Introduction: Heaps

A heap is a tree-like data structure that satisfies the heap-order property.

**Definition** (Heap-Order Property). A tree has the *heap-order property* if for any parent node \( P \) with a child \( C \), the key of \( P \) is ordered with respect to the child \( C \).

Common examples of orderings on a heap would be \( \geq \) (max-heap) or \( \leq \) (min-heap). For \( \geq \), the key in each node in the heap \( T \) is greater than or equal to the keys of all nodes in its subtree.

![An example binary max-heap. Note that the root contains the maximum key.](image)

Notice that this definition immediately implies that the root must contain either the “maximum” or the “minimum” of the ordering relationship that we define, since the root is the parent/ancestor of every other node. Specializing this definition to keys that act like natural numbers, or keys that implement `Comparable`, we have our classic min-heap and max-heap. This basic idea is really powerful, as the heap data structure maintains the “maximum” or “minimum” element whenever we add to it or remove from it. This means that we can retrieve the max/min element quickly!

Binary Heaps

A binary heap is a binary tree, but with the heap-order property. A binary heap is most commonly implemented by flattening a tree in level order into an array. It satisfies the following property:

**Definition** (Shape Property). A tree has the *heap-shape property* if the tree is a *complete binary tree*. That is, all levels of the tree are fully filled, except for possibly the last, where all nodes are as far left as possible.

With the shape property, we can easily index into a binary heap, since we will not have to worry about “gaps.”
A max-heap visualized as both a tree and an array.

For an element at index $i$ of $A$, its left and right children can be found at indices $2i$ and $2i + 1$ respectively. Conversely, an element at index $i$ has its parent at index $\left\lfloor i/2 \right\rfloor$.

This property holds true only if the heap begins at index 1 of the array (or if the array is one-indexed).

**Running time of Operations**

*Insert* $(x, k)$: An element $x$ with key $x$ may be inserted in $O(\log n)$ time.

*Find-Min/Max()*: Finding the min/max of a binary heap takes $O(1)$ time.

*Extract-Min/Max()*: Removing the root and restoring the min/max heap property takes $O(\log n)$ time.

*Decrease/Increase-Key* $(x, k)$: Changing the key of an element can be done in $O(\log n)$ time. Note that the Java implementation of a priority queue does not support this operation.

**Partial Ordering**

We say that the heap-order property induces a partial order over its elements. Intuitively, a partial order means that not every pair of elements are related. Even though we know that 17 is less than 23, when we insert these numbers into the heap, we cannot determine which number is “greater” solely by its position in the heap. Compare this to inserting both elements in a binary search tree, where we can determine the order by examining their relative positions. We say that the binary search tree establishes a total order.

For some problems, it is enough to have just a partial ordering. For example, if you want to get the $k$-largest elements of a list relatively fast, you can use a heap to achieve this. As you’ve seen with MERGESORT and some implementations of QUICKSORT, you can get a stronger, total ordering at the cost of a larger running time $[\Omega(n \log n)]$. However, building a heap only takes time linear in the number of elements. Therefore, we can get the maximum/minimum in linear time and the partial ordering!

**Building a (Max) Heap**

In order to build a heap, we define the following subroutine: max-heapify. Under the assumption that the left and right subtrees of the $i$'th vertex are valid max heaps, max-heapify ensures that the subtree rooted at $i$ is also a valid max heap. The running time analysis of max-heapify is left as a discussion topic. We can then write:
function `Max-Heapify(A, i)`
\[
\begin{align*}
  l & \leftarrow \text{left}(i) \\
  r & \leftarrow \text{right}(i) \\
  \text{if } l \leq A.heapsize \text{ and } A[l] > A[i] \text{ then} \\
  & \quad \text{largest} \leftarrow l \\
  \text{else} \\
  & \quad \text{largest} \leftarrow i \\
  \text{if } r \leq A.heapsize \text{ and } A[r] > A[\text{largest}] \text{ then} \\
  & \quad \text{largest} \leftarrow r \\
  \text{if } \text{largest} \neq i \text{ then} \\
  & \quad \text{swap}(A[i], A[\text{largest}]) \\
  & \quad \text{MAX-HEAPIFY}(A, \text{largest}) \quad \triangleright \text{One of children is larger. Swap and recurse.}
\end{align*}
\]

function `Build-Max-Heap(A)`
\[
\begin{align*}
  A.heapsize & \leftarrow A.length \\
  \text{for } i \leftarrow \lfloor A.length/2 \rfloor \text{ downto } 1 \text{ do} \\
  & \quad \text{MAX-HEAPIFY}(A, i)
\end{align*}
\]
The `Build-Max-Heap` algorithm starts from the last internal node of the binary tree representation of \( A \) and converts each subtree to a max-heap, recursing upwards. As above, the running time analysis of `Build-Max-Heap` is left as a discussion topic.

Heapsort

function `Heapsort(A)`
\[
\begin{align*}
  & \quad \text{BUILD-MAX-HEAP}(A) \\
  & \quad \text{for } i \leftarrow A.length \text{ downto } 2 \text{ do} \\
  & \quad \quad \text{swap}(A[1], A[i]) \\
  & \quad \quad A.heapsize \leftarrow A.heapsize - 1 \\
  & \quad \quad \text{MAX-HEAPIFY}(A, 1)
\end{align*}
\]
The `Heapsort` algorithm works by first converting the input array \( A \) to a max-heap. It grows the sorted subarray from right to left by swapping out the root (largest element at \( A[1] \)) to its proper place in the sorted subarray and restoring the max-heap property on the unsorted subarray. (Does this notion of dividing the input into an unsorted/sorted region remind you of another sorting algorithm...?) The running time analysis of `Heapsort` is also left as a discussion topic.

Discussion Topics

- What is the worst case running time of `Max-Heapify`? Why?

\textit{Solution.} The worst case running time is \( O(\log n) \). For our algorithm, we do \( \Theta(1) \) work at each level of the recurrence (comparisons/swap) and recurse on either of the subtrees. Therefore, our recurrence will look like this:
\[
T(n) = T(\text{size of subtree}) + \Theta(1)
\]
In the worst case, we want to examine the case where the size of the subtree we recurse on is maximal \textit{with respect to} \( n \). That is, at each level of the recurrence, we want to choose the largest possible fraction of the \( n \) nodes to maximize \( T(n) \). This case occurs when the bottom level of \( T \) is \textit{half full} (the right subtree’s bottom level is empty).
To determine the maximum size of the subtree chosen, we use the following theorem:

**Theorem 1.** Let $T$ be a nonempty, full binary tree. Then the number of leaf nodes in $T$ is one more than the number of internal nodes in $T$.

Let $|R| = k$ be the number of nodes in the right subtree of $T$. Then we have $|L| = k + (k + 1)$ by the above theorem (as $|R|$ would be the number of internal nodes in $L$). Then, $|T| = n = |R| + |L| = 3k + 1$, and $|L|/|T| < 2/3$.

Therefore, we have that the worst case for $T(n) \leq T(2n^{3/2}) + \Theta(1) = O(\log n)$.

- Why does constructing a heap (BUILD-MAX-HEAP) take linear time? What happens if we try to build a heap by running INSERT $n$ times instead?

**Solution.** We first observe that the loop in BUILD-MAX-HEAP begins halfway in $A$ because the latter half of the heap represents the leaves (individual nodes are already heaps). Because of the shape property, the heap contains $2^{h-j}$ nodes with height $j$ at each level of the tree. A node at height $j$ can be swapped down at most $j$ levels. Counting with respect to the number of swap operations, we have at most $T(n) = \sum_{j=0}^{h} j 2^{h-j}$ swaps.

Therefore,

$$T(n) = \sum_{j=0}^{h} j 2^{h-j} \leq \sum_{j=0}^{h} \frac{n}{2^{j}} \leq \frac{n}{2} \sum_{j=0}^{h} \frac{j}{2^{j}}$$

since $n < 2^{h+1}$. (We can assume for simplicity that $n$ is a power of 2).

$$T(n) \leq \frac{n}{2} \sum_{j=0}^{h} \frac{j}{2^{j}} \leq \frac{n}{2} \sum_{j=0}^{\infty} \frac{j}{2^{j}} = O(n)$$

If we try to build a heap by running INSERT given an input of size $n$, we will end up with a $O(n \log n)$ running time:

$$T(n) = c \sum_{i=1}^{n} \log i = c \left[ \log 1 + \log 2 + \log 3 + \cdots + \log n \right] = c \log n! = O(n \log n)$$

- Given that both BUILD-MAX-HEAP and HEAPSORT call MAX-HEAPIFY at least $n/2$ times, why does HEAPSORT run in $\Theta(n \log n)$ time and not BUILD-MAX-HEAP?
Solution. Intuitively, the amount of work performed by BUILD-MAX-HEAP is less than that of HEAPSORT. For most nodes $i$ being swapped down the tree in BUILD-MAX-HEAP, the total number of swaps will not be $\Theta(h) = \Theta(\log n)$. No work will be done for half the nodes in the tree at the leaf-level, and at higher heights, the number of nodes that have to do more work decreases exponentially ($/2$ at each level, to be precise). The root is the only node that might have to be swapped down $\log n$ times.

In contrast, in HEAPSORT, at each iteration of the loop we extract the maximum of the heap and have to sift down a value at the root each time. As a result, the amount of comparisons MAX-HEAPIFY will need is always going to be $\Theta(h)$. The tree will shrink as we remove elements, but it doesn’t shrink nearly as fast! The height only decreases by 1 once half of the nodes have been removed.

For the exact math, you can read CLRS for detailed explanations.

- Discuss INSERTION-SORT, MERGESORT, QUICKSORT, and HEAPSORT. What are their relative advantages? When might one sorting algorithm be preferred over the others?

Solution. INSERTION-SORT is simple to implement, efficient for (very) small inputs, adaptive (efficient for mostly sorted inputs), stable, in-place, and online. Terribly inefficient for large inputs (like most quadratic-time sorts). $O(n^2)$ worst case running time, $O(n)$ best case running time. $O(1)$ additional space.

MERGESORT is guaranteed $\Theta(n \log n)$ running time and stable. Better at handling inputs that are slower to access than quicksort. $O(n \log n)$ best and worst case running time. $O(n)$ additional space.

QUICKSORT can be implemented in-place. Randomized quicksort performs (perhaps surprisingly) very well in practice. $O(n \log n)$ best and average case running time, $O(n^2)$ worst case running time. $O(1)$ additional space if in-place.

HEAPSORT is in-place and directly competes with QUICKSORT. Slower in practice than well-implemented QUICKSORT but has better guaranteed worst case running time of $O(n \log n)$. $O(1)$ additional space. Used more frequently in cases with limited memory or systems with real-time constraints/security concerns.

Testing Your Understanding

Answer the following questions regarding implementations of binary heaps.

Problem 1. Consider the following array:

null 6 7 9 15 13 17 14 20 16 23 18 19 37 42 ···

Let this array be the underlying storage for a binary heap. Is this a max-heap or a min-heap? What is the parent of the key 17? What is the left child of 17? The right child?

Solution. We traverse the array from right-to-left, and fill the levels of a binary tree. We get the following heap:

```
          6
        /   \
       7     9
      /   \   /   \
     15   13  17   14
   /   \ /   \   /   \   /   \
  20  16 23  18  19  37  42 ···
```
This is a min-heap. Notice that 17 has index 6 in the array (zero-indexed).

For a heap that starts on index 1 of an array (and has index 0 unused), for an index \( k \), the following are true:

1. The parent has index \( \left\lfloor \frac{k}{2} \right\rfloor \).
2. The left child has index \( 2k \).
3. The right child has index \( 2k + 1 \).

The parent of 17 is 9. The parent, 9, has index 3.
The left child of 17 is 19. The left child, 19, has index 12.
The right child of 17 is 37. The right child, 37, has index 13.

**Problem 2.** Is it possible for the following array to be the underlying array for a heap?

| null | 64 | 42 | 37 | 19 | 21 | 38 | 43 | 23 | 17 | ... |

If it cannot be the underlying array for a binary heap, what key(s) would you have to change in order to make it a heap?

**Solution.** We begin by drawing the heap in tree form:

![Heap Diagram](image)

This is not a heap. On the left side, 19 is not greater than 23. On the right side, 37 is not greater than 38. We can make this a heap by swapping 19 and 23, and 37 and 43.

**Problem 3.** You are now in the shoes of the Java Virtual Machine, and you are tasked with maintaining the min-heap property for a binary heap that is represented in the following array:

| null | 1 | 2 | 3 | 4 | 5 | 6 | 7 | x | ... |

A pesky CIS 121 student has called the `insert` method, which begins by placing the variable \( x \) into the underlying array at the location indicated above. If \( x = 0 \), what is the final state of this array after the `insert` method completes?

**Solution.** These are the sequence of states that the heap goes through:

![Insert Method Diagram](image)
Problem 4. You are still in the shoes of the Java Virtual Machine, and you have to maintain the min-heap property for a binary heap that is represented in the following array:

```
null — 1 2 3 4 5 6 7 ···
```

The same pesky CIS 121 student has called the `removeMin` method, which already removed and returned the value at the location indicated above. What is the final state of this array after you, the JVM, fix the array again so that it has the min-heap property?

Solution. These are the sequence of states for the heap:
Problem 5. You have been hired to write an application for 121-CIS’s new network router! 121-CIS plans to sell their routers to businesses with large corporate networks that need swift detection of network attacks. A network attack is characterized by a large amount of traffic from a single IP address. For the application, you are parsing a stream of packets containing an IP-address and their frequency. Routers have limited memory, and you can only maintain $O(k)$ space for your application, where $k \ll n$.

Design an $O(n \log k)$ time algorithm to find the $k$-th most frequent IP-address, where $n$ is the total number of IP addresses in the stream.

Solution. Take the first $k$ packets of the input stream, and construct a min-heap of size $k$, where IP addresses are inserted into the min-heap and ordered by their frequency. For each IP address in the input, if the frequency is greater than or equal to the frequency of the address at the root of the heap, remove the root, insert the new address as the new root, and perform min-heapify. Else, if the frequency of the new address is less than or equal to the frequency of the root, do nothing. After processing all input, return the address at the root of the heap.

Proof of correctness. We want to show that the algorithm, as described above, returns the IP address of the input that is the $k$-th most frequent. Consider any address that enters the heap. By construction, any address that enters the heap must have a frequency that is greater than or equal to some other address. Assume for the sake of contradiction that an address $a_{bad}$ with a frequency greater than order $k$ (i.e., less frequent than the $k$-th most frequent) remains on the heap at the termination of the algorithm. But because we always maintain a heap that has a maximum size of $k$, this implies that some address $a_{good}$ with frequency order less than or equal to $k$ (more frequent than the $k$-th most-frequent) is excluded from the heap. But by construction, it is impossible for $a_{good}$ to have been excluded, since it would have compared more frequent than $a_{bad}$, which, in turn, would also be more frequent than the root element by transitivity! This is a contradiction, so our algorithm must be correct.

Running time analysis. Constructing a min-heap from the first $k$ elements (unsorted) takes time $O(k)$. We maintain the heap at size $k$ by removing the root in constant time and inserting a new address at the root and percolating downwards. Since each address can be inserted in the heap as the root at most once, each address is percolated downwards to its final position in time $O(\log k)$. Since the input has $n$ addresses, our overall running time is $O(k + n \log k)$. But since $n \gg k$, we have a final complexity of $O(n \log k)$.
A Quick Introduction to Greedy Algorithms

Throughout the rest of the course, we will be discussing a fundamental paradigm called greedy algorithms. Much of these notes are adapted from CLRS Chapter 16.

**Definition** (Greedy Algorithms). A *greedy algorithm* obtains an optimal solution to a problem by making the choice that seems ‘the best’ at the moment. It is a heuristic strategy that does not work all of the time, yet for certain problems, it produces an optimal solution.

Greedy algorithms show up in many parts of computer science. We will see next week how we can use greedy algorithms to perform optimal data compression (Huffman’s Algorithm) and we will soon see how greedy algorithms can be used to find unique graph properties (Dijkstra’s Algorithm for shortest path and Prim’s/Kruskal’s Algorithms to find the minimum spanning tree).

**Greedy-choice Property**

The key ingredient to greedy algorithms is the *greedy-choice property*. This properties states that we can assemble a globally optimal solution by making locally optimal choices. This means that when we are considering a choice in our problem, we will always make the choice that is the best in our current situation without considering any future problems that we may encounter.

You can think of this as a ‘bottoms up’ approach. Greedy algorithms will solve sub problems one by one, choosing what is best at the current iteration, until it finds a globally optimal solution for the entire problem. For any greedy algorithm to be valid, we need to show that a greedy choice at each step yields a globally optimal solution. We can do this with the exchange argument.

**Definition** (The exchange argument). We first examine some globally optimal solution to our problem. We want to show how to modify this solution to substitute a greedy choice for some other choice in the problem that results in a similar but smaller sub problem. If we can show that the optimal solution to our problem includes our greedy choice along with the same optimal solution to a smaller subproblem, then we can ensure our greedy solution is correct.

If you want to learn more about greedy algorithms, please read CLRS Chapter 16.1 and 16.2 for a more in depth analysis.