Lab 12 - Heaps

CS 251, Fall 2024

Copyright Notice

Learning Goals

By the end of this lab, you'll:

- Understand how heaps are implemented using a vector.
- Apply a minheap to efficiently solve a real-world problem like computing top-k values on a list.

Starter Code

lab12-starter.zip

Tasks

For this lab worksheet, work on the tasks in the **given order**. We strongly encourage collaboration in labs. Work with your peers and ask your TAs questions!

This lab will be a fully coding lab in which you will implement a heap data structure. As seen in lecture 31, heaps (a kind of tree) can be implemented using an array or vector. In this lab, we will use vectors to save the data in the heap.

Part of the goal of the lab will be to apply heaps in a problem with real-world data. We will use them to save information about CTA train stations and determine what are the trains that have received the most ridership in the past years.

To do this, in `heap.h` we have defined for you a `struct` that holds the information we want to save about train stations.

Throughout the lab we will use the following definition for the `TrainStation` type of value that will be saved in our heap.

```
struct TrainStation {
  int id;
  string name;
  int ridership;
};
```

Since we are interested in determining the stations with the highest ridership, we will save `TrainStation` values in the heap according to their ridership values.

Thus, when inserting into the min-heap we will implement, a `TrainStation t1` will be deemed “less than” another `TrainStation t2`, if `t1.ridership < t2.ridership`. Thus, throughout the lab, when inserting into or removing from the heap, comparisons between `TrainStation`s will be done based on the ridership.

As you can see from `heap.h`, our heap will save the data in a `vector<TrainStation>` called `data`. Recall from lecture that a heap can be saved using an array (or vector) because it contains no gaps in its structure.

Task 1: Implement peek and insert functions

For these first two tasks you will implement the functions we ask of you in `heap.cpp`.

peek()

The first thing you will implement is the `peek` function.

`TrainStation MinHeap::peek()`

Implement this function which returns the minimum number in the heap. Where is this value saved in the heap?

After implementing the `peek` operation you will now implement the `insert` function.

insert(TrainStation ts)

```
void MinHeap::insert(TrainStation ts)
```

Implement this function which inserts the `TrainStation ts` in the heap.

Take time to review from the lecture slides, how insertion happens in the heap (in lecture it was called `enqueue`). The values first inserted at the end of the heap (which vector operation should you use?). Then, the value is sifted up until its proper place. Recall from lecture that this function makes use of the `_siftUp` function. In lecture we described the pseudocode for this operation as:

```
siftUp(node) {
    while (node's is smaller than parent) {
        swap node with parent
    }
}
```

In our program, the idea of the pseudocode algorithm is the same, but we will pass to `_siftUp` the index of the node where the value we are sifting up is stored. Thus, calling `siftUp(7)` will look to sift up the value currently stored at index 7 in our heap.

`void MinHeap::_siftUp(int index)`

Implement this function which sifts up the `TrainStation` that is stored at the given `index`.

After correctly implementing the `peek`, `insert`, and `_siftUp` functions you can test what you have so far by using the command `make test_insert`.

Task 2: Implement the remove function

remove()

`TrainStation MinHeap::remove()`

Implement this function which removes the minimum `TrainStation ts` in the heap (i.e. the `TrainStation` with the least ridership).

Take time to review from lecture how this remove function works (in lecture it was called `dequeue`). The value from the root is saved for returning. Then the value at the end of the heap is put at the root. Then this new root value is sifted down to its proper position. Recall from lecture that this function makes use of the `_siftDown` function. In lecture we described the pseudocode for this operation as:

```
siftDown(node) {
    while (node's data is greater than either child's data) {
       Swap node with smaller child
    }
}
```

In our program, the idea of the pseudocode algorithm is the same, but we will pass to `_siftDown` the index of the node where the value we are sifting up is stored. Thus, calling `siftDown(0)` will look to sift down the value currently stored at index 0 in our heap.

`void MinHeap::_siftDown(int index)`

Implement this function which sifts down the `TrainStation` that is stored at the given `index`.

You should note that the `remove` function does return the `TrainStation` that is being removed. Also, for proper implementation of the heap, the value at the end of the list needs to be removed from the vector (which function can you use to safely remove from the end of a vector?).

After correctly implementing the `remove`, and `_siftDown` functions you can test this function by using the command `make test_remove`.

Task 3: top-k using heaps

In class we described how we can use heaps to solve the "top-k problem" using heaps. The top-k problem looks to find the top k values in a list or set of values. If n is the size of the list, the naive approach of solving the problem would sort the n values in the list and return the first k values in the sorted list. This approach has a worst case runtime of O(n log n). The min-heap can be used for solving the problem in O(n log k) time, instead of the O(n log n) approach of the naive approach. k is usually *much* smaller than n, so this is good! For this last exercise of the lab, we will use a min-heap to compute the train stations that have the highest ridership over the years.

We can use a min-heap to solve this problem by iterating over the list and inserting all values in the min-heap. However, every time the size of the heap is k+1, we remove the minimum element from the heap. That way, by removing the minimum value from those k+1 values, we are, at all times, keeping in the heap, the top k elements we have seen so far.

The top-k function that you will implement is located in `heap_main.cpp`. It receives as input a vector of `TrainStation`s called `allStations` and returns another vector of `TrainStation`s that only contains the `k` `TrainStation`s with the highest ridership in the given list.

```
vector<TrainStation> topK(vector<TrainStation> allStations)
```

Implement this function based on the description that we gave above.

Once you finish your implementation, you can run your program by using the command: `make run_heap`. Since the dataset is large, the program may take around 10 seconds to complete (at least in Daniel's computer that was the case). The output of your program will look as follows:

```
Top 10 stations in terms of ridership:
1. Lake/State - 109631418 rides
2. Clark/Lake - 107422778 rides
3. Chicago/State - 98880002 rides
4. Belmont-North Main - 80927000 rides
5. Fullerton - 79744689 rides
6. 95th/Dan Ryan - 78547128 rides
7. O'Hare Airport - 74496206 rides
8. Grand/State - 74377038 rides
9. Roosevelt - 68184360 rides
10. Jackson/State - 65284124 rides
```

Deliverables

- A completely passing test suite of insertion and removal in the heap.
- A correct implementation of `topK` with the valid output of the top 10 train stations, based on ridership.