(\log n)$.

The Build-Max-Heap procedure builds up a max-heap in $\Theta(n)$.

The Heapsort algorithm solves the sorting problem in $\Theta(n \log n)$ time.

The Max-Heap-Insert, Heap-Extract-Max, Heap-Increase-Key, and Heap-Insert procedures, all running in $\Theta(\log n)$, allow the max-heap to be used as a max priority queue.
Stay heapy

There are two inputs for the Max-Heapify procedure, an array \( A \) corresponding to a binary tree, and an index \( i \) of a position in \( A \).

When it is called, it is assumed that the binary trees rooted at \( \text{Left}(i) \) and \( \text{Right}(i) \) are both max-heaps, but \( A[i] \) may be smaller than its children, thus \( A \) might not be a max-heap.

The goal of this procedure is to let the value of \( A[i] \) “float down” so that the subtree rooted at \( i \) becomes a max-heap.
An example

Below shows the action of Max-Heapify(A, 2).

Homework: Exercises 6.2-1(*), 6.2-2(*) and 6.2-3.
The code

MAX-HEAPIFY(A, i)
1. l<-Left(i)
2. r<-Right(i)
4. then largest<-l
5. else largest<-i
7. then largest<-r
8. if largest != i
9. then exchange(A[i], A[largest])
10. MAX-HEAPIFY(A, largest)

At each step, the largest of A[i], A[Left(i)] and A[Right(i)] are determined in lines 3-7. If A[i] does not hold the largest one, its value is exchanged with that of A[i].

Now A[largest] contains the original value of A[i], which may be smaller than its child, thus the process repeats itself until it hits the end.
The running time

It takes $\Theta(1)$ to go through lines 1–7. It then might run recursively on a subtree rooted at one of its children.

The worst case happens when the nodes of the last row of the heap also occur in the chosen subtree, when the chosen subtree has $\frac{2n-1}{3}$ nodes (Cf. Next page). Hence, the recurrence for the worst case is

$$T(n) \leq T\left(\frac{2n}{3}\right) + \Theta(1).$$

It is then easy(?) to see that

$$T(n) = O(\log n).$$

**Homework:** Exercises 6.2-5 and 6.2-6(*).
The worst case...

Let \( T(r, T_l, T_r) \) stand for a binary tree with \( r \) being its root, \( T_l (T_r) \) being its left (right) subtree, and let its height be \( h \). When \( T \) is a heap, a worst case is that the height of \( T_r \) is \( h - 2 \), and that of \( T_l \) is \( h - 1 \), and the bottom layer of \( T_l \) is completely filled.

By the results as stated in page 5, the bottom layer of \( T_l \) contains exactly \( 2^{h-1} \) nodes. Moreover, the first \( h - 2 \) layers of \( T_l \) and \( T_r \) contains \( \sum_{j=0}^{h-2} 2^j = 2^{h-1} - 1 \) nodes.

Thus, \( T \) contains \( 2(2^{h-1} - 1) + 2^{h-1} \) plus one more vertex for \( r \), i.e., \( n = 3 \times 2^{h-1} - 1 \) nodes, i.e., \( 2^{h-1} = \frac{n+1}{3} \).

Finally, \( T_l \) contains \( 2^h - 1 = 2 \times 2^{h-1} - 1 \) nodes, which is exactly \( \frac{2n-1}{3} \) nodes.
Build a heap

We already know that the elements in the sub-array \( A \left[ \left\lfloor \frac{n}{2} \right\rfloor + 1, n \right] \) are all leaves. Thus, all such elements are necessarily heaps.

For example, when \( n \) is even, the left (right) child of \( A \left[ \left\lfloor \frac{n}{2} \right\rfloor + 1 \right] \) is \( A[n + 2] \) \( A[n + 3] \), which cannot be in \( A \). What about the case when \( n \) is odd?

The following procedure goes through the remaining nodes backwards (Why?) and runs MAX-HEAPIFY on each and every one of them.

**BUILD-MAX-HEAP(A)**

1. heap-size[A]<-length[A]
2. for i<-length[A]/2 downto 1
3. do MAX-HEAPIFY(A, i)

It can be shown that it takes \( \Theta(n) \) to complete (Page 15-19).
An example

Below shows the action of this procedure.

Homework: Exercises 6.3-1 and 6.3-2(*).
The correctness

We will show that the following loop invariant does hold:

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.

Indeed, prior to the first iteration, $i = \left\lfloor \frac{n}{2} \right\rfloor$. Each of the node $\left\lfloor \frac{n}{2} \right\rfloor + 1, \left\lfloor \frac{n}{2} \right\rfloor + 2, \ldots, n$ is a leaf, thus trivially a max-heap.
Assume that before an iteration, the loop invariant holds, and the value of the loop variable \(i\) is \(i_0\).

Since both \(\text{Left}(i_0)\) and \(\text{Right}(i_0)\) are strictly larger than \(i_0\), thus, by the loop invariant, both the left-subtree and the right-subtree of node \(i_0\) are max-heaps. Hence, the precondition of executing \text{Max-Heapify} is met, which makes the tree rooted at \(i_0\), into a max-heap, while preserving the heap properties of all of the subtrees rooted at \(j \in [i_0 + 1, n]\). Now, all the trees rooted at \([i_0, i_0 + 1, \ldots, n]\) are max-heaps.

At the end of this loop, the loop variable \(i\) is decremented by 1 to contain \(i_0 - 1\), thus withholding the invariant.

At the end of the procedure, the loop variable becomes 0, thus, all the subtrees rooted at \(1, 2, \ldots, n\) are max-heaps. In particular, the one rooted at 1 is a max-heap.
The running time

Since each call to Max-Heapify takes $\log n$, and there are $O(n)$ calls, the running time is $O(n \log n)$. This upper bound, although correct, is not tight.

We may observe that, to adjust an element, we have to do 2 comparisons: 1) get the bigger child; 2) compare the element with this bigger child.

The adjustment of an element could lead to adjustments of all its descendants in the path from this element to the bottom. For example, in the previous case, the element 4 went down all the way to the bottom.

Thus, an adjustment of an element with height $h$ takes $\Theta(h)$. 
So what?

To build a heap, we have to adjust all the elements, if we consider the adjustment of all the leaves are trivial.

Put in everything, the time complexity of the initialization process is bounded by the sum of the heights of all the nodes in the heap.

A *complete binary tree* is a nearly complete binary tree with its bottom level is also completely filled, so such a tree contains at least as many nodes as a nearly complete binary tree with the same height.
Finally,…

**Theorem:** For a complete binary tree of height $H$, containing $2^{H+1} - 1$ nodes, the sum of the height of all the nodes is $\Theta(n)$.

**Proof:** As there is one node at height $H$; 2 nodes at height $H - 1$, ..., $2^H$ nodes at height 0; the height sum is the follows:

$$S = \sum_{i=0}^{H} i \times 2^{H-i} \leq 2^H \sum_{i=0}^{\infty} \frac{i}{2^i} = 2^H \times 2 = 2^{H+1} = n + 1 = \Theta(n).$$

For a nearly complete binary tree, the sum of its heights can not be more than that of the corresponding complete binary tree with the same height.

Therefore, the total number of comparisons for this initialization process is $O(n)$. 

We are ready for Heapsort

Given (16, 14, 10, 8, 7, 9, 3, 2, 4, 1), we “put” it into a heap as shown in Part 2:

Isn’t this neat? 😊
The heapsort algorithm

The heapsort algorithm, when applied to a list of numbers kept in an array $A$, starts by building a max-heap out of $A$, so that the largest element ends up at $A[1]$.

The process continues by exchanging this largest element with $A[n]$. Since $A[n]$ is in its final place, we cut it off the heap.


Now that $A[1]$ again contains the largest element of the leftover, the process will repeat until the shrinking heap contains only one element, which is the smallest of all, and is already located in the proper place.

We are done.
The code

HEAPSORT(A)
1. BUILD-MAX-HEAP(A)
2. for i<-length[A] downto 2
3. do exchange(A[1], A[i])
4. heap-size[A]<-heap-size[A]-1
5. MAX-HEAPIFY(A,1)

The algorithm runs in \(O(n \log n)\), since the initialization takes \(O(n)\), and for the \(n - 1\) loop, MAX-HEAPIFY always runs at \(O(\log n)\).

**Homework:** Make a more detailed analysis of the last statement. (Hint: You need to go back to Chapter 3.) Exercises 6.4-1, 6.4-2(*) and 6.4-3(*).
Priority queues

A priority queue maintains a set $S$ of elements, each of which is associated with a key. We assume there exists a way for us to compare the keys of all the elements.

Priority queue can be used to schedule jobs to be processed with limited resource. Jobs come in with different, adjustable, priority, and will be inserted into such a queue. When a resource becomes available, the job with the maximum priority will be chosen and deleted from the queue. Priority of a job might be increased after its joining the queue.

Priority queue is a quite valuable and useful data structure, and have been made extensive use in operating systems.
A max priority queue supports the following operations:

- **INSERT(S, x)** inserts x into 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)** increases the value of element x to the new value k, which is no smaller than its original key value.

The implementation of these operations depend on the organization of such a queue.
Priority queue implementation

We can implement such a priority queue with an initially empty *unsorted array*, and use a position to indicate where another element can be inserted, initially set at 1.

To *insert* an item, you just add it in the first available place in $\Theta(1)$ time. To look for the *maximum* element in the list, you have to do a *bear/corn* style search in $O(n)$, where $n$ is the number of elements as contained in such a list.

Once you have *extracted* the maximum element from the list, you also have to fill this “hole” by moving all the elements to the right of such a maximum element one position to the left, also in $O(n)$.

Finally, when you want to *increase* the value of an element, you can just do it in $\Theta(1)$.
Other implementations

You can also implement a priority queue with a sorted array, which also takes linear time for some of the four operations.

It turns out that the heap structure is an excellent, $\Theta(\log n)$, implementation of the priority queue. More specifically, the heap implementation leads to $\Theta(1)$ inspection, and $\Theta(\log n)$ running time for insertion, delete-Max and increase-key operations for a priority queue containing $n$ elements.

We will mainly discuss the max priority queue, which always deletes the maximum element. The min priority queue is the same.

**Question:** Have you checked the project page recently?
Implementation

When $S$ is implemented with a heap $A$, the MAXIMUM($S$) operation is simply return $A[1]$.

The code of EXTRACT-MAX takes out the biggest one, i.e., $A[1]$, fills the “hole” with the last one, then adjusts the heap.

**HEAP-EXTRACT-MAXIMUM**(A)
1. if heap-size[A] < 1
2. then error "heap underflow"
3. max <- A[1]
6. **MAX-HEAPIFY**(A, 1)
7. return max

This procedure runs in $O(\log n)$, since it only does a constant amount of work, besides running **MAX-HEAPIFY** once.
Key increase

The code increases $A[i]$ to its new value $key$, which might violate the max-heap property since it is now might be bigger than that of its parent.

Thus, we have to go *upwards* toward the root to find a proper place for this newly adjusted key.

**HEAP-INCREASE-KEY**($A$, $i$, $key$)

1. if $key < A[i]$
2. then error "new key is too small"
3. $A[i] \leftarrow key$
4. while $i > 1$ and $A[\text{Parent}(i)] < A[i]$
5. do exchange($A[\text{Parent}(i)]$, $A[i]$)
6. $i \leftarrow \text{Parent}(i)$

This procedure also takes $O(\log n)$ since the length of the path from $i$ to the root is at most $\log n$. 
An example

Below shows the action of this procedure of key increase.

Homework: Exercise 6.5-1(*).
Add in another piece

When adding in another element with key, we naturally expand A, add in an element with the minimum key, then call the just discussed key increase operation to beef its value to key.

MAX-HEAP-INSERT(A, key)
1. heap-size[A] <- heap-size[A] + 1
2. A[heap-size[A]] <- minInt
3. HEAP-INCREASE-KEY(A, heap-size[A], key)

It is clear that its running time is also $O(\log n)$.

Hence, the heap structure can implement a priority queue in $O(\log n)$ time. Yeah! 😊

Homework: Exercises 6.5-2, and 6.5-4(*).