Heaps—Monday, October 29 / Tuesday, October 30

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 Max-Heapify(A, i)
l ← left(i)
r ← right(i)
if \( l \leq A.heapsize \) and \( A[l] > A[i] \) then
    largest ← l
else
    largest ← i
if \( r \leq A.heapsize \) and \( A[r] > A[largest] \) then
    largest ← r
if largest ≠ i then
    \( \triangleright \) One of children is larger. Swap and recurse.
    swap(A[i], A[largest])
    max-heapify(A, largest)

function Build-Max-Heap(A)
\( A.heapsize \leftarrow A.length \)
for \( i \leftarrow \lfloor A.length/2 \rfloor \) downto 1 do
    max-heapify(A, i)

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)
    build-max-heap(A)
    for \( i \leftarrow A.length \) downto 2 do
        swap(A[1], A[i])
        \( A.heapsize \leftarrow A.heapsize - 1 \)
        max-heapify(A, 1)

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?

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:

\[
T(n) = T(size \ of \ subtree) + \Theta(1)
\]

In the worst case, we want to examine the case where the size of the subtree we recurse on is maximal 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 half full (the right subtree's bottom level is empty).
Worst-case input to MAX-HEAPIFY.

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 $T(n) \leq T(\frac{2n}{3}) + \Theta(1) = O(\log n)$.

**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 $T(n) = \sum_{j=0}^{h} j2^{h-j}$ swaps.

Therefore,

$$T(n) = \sum_{j=0}^{h} j2^{h-j} = \sum_{j=0}^{h} j \cdot \frac{2^h}{2^j} < \frac{n}{2} \sum_{j=0}^{h} \frac{j}{2^j}$$

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

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

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.

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:

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 preform 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.

---

1This 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.
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.

**Discussion Questions**

What are pros and cons of Huffman Coding?

**Solution.** Pros: Huffman codings represents only the characters that are contained in the next, without wasting space in the encoding length for characters that are not present. Additionally, characters that occur most take less bits to encode each time.

Cons: It does not help with longer patterns in the text. Storing the symbol table takes up additional space. Therefore it isn’t appropriate for very small files where the table size would be significant. Encoding a text requires a preprocessing step to generate the table, does not work with a data stream.

**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?

**Solution.** The following tree would be produced:
The average length is given by:
\[ l_{\text{avg}} = \sum_{c \in R} d(c) \times \text{freq}(c) = 0.4 \times 1 + 0.3 \times 2 + 0.15 \times 3 + 0.1 \times 4 + 0.05 \times 4 = 2.05 \]

**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.

**Solution.** Multiple solutions exist. One possible is:

<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>0.6</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>Encoding</td>
<td>1 000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td></td>
</tr>
</tbody>
</table>

Represented as a tree:

```
A
/ \
/  \
/   \
B C D E
```

**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?

**Solution.** We can keep a min-heap and a max-heap simultaneously. The max-heap contains the smaller half of numbers and the min-heap contains the larger half of numbers. Maintain the following two invariants:

1. The difference in size of the max-heap and the size of the min-heap is at most 1.
2. The root of the max-heap is always less than or equal to the root of the min-heap.

For the first two elements of the stream, put the smaller element into the max-heap, and the larger element into the min-heap. Whenever a new element of the stream is encountered, compare it against the root of the max-heap (this is an arbitrary choice, we could have compared to the min-heap root). If it is smaller than the max-heap root, insert in the max-heap. Otherwise, insert it into the min-heap. If invariant (1) is violated, remove the root from the heap of larger size and insert that newly-removed element into the heap of smaller size. To retrieve the median at any given time, if the number of total elements is odd, take the root of the heap with larger size; otherwise, take the average of the roots of both heaps.

**Proof of correctness.** The correctness of the computation of the median from the invariants is immediate. We want to show that our algorithm maintains these invariants. We leave justification of these facts to the reader.

**Running time analysis.** Let \( n \) be the number of elements seen in the stream. In this algorithm, we perform at most two insertions and at most one extraction from heaps of size at most \([n/2]\), which is a running time that is \( O(\log n) \). We can access the roots of the heaps for the median computation in constant time, so finding the median is \( O(1) \). For every element of the stream, we maintain our data structures in \( O(\log n) \) time. Since every element is stored internally, we use \( O(n) \) space.

**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 the following ways.

---

7
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 = 31. This is not the optimal way of connecting these ropes. Give an algorithm which always finds the minimum cost of attaching the rope.

Solution. If we observe the above problem closely, we can notice that the lengths of the ropes which are picked first are included more than once in total cost. Therefore, the idea is to connect smallest two ropes first and recur for remaining ropes. This is a very similar principle as in Huffman encoding. We put smallest ropes down the tree so that they can be repeated multiple times rather than the longer ropes.

Following is complete algorithm for finding the minimum cost for connecting n ropes. The total runtime is the same as Huffman, so \( O(n \log n) \). Let there be n ropes of lengths stored in an array \( len[0...n−1] \)

1. Create a min heap and insert all lengths into the min heap.

2. Do following while number of elements in min heap is not one.
   (a) Extract the minimum and second minimum from min heap
   (b) Add the above two extracted values and insert the added value to the min-heap.
   (c) Maintain a variable for total cost and keep incrementing it by the sum of extracted values.

3. Return the value of this total cost

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 \( p_1 < \frac{1}{3} \)?

Solution. This claim is false. We can use a simple proof by contradiction. Assume that there exists an assignment such that \( p_1 < \frac{1}{3} \). Look at the last step of the Huffman algorithm, that is the step when our two final nodes are merged into one node. Let these two final nodes be called \( x \) and \( y \). Because character 1 has an encoding length of 1, it must have been included in this step. WLOG let \( x \) be our single character 1. This is the final step of Huffman, so we know that \( p_x + p_y = 1 \). Given our assumption that \( p_1 < \frac{1}{3} \), this tells us that \( p_y = 1 - p_x \implies p_y > \frac{2}{3} \).

We know that because \( n > 2 \), \( y \) must be a group containing at least 2 characters. So, examine the time when \( y \) was created. Let the nodes which combined to \( y \) be called \( a \) and \( b \). We know that \( p_a + p_b > \frac{2}{3} \), which implies that \( \max\{p_a, p_b\} > \frac{1}{3} \). Here we have reached contradiction. We know Huffman uses the smallest two nodes at all steps, but at the step \( y \) was created node \( x \) was still available and unpaired. \( p_x < \frac{1}{3} \), so it would have been chosen instead of \( \max\{a, b\} \). This contradiction then proves the original claim.

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.

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 \(a_{\text{bad}}\) 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 \(a_{\text{good}}\) 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 \(a_{\text{good}}\) to have been excluded, since it would have compared more frequent than \(a_{\text{bad}}\), 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 \(O(n \log k)\). \(\square\)
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.