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:
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

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?

• 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?

• Given that both BUILD-MAX-HEAP and HEAPSORT call MAX-HEAPIFY at least n/2 times, why does HEAPSORT run in Θ(n log n) time and not 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?

**Testing Your Understanding**

Answer the following questions regarding implementations of binary heaps.

**Problem 1.** Consider the following array:

```
null 6 7 9 15 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?

**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?

**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?

**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?

**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.
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.