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 $\lfloor i/2 \rfloor$.

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**

*Running times are given with respect to $n$, where $n$ is the number of elements in the binary heap.*

- **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 \textsc{Max-Heapify}(A, i)
\begin{align*}
l &\leftarrow \text{left}(i) \\
r &\leftarrow \text{right}(i) \\
\text{if } l &\leq A.\text{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.\text{heapsize} \text{ and } A[r] > A[\text{largest}] \text{ then} \\
&\quad \text{largest} \leftarrow r \\
\text{if } \text{largest} \neq i \text{ then} &\quad \triangleright \text{ One of children is larger. Swap and recurse.} \\
&\quad \text{swap}(A[i], A[\text{largest}]) \\
&\quad \text{MAX-HEAPIFY}(A, \text{largest})
\end{align*}

function \textsc{Build-Max-Heap}(A)
\begin{align*}
A.\text{heapsize} &\leftarrow A.\text{length} \\
\text{for } i &\leftarrow \lfloor A.\text{length}/2 \rfloor \text{ downto } 1 \text{ do} \\
&\quad \text{MAX-HEAPIFY}(A, i)
\end{align*}

The \textsc{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 \textsc{build-max-heap} is left as a discussion topic.

\textbf{Heapsort}

function \textsc{Heapsort}(A)
\begin{align*}
&\textsc{build-max-heap}(A) \\
&\text{for } i \leftarrow A.\text{length} \text{ downto } 2 \text{ do} \\
&\quad \text{swap}(A[1], A[i]) \\
&\quad A.\text{heapsize} \leftarrow A.\text{heapsize} - 1 \\
&\quad \textsc{MAX-HEAPIFY}(A, 1)
\end{align*}

The \textsc{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 \textsc{heapsort} is also left as a discussion topic.

\textbf{Discussion Topics}

- What is the worst case running time of \textsc{max-heapify}? Why?

- Why does constructing a heap (\textsc{build-max-heap}) take linear time? What happens if we try to build a heap by running \textsc{insert} \( n \) times instead?

- Given that both \textsc{build-max-heap} and \textsc{heapsort} call \textsc{max-heapify} at least \( n/2 \) times, why does \textsc{heapsort} run in \( \Theta(n \log n) \) time and not \textsc{build-max-heap}?
Discuss insertion-sort, mergesort, quicksort, and heapsort. What are their relative advantages? When might one sorting algorithm be preferred over the others?

**Huffman Encoding**

Huffman coding is a common technique used for lossless compression of text. It uses relative character frequency to encode text such that the least possible space is taken on average. In this lab we will be focusing on encoding text into binary, though later you will learn other methods.

**Encoding Implementation**

Characters are encoded by runs of bits of non-constant length. The encoding is prefix free, which means that there doesn’t exist a character with a binary representation that is the prefix of another character’s representation. We represent an encoding as a trie, with each leaf representing a character, and the path from the root to the leaf describing its encoding. Each edge in the tree is assigned either 1 or 0. By convention, each left edge is 0. We can construct such a tree as follows:

1. As input, assume that we have access to the relative probabilities with which each character appears. For a given text to compress, this can be generated in a preprocessing step before the compression. Treat each character as a tree of one node, with weight equal to its frequency, and construct a min-heap of the trees. While the tree is not empty, we perform the following steps.
   1. Remove the lowest frequency item from the heap (tree A), then remove the new lowest frequency item (tree B).
   2. Construct a new tree by creating a root node with two children: A as the left child, and B as the right child.
   3. Add this tree back into the heap, with weight equal to the combined weights of A and B.

This process is complete when only one tree is left in the heap, our final Huffman coding tree.

**Encoding Length**

When transmitting an Huffman coded file, both the trie-backed encoding and the compressed contents have to be transmitted.

Relative to single character encoding schemes, Huffman’s algorithm produces an optimal encoding, that is, the compressed contents of the text will be the shortest length possible. Using the frequency with which each character appears, and the number of bits needed for each compressed character, we can find the length of the compressed text.

**Definition 1.** The number of bits to represent a character c is equal to depth of the leaf representing it, \(d(c)\) in the trie. The length, or number of bits to represent the entire text is therefore equal to the following expression.

\[
L = \sum_{c \in R} d(c) \times \text{freq}(c)
\]

Loosely speaking, Huffman Coding makes the term \(d(c) \times \text{freq}(c)\) vary less between characters than it otherwise would, and therefore minimizes this summed over all characters. Each character is represented with brevity consistent with its relative importance, or frequency, in the text.

**Decoding Implementation**

Because the representation of the encoding is included, decoding is simple. While processing the text, follow bit by bit through the trie until a leaf node is reached, then output that character.

---

1 This direction of this step is convention; the reverse is just as valid, but for this course we will expect you to abide by this standard.
Discussion Questions

What are pros and cons of Huffman Coding?

Testing Your Understanding

Answer the following questions about heaps and Huffman encoding.

Problem 1. Construct an optimal Huffman coding for the following alphabet and frequency table:

<table>
<thead>
<tr>
<th>Character</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>0.4</td>
<td>0.3</td>
<td>0.15</td>
<td>0.1</td>
<td>0.05</td>
</tr>
</tbody>
</table>

What is the average encoded character length for the above encoding?

Problem 2. Construct an alphabet $A$ with frequencies such that in an optimal Huffman coding there exist at least two encodings of length exactly $(n - 2)$, where $n$ is the size of the alphabet. $n$ must be at least 5.

Problem 3. Consider an indefinitely long stream of unsorted integers. We are interested in knowing the median (in sorted order) at any given time. How would we do this in an efficient manner?

Problem 4. Your task is to connect $n$ ropes with a minimum cost. You are given $n$ ropes of different lengths, and you need to connect these ropes into one rope. The cost to connect two ropes is equal to sum of their lengths. You need to connect the ropes with minimum cost.

For example if we are given 4 ropes of lengths 4, 3, 2 and 6. We can connect the ropes in following ways.

1. First connect ropes of lengths 4 and 3. Now we have three ropes of lengths 2, 7, 6.
2. Now connect ropes of lengths 2 and 7. Now we have two ropes of lengths 6 and 9.
3. Finally connect the two ropes and all ropes have connected.

Total cost for connecting all ropes is $7 + 9 + 15 = 29$. Give an algorithm which always finds the minimum cost of attaching the rope.

Problem 5. You have an alphabet with $n > 2$ letters and frequencies. You perform Huffman encoding on this alphabet, and notice that the character with the largest frequency is encoded by a 0. In this alphabet, symbol $i$ occurs with probability $p_i; p_1 \geq p_2 \geq p_3 \geq ... \geq p_n$.

Given this alphabet and encoding, is it true or false that there exists an assignment of probabilities to $p_1$ through $p_n$ such that $p_1 < \frac{1}{3}$?

Problem 6. 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.
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 sub problem, 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.