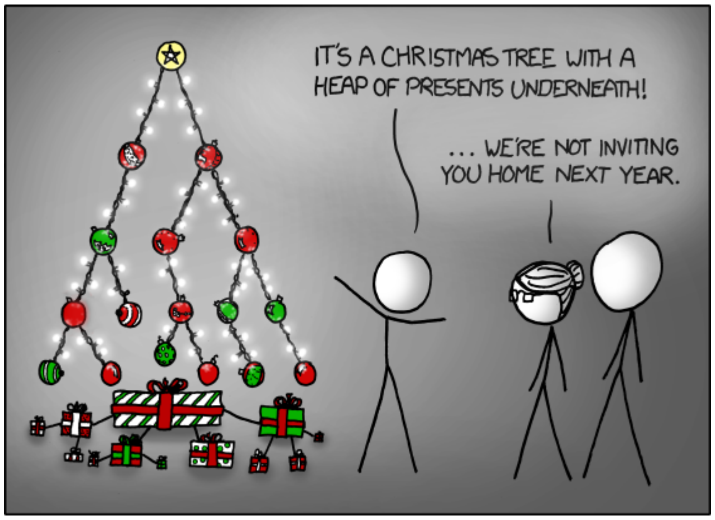
# BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

*CS-F211: Data Structures and Algorithms*

**Lab 11: Heaps and Heapsort**


IT'S A CHRISTMAS TREE WITH A HEAP OF PRESENTS UNDERNEATH!
... WE'RE NOT INVITING YOU HOME NEXT YEAR.
[1]

## Introduction

Welcome to week 11 of Data Structures and Algorithms! This week, we shall learn yet another important data structure known as the "heap". Heaps are commonly used in algorithms that involve sorting, searching, and prioritisation, such as Dijkstra's algorithm for finding the shortest path between nodes in a graph. At the heart of all these is the most essential operation that can be performed on a heap: "heapify". We shall then also learn a new sorting algorithm that uses heaps. Understanding how heaps and heapify work and how to implement them can significantly improve one's ability to design efficient algorithms and solve complex problems.

## Topics to be covered in this labsheet

- Introduction to the Heap data structure
    - Kinds of heaps
    - Heap property
- Implementation of Heaps
- Heapify
    - Using heapify to build a heap
- Heapsort
- Priority Queues

[1] https://xkcd.com/835/

## Introduction to the Heap Data Structure

The heap data structure is defined as a nearly complete binary tree that satisfies the heap property[2]. There are two kinds of heaps: max-heaps and min-heaps. In both kinds of heaps, a special property is satisfied by the underlying tree. This property is known as the *heap property*.

### Heap Property

Max-heap property: The child of a node has to have a key that is smaller than the node itself. In other words, the parent of a node has to have a key greater than or equal to the node itself.

Min-heap property: The child of a node has to have a key that is greater than the node itself. In other words, the parent of a node has to have a key smaller than or equal to the node itself.

Therefore, a heap satisfying the max-heap property is known as a max-heap, and likewise for a min-heap. We shall be restricting ourselves to max-heaps and max-heap property for the purposes of our discussion in this labsheet, although the discussion would be equivalent to that of a min-heap satisfying the min-heap property.

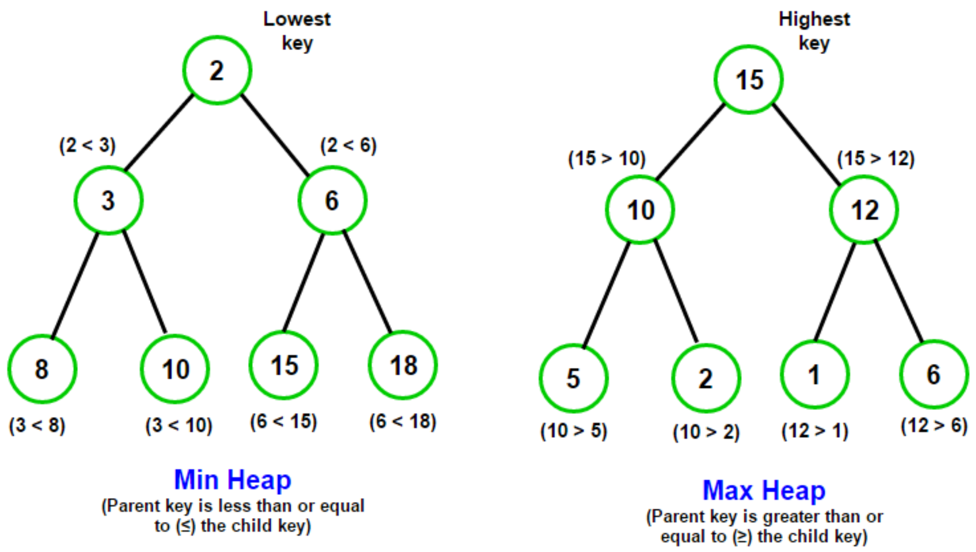


Figure 1: Illustrating heaps and the heap property

The above figure (Figure 1) displays the two kinds of heaps and how each satisfies their property.

[2] Note that whenever we refer to "heaps" in this labsheet we are always referring to "binary heaps".

## Implementation of Heaps

One might think that since a heap is a tree, and a tree would be best implemented as a graph-like data structure with pointers to children and so on. While that is one way to implement the binary tree that makes up the heap ADT, it is not the only way. Often, when faced with design choices during implementation, one goes with the simplest implementation wherever possible. In this case, we shall see how a heap *need not* be implemented using a graph data structure with nodes and edges. We shall learn how to implement the heap ADT using nothing but an *array*.

We can prepare our array such that each node in the binary tree corresponds to an element in the array. However, we need to come up with a scheme such that we are able to check and incorporate the heap property into the array implementation as well. This can be done by abiding by the following scheme:

1. The root of the tree corresponds to the first element in the array (index = 0).
2. The children of the root are found at the two positions after the double of the root's index, ie., the second and third elements (indices = 1 and 2, which are equal to 0*2+1 and 0*2+2 respectively) in the array are the children of the root.
3. Inductively repeat the above for all nodes in the heap, ie., for all elements in the array, till the nodes in the heap are exhausted.

Refer to Figure 2 for a visual representation of the above scheme for mapping a tree-heap to an array-heap.

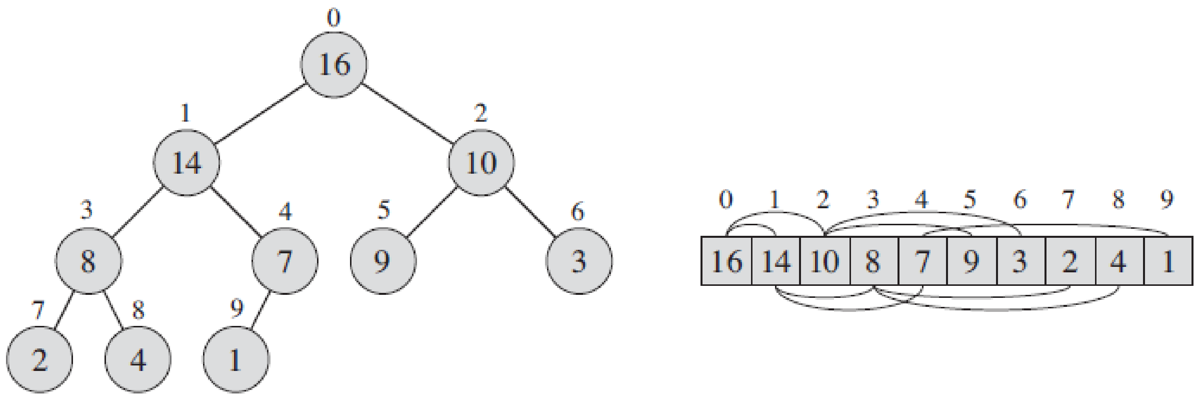


Figure 2: Demonstrating the mapping from tree to array

We shall see how easily functions can be implemented for navigating the binary tree, without having to store and allocate and deallocate or reallocate pointers. This adds to our convenience when dealing with heaps. While we are dealing actually with a normal dynamic array, we must always have in mind that this array is just an “encoding” for the underlying tree that it refers to. We have simply come up with a mechanism that stores it more conveniently for us. This approach can be generalised for any binary tree, and in fact can even be applied to any tree in general.

However, our implementation is not over just yet. Since a binary heap can have its lowest level partially filled (anywhere from 1 to $2^d$ nodes, where ‘d’ is the depth of the tree), we must account for that in our array. We shall add complete $2^d$ more space (for $2^d$ elements) to our array if a new level needs to be added to the tree at depth d. This will help our cause since repeated realloc calls will incur high time penalties (system calls are expensive; realloc is an encapsulated system call to the OS requesting for more memory), therefore we want to minimise our realloc calls while keeping the array size reasonable at any given instant. But what this means is that our array will have some garbage elements if the heap is not a complete binary tree.

To keep track of which elements are valid and which are not, we can simply maintain an integer (we can call it “heap size”) that specifies the last index up to which the elements in our array are valid heap elements, and beyond which are garbage.

Keeping the above points in mind, let us design the struct to implement the binary tree on which our max-heap will be built. Even though it is just a binary tree (at this stage), we shall call it the “heap” nonetheless.

```c
struct heap {
    int *data;
    int size;
    int capacity;
    int depth;
};
typedef struct heap* Heap;
```

It would help our goal of encapsulation if we create a function that creates such a struct for us and returns a pointer to it. Let’s call this function heap_create().

```c
Heap heap_create()
{
    Heap h = malloc(sizeof(struct heap));
    h->data = malloc(sizeof(int));
    h->size = 0;
    h->capacity = 1;
    h->depth = 0;
    return h;
}
```

Task 1: Write a function add_to_tree() that takes as input a binary tree that has been implemented just like the heap struct above and an integer and places that element at the next free slot in the binary tree keeping it nearly complete. What this means is that a new level cannot be introduced into the tree unless the previous level has been completely filled with elements.

Now, implement a binary tree using nodes and edges just like you did in Lab 9 for BSTs. Write a similar add_to_tree() function for this implementation. Instantiate two different binary trees, one using each implementation, and insert ten elements into them (these ten elements can be hardcoded into your main() function for the purposes of this exercise). Now, measure the total memory occupied by each implementation (note that direct use of sizeof will not work for attributes that are stored as pointers, and you will need to create your own functions to obtain the correct sizes for the same). Compare and contrast the memory usage of both implementations.

Task 2: This scheme we discussed for implementing the binary tree of the heap directly entails three constant-time operations that will help us navigate the "tree" when it has been implemented in the form of an array. They are:

- `parent(Heap h, int node)`
- `left_child(Heap h, int node)`
- `right_child(Heap h, int node)`

Implement the above functions as described earlier. [Hint: They can be implemented in effectively only one line of code, excluding validity checks.]

## Heapify

What we have implemented so far is no more than a binary tree. The only thing separating a binary tree from a heap is the heap property. To introduce the heap property into a binary tree (that is not yet a heap), we shall make use of the famous *heapify* routine. This operation is described as follows. It takes in as input a binary tree (say, the array implementation) and a node (an index) in the tree. Under the pre-condition that the left and right subtrees of the given node are already satisfying the heap property, the routine makes the heap property satisfies at the given node's level too by recursively making the largest node "float down" till the heap property is satisfied by the subtree rooted at the given node's position.

This process is illustrated in Figure 3. Notice how the node with key 4 keeps floating downward (through swaps) while the nodes with higher keys get floated up, till the max-heap property is satisfied.

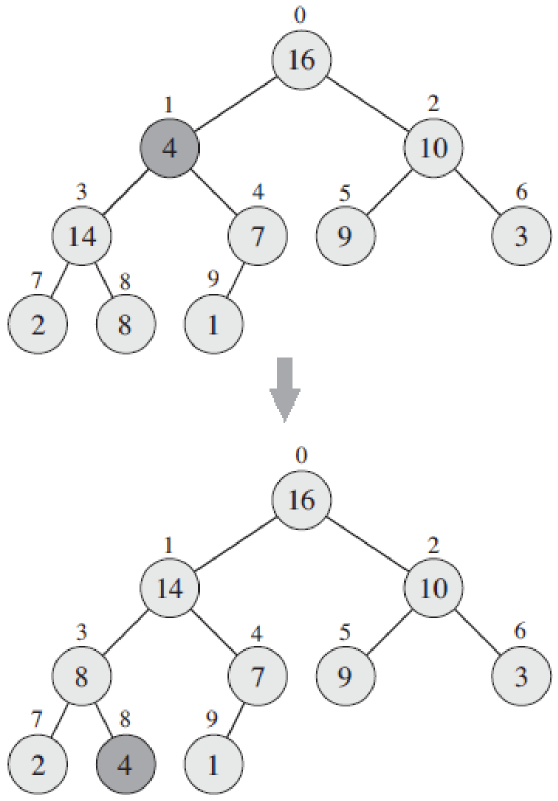


Figure 3: Illustrating the heapify procedure when called on node at index "1"

This function can be implemented as follows:

```
void max_heapify(Heap h, int index)
{
```

```c
    int left = left_child(h, index);
    int right = right_child(h, index);
    int largest = index;

    if (left < h->size && h->data[left] > h->data[largest])
    {
        largest = left;
    }

    if (right < h->size && h->data[right] > h->data[largest])
    {
        largest = right;
    }

    if (largest != index)
    {
        int temp = h->data[index];
        h->data[index] = h->data[largest];
        h->data[largest] = temp;
        max_heapify(h, largest);
    }
}
```

The worst-case time complexity of the heapify operation is O(log n), where n is the number of nodes in the heap. Notice how log n is just the height of the tree.

The heapify operation does not convert the entire binary tree to a heap. It only prepares a subtree that satisfies the heap property – that too, assuming that the left and right subtrees of the given node are already satisfying the heap property. Therefore, this in and of itself does not help us construct a heap.

To build an entire heap from a binary tree, we would need to make several calls to the heapify function in a *bottom-up* fashion. What this means is that we would call the heapify function on the leaf nodes of the tree, and then on the level above the leaves, ie., their parents, and so on.

Task 3: Write a function build_max_heap() that takes as input an array of integers that represents a binary tree and outputs another array that represents the heap that would be formed from the binary tree. Your function obviously should call the max_heapify() function within it.

Contrary to common sense, the build_max_heap() operation actually runs in O(n) time instead of the intuitively expected O(n log n). A rigorous proof of this can be found in your reference book.[3]

Task 4: Write a function that takes as input a heap (an array) and a particular level or depth, and returns the total number of nodes present in the heap at that depth.

Home Exercise 1: Write functions min_heapify() and build_min_heap() that do the min-heap equivalent of the aforementioned functions.

[3] Refer to pages 157-159 of Cormen T.H., Leiserson, C.E., Rivest, R.L., and C. Stein. *Introduction to Algorithms*, MIT Press, 3rd Edition, 2009.

## Heapsort

One of the most important reasons to study the heap data structure is to gain an understanding of the heapsort algorithm as well. Like merge sort, but unlike insertion sort, heapsort's running time is O(n log n). Like insertion sort, but unlike merge sort, heapsort sorts in-place: only a constant number of array elements are stored outside the input array at any time. Thus, heapsort combines the better attributes of the two sorting algorithms we have already seen.

The heapsort algorithm desires to build a max heap and then perform the heapify operation repeatedly. It starts out by calling the build_max_heap() function to create a max heap out of the given elements. At this stage (and at every iteration of this algorithm), the element at the top of the heap is the largest element in the heap. Therefore, we take this element and place it at its deserved location (at the end of the heap, which is also the end of the array). We would need to decrease the size of the heap so that the element we just considered and placed in its sorted location no longer interferes with our sorting process. We can then perform heapify on the root of the heap, and then repeat this process until our heap is emptied.

Following is an implementation of the above algorithm:

```c
void heap_sort(Heap h)
{
    h = build_max_heap(h);
    for (int i = h->size - 1; i >= 1; i--)
    {
        int temp = h->data[0];
        h->data[0] = h->data[i];
        h->data[i] = temp;
        h->size = h->size - 1;
        max_heapify(h, 0);
    }
}
```

Refer to Figure 4 for a pictorial illustration of how exactly the heapsort algorithm works to sort our array. Note that Figure 4 starts out with the max heap equivalent of a sample input array.

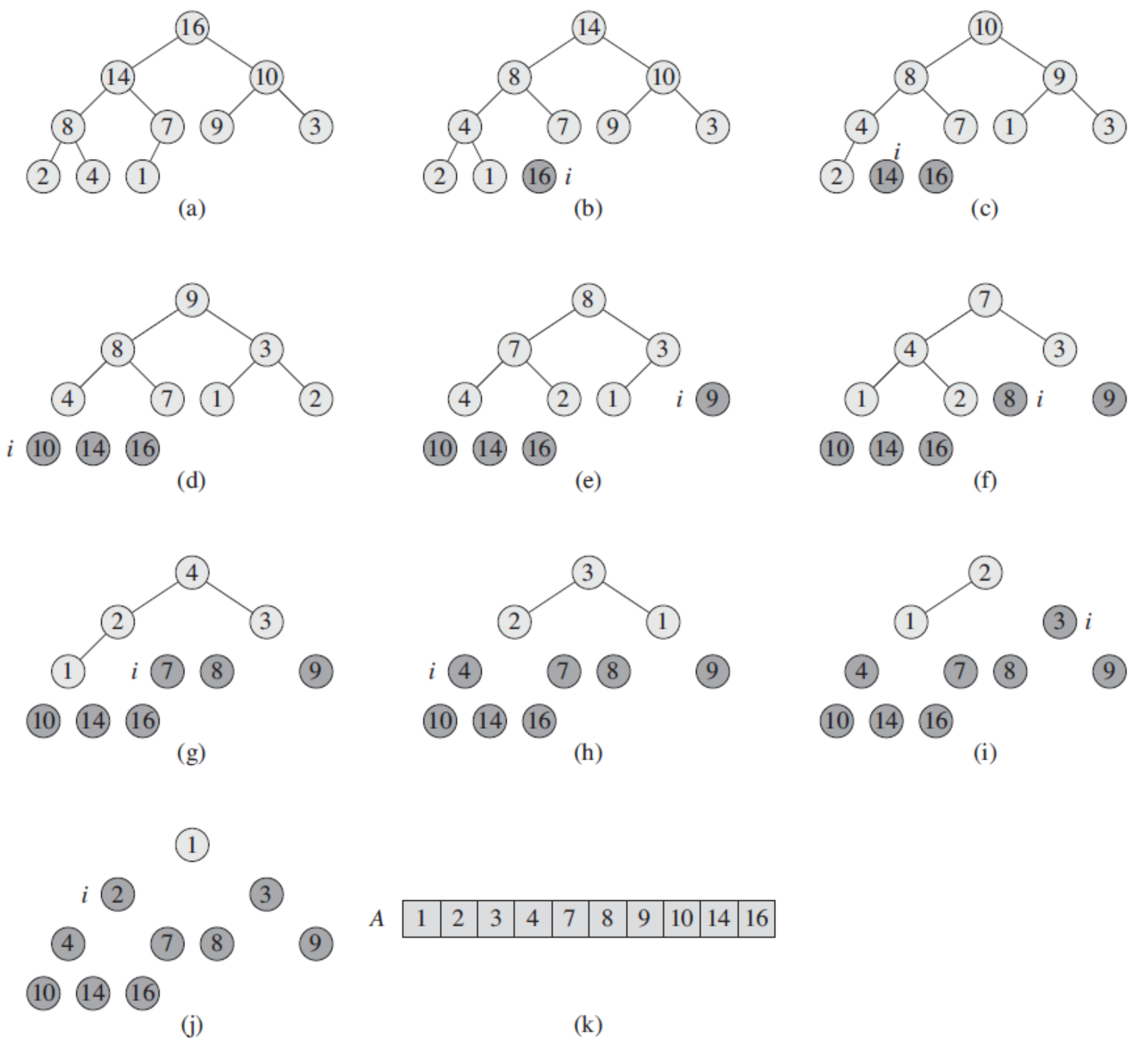


Figure 4: The working of heapsort. (a) The max-heap data structure just after build_max_heap. (b)–(j) The max-heap just after each call of max_heapify, showing the value of i at that time. Only lightly shaded nodes remain in the heap, the rest fall outside the "heapsize". (k) The resulting sorted array A.

Task 5: You have been provided with the set of files having file names *datX.csv* where *X* stands for the input size (as in earlier labs). Recollect that **struct person** was defined as follows:

```
struct person
{
    int id;
    char *name;
    int age;
    int height;
    int weight;
};
```

These files contain comma-separated entries for the details of the students. Write a program to read the data from these files and store them in a dynamically allocated array of `struct person`. In the previous labs, you had seen the performance of the other sorting algorithms on these files. Now, modify the **heap sort** algorithm discussed in this labsheet to sort arrays of `struct person` based on the **`height`** field. This would require you to modify *every* function discussed so far because now you need to accommodate an array of structs within the heap. Plot the time taken and maximum heap space (not to be confused with your heap data structure) utilised and observe how they vary with the size of the input. Report the comparative performance of the heap sort algorithm against the earlier algorithms.

## Priority Queues

A heap can function as an efficient priority queue as well. This is another useful application of heaps. As with heaps, priority queues come in two forms: max-priority queues and min-priority queues. We will focus here on how to implement max-priority queues, which are in turn based on max-heaps.

A priority queue is a data structure that maintains a set of elements *ordered* based on some priority associated with a key field of the elements, to enable efficient retrieval of the maximum (in case of a max-priority queue) or minimum (in case of a min-priority queue) element from the queue. There are implementations of priority queues that do not use heaps, for instance, a simple array implemented with insertion sort. But the heap implementation naturally leads to the idea of a priority queue, which also makes it more efficient compared to alternate implementations (we shall discuss this soon).

The name "priority queue" is no coincidence. You can correlate it with the queue data structure that you learnt earlier in the course. The only difference here is that a normal queue orders its elements in the sequence that they were entered into the queue, whereas a *priority* queue uses some characteristic property (known as the "key") of the elements to order them differentially regardless of the sequence in which they were enqueued. In other words, we can also claim that a "normal" queue is just a priority queue where the priority key is the "time spent in the queue" (the element that has spent the most time in the queue, ie., the element that was the first to be enqueued among into the queue, would be dequeued before the other elements).

A typical max-priority queue supports the following operations:

a. `INSERT(P, x):` Insert the element x into the priority queue P.
b. `MAXIMUM(P):` Return the element with the maximum key currently present in the priority queue P.
c. `EXTRACT_MAXIMUM(P):` Return the element with the maximum key currently present in the priority queue P and dequeue/remove it from the queue.
d. `INCREASE_KEY(P, x, k):` Increase the key associated with element x to a new value k, where k is assumed to be greater than the previous key value of x.

Our goal is to implement the necessary data structure for a priority queue and also implement these operations, ideally, such that they run efficiently. Let us see how we might use a heap to do the same.

Firstly, it should feel intuitive to you how a max-heap could be used to implement a max-priority queue. A max-heap always holds the max element at the top of the heap, and that is the element with the highest priority that we want to retrieve at any point from our max-priority queue. If we remove that element from the top of the heap, we have seen that we can perform heapify on the root to prepare a heap, yet again, from the remaining elements. In this manner, we always have our max-priority element on the top of the heap, and therefore we can use this implementation. Equivalently, a min-heap can be used to implement a min-priority queue.

The alternative to using a heap would be to use standard arrays and repetitive sorting (using a standard sorting algorithm like insertion sort, for instance), which is very much a noob solution. This would cost us O(n log n) time after each retrieval, in the worst case; whereas heapify allows us to do the same with only O(log n) time post each retrieval, in the worst case.

Therefore, we can say that a heap *is* a priority queue (we do not even need to create a separate struct for the priority queue!). This is not an exaggeration, in fact, people actually end up using the two terms synonymously at times. So, now, we only need to write the functions that perform the priority queue operations on a heap.

Task 6: You have been provided with a file *priorities.c* that contains some half-implemented priority queue functions (on a heap). Complete the implementation of priority queue on that file taking help of the other functions that you have implemented so far in this labsheet.

Note: In general, whenever a priority queue is used, the elements of the queue are structs containing a bunch of fields. This means that the underlying tree is also implemented as an array of structs. One of the attributes of the struct will be used as the key. For the purposes of this question, however, you can use simple integer elements with the integer itself as the key.

Home Exercise 2: Implement a min-priority queue just like the max-priority queue.

Home Exercise 3: Consider the following situation. You are given an integer array *gifts* that denotes the number of gifts present in various piles. This means that gifts[i] denotes the number of gifts present in the $i^{th}$ pile. Every second, you greedily find the pile that has the most number of gifts and pick enough gifts from the pile to always leave behind the square root of the number of gifts initially present in the pile (if the square root is fractional, you leave behind only the integer part).

Write a function *pick_gifts()* that takes as input the array and an integer *k* and returns the total number of gifts remaining in the piles after *k* seconds have passed.

Home Exercise 4: Write a function that takes as input an array containing *k* sorted linked lists (each list contains at most *n* elements), merges them all into one single sorted linked list, and returns that linked list. Use a priority queue to solve this problem. [This problem can be solved with a time complexity of O(kn log k).

Home Exercise 5: Suppose for a moment that you are an engineer working at **Spotify**. Your colleagues in the data analytics team have done thorough research and devised an algorithm that assigns a **score** to every song for every user (based on how much they are expected to like that song), and these scores are updated regularly. Now, suppose that a candidate user opens up a random playlist they find on the Spotify web and starts playing it on **shuffle** (this means that the order of the songs in the playlist can be randomised). In this scenario, Spotify thinks that it might be more profitable to play songs in decreasing order of the scores assigned to them by the aforementioned algorithm. Arranging them in this order makes it quite likely that the songs will be played in some seemingly random order, but if the analysis is correct, the user is more likely to enjoy the songs than a random shuffle (because the first couple of songs would be more to their liking, as per our analysis) if they listen to only a couple of songs from that playlist. To make the "random" feeling even better, their algorithm also perturbs the score a little bit in some random direction every so often, so that the user does not get suspicious about this internal data manipulation.[4]

For a sample user, you are given a file *playlist.txt* that contains comma-separated fields in the following format: <artist; song; priority>, where the **priority** contains the priorities that have been calculated for these songs for this user by the data analytics team. This file represents a typical playlist on Spotify.

Assuming that the user puts this playlist on shuffle, your task is to read the .txt file and insert each song one by one into a priority queue. Thereafter, your priority queue would be usable when the user decides to, say, skip a song, or when one song gets over, etc. For demonstration purposes, extract each song from the priority queue one by one in the order dictated by the priority queue, and output that to the console.

[4] This is a completely fictitious example made up for demonstrating the possible uses of the concepts learnt in this labsheet. Priority queues are the go-to data structure wherever the concept of "scheduling" or "differential selection" comes up, just like in this exercise.

Spotify has not revealed any of these actual technical details to us. One can feel free to interchange "Spotify" with "Youtube Music", "Wynk", "Saavn", or any of their favourite music streaming service apps, as far as we are concerned.