COSC 320 - Advanced Data Structures and Algorithm Analysis
Lab 3: Heapsort

Dr. Joe Anderson

Due: 19 September 2019

1 Objectives

In this lab you will focus on the following objectives:

1. Develop familiarity with a *Heap* data structure and its theoretical properties
2. Review array-based implementations of binary trees in `c++`

2 Preliminaries

In lecture we discussed a special type of binary tree, called a *heap*. Consider a (binary) heap with n elements labeled with $1, \ldots, n$, where the root is node 1. If we store this binary tree in the array A, with $A[0]$ being the root, we can access node i at $A[i-1]$, where the parent node of i (denoted by $parent(i)$) is found at $A\left[\lfloor i/2 \rfloor - 1\right]$, the left child of i (denoted $left(i)$) is at $A[2i-1]$, and the right child (denoted $right(i)$) is at $A[2i]$. This is illustrated by Figure 1 (also found in Section 6.1 of the CLRS textbook).

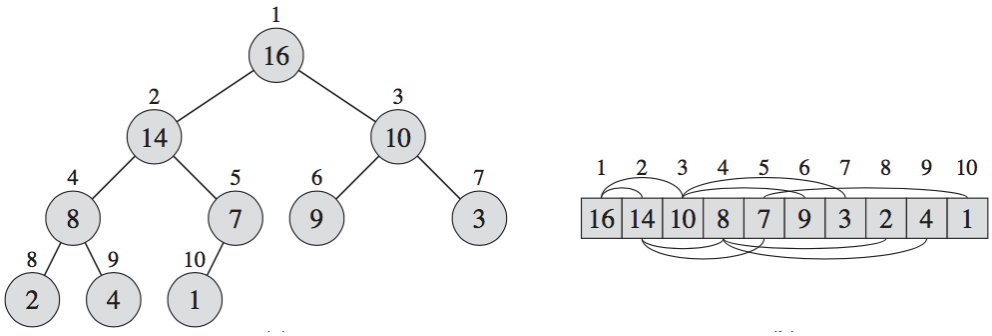

Figure 1

A Max-Heap is a binary tree in which all levels except the last are full, with the property that except for all nodes i except the root, $A[parent(i)] \geq A[i]$. This formulation is called the *max-heap-property*. If you were to reverse the inequality, it is the *min-heap-property*. In a max-heap (resp. min-heap), the largest (resp. smallest) element is located at the root.

Studying the above image of a heap, we can see that a traversal of the leaf elements would *almost* give the proper sort of the array. However, as we saw in lecture, we can repeatedly remove the largest element, fix up the heap, and repeat, in order to extract the elements in the proper order. The question now becomes, given n new elements, how quickly can we build the heap structure and, implicitly, sort the elements? The

answer, as discussed in class, is that we can build a heap with only $\Theta(n \log n)$ time, matching the asymptotic speed of mergesort!

In this lab, you will implement a heap data structure, and use it to perform sorting, and report on its performance. The array, A, which has a heap structure, will need two attributes: `length` and `heap_size`.

A useful construct with `c++` will be a `struct` of a form similar to the following:

```
struct Heap {
  int* arr; // the underlying array
  int length; // should always be the size of arr
  int heap_size; // will change based on which portion of arr is a valid heap
};
```

Furthermore, you may find it interesting or helpful to override the `[]` operator to re-index your array from 1:

```
struct Heap {
  int* arr; // the underlying array
  int length; // should always be the size of arr
  int heap_size; // will change based on which portion of arr is a valid heap

  // give the struct an [] operator to pass through access to arr
  // we return an int reference so we can assign into the structure,
  // otherwise it would only return a copy of the indexed element
  int& operator[](int i){
    // good idea to also check that 1 <= i <= length!
    return arr[i-1]; // so A[1] is the first and A[n] is the last
  }
};
```

3 Tasks

Note that the psuedocode is written for an array A that starts indexing at 1!

1. Put your code in a folder called "Lab-3". This folder will be zipped and turned in at the end.
2. Write the following methods (global, for now), which will perform the heap operations on a given array of positive integers, in-place.
   (a) `MaxHeapify` which will perform maintenance on the heap to maintain the max-heap property. This should take only $\Theta(\log n)$ time on an array of length n.
   (b) `BuildMaxHeap` which takes as input an un-ordered array and creates a max-heap from the elements using $\Theta(n)$ time.
   (c) `HeapSort` to take an array argument and sort the array in-place using the above subroutines. This procedure should take $\Theta(n \log n)$ time. The general algorithm (will be covered in lecture) follows the following procedure:
3. Recall that you will also need to track the value of $A.heap\_size$ during your program. Since arrays do not have member attributes in `c++`, you can solve this by passing around a second `int` along with A and its length.

Algorithm 1 MaxHeapify(A,i)

$l = left(i)$
$r = right(i)$
// Find the largest among node i and its children, and swap with i
if $l \leq A.heap\_size$ and $A[l] > A[i]$ **then**
    $largest = l$
else
    $largest = i$
end if
if $r \leq A.heap\_size$ and $A[r] > A[largest]$ **then**
    $largest = r$
end if
if $largest \neq i$ **then**
    // We may have violated the heap property, so recurse downward
    Swap $A[i]$ and $A[largest]$
    MaxHeapify$(A, largest)$
end if

Algorithm 2 BuildMaxHeap(A)

$A.heap\_size = A.length$ // the whole array will be a heap when we're done
for $i = \lfloor A.length/2 \rfloor$ down to 1 **do**
    MaxHeapify(A, i)
end for

Algorithm 3 HeapSort(A)

// Make sure the array is a valid heap, where the largest element must be $A[1]$
BuildMaxHeap(A)
for $i = A.length$ to 2 **do**
    // We know $A[i]$ is the largest among $A[1, \ldots, i]$, so move it
    // to the back, and consider it removed from the heap
    Swap $A[1]$ and $A[i]$.
    Set $A.heap\_size = A.heap\_size - 1$.
    // We moved one of the smaller elements to the root, so we have to clean up
    MaxHeapify$(A, 1)$
end for

4. Write a routine to take an array and print out its elements as a heap, so you can use this to visualize what is happening while you test your code. A good way to do this is to specify a node, i, and a depth, d along with the array pointer, then print only the subtree rooted at node i and only go to at most depth d.

   (a) In Figure 1 above, if you passed $(A, 2, 2)$ as (A, i, d), then your print routine might print

```
  14
 /  \
8    7
```

   (b) If instead you passed $(A, 2, 3)$ then it might print

```
     14
    /   \
   8     7
  / \   / \
 2   4 1  *
```

   (c) Alternatively, you can simply print each level of the tree left-aligned on its own line:

```
14
8 7
2 4 1 *
```

5. Include a `Makefile` to build your code.

6. Include a `README` file to document your code, any interesting design choices you made, and **answer the following questions completely and thoroughly**:

   (a) Summarize your approach to the problem, and how your code addresses the abstractions needed.

   (b) What is the theoretical time complexity of your sorting algorithm (best and worst case), in terms of the array size?

   (c) Test your sorting algorithm on different size and types of arrays, as you did with labs 1 and 2. Be sure to vary the parameters enough to use the observations to answer the next questions!

   (d) How does the absolute timing scale with the number of elements in the array? The size of the elements? Use the data collected to rectify this with the theoretical time complexity, e.g. what non-asymptotic function of n mostly closely matches the timings that you observe as n grows?

   (e) Aggregate your data into a graph of the complexity for the various array sizes, for example with a spreadsheet program like LibreOffice Calc or Microsoft Word.

   (f) How does the sort perform in different cases? What is the best and worst case, according to your own test results?

   (g) How could the code be improved in terms of usability, efficiency, and robustness?

4 Bonus

(10 pts) Write your heap methods to work on an array of *any* data type that has defined comparison operators using `c++` templates. Define a custom `struct` and overload the comparison operators for that `struct` to demonstrate that your templated subroutines work correctly on an array of that type. Be sure to document this code in your `Makefile`.

5 Submission

All submitted labs must compile with your provided `Makefile` and run on the COSC Linux environment.

Upload your project files to MyClasses in a single `.zip` file.

Turn in (stapled) printouts of your source code, properly commented and formatted with your name, course number, and complete description of the code.

Also turn in printouts reflecting several different runs of your program (you can copy/past from the terminal output window). Be sure to test different situations, show how the program handles erroneous input and different edge cases.