CS 202, Spring 2019
Homework #3 – Heaps and AVL Trees
Due Date: April 20, 2019

Important Notes

Please do not start the assignment before reading these notes.

- Before 23:55, April 20, upload your solutions in a single **ZIP** archive using Moodle submission form. Name the file as `studentID_hw3.zip`.
- Your ZIP archive should contain the following files:
  - `hw3.pdf`, the file containing the answers to Part 1,
  - `MaxHeap.h`, `MaxHeap.cpp`, `MinHeap.h`, `MinHeap.cpp`, `MedianHeap.h`, `MedianHeap.cpp`, `HuffmanHeap.h`, `HuffmanHeap.cpp`, `HuffmanQueue.h`, `HuffmanQueue.cpp`, `HuffmanCode.h`, `HuffmanCode.cpp`, `main.cpp` files which contain the C++ source codes, and the `Makefile`.
  - Do not forget to put your name, student ID, and section number in all of these files. Well comment your implementation. Add a header as in Listing 1 to the beginning of each file:

Listing 1: Header style

```
/**
 * Title: Heaps and AVL Trees
 * Author: Name Surname
 * ID: 21XXXXXX
 * Section: X
 * Assignment: 3
 * Description: description of your code
 */
```

  - Do not put any unnecessary files such as the auxiliary files generated from your favorite IDE. Be careful to avoid using any OS dependent utilities (for example to measure the time).

- **You should prepare the answers of Part 1 using a word processor (in other words, do not submit images of handwritten answers).**
- Please use the algorithms as exactly shown in lectures.
- Although you may use any platform or any operating system to implement your algorithms and obtain your experimental results, your code should work in a Linux environment (specifically on the **Dijkstra** 'dijkstra.ug.bcc.bilkent.edu.tr' server). We will compile your programs with the g++ compiler and test your codes in a Linux environment. Thus, you may lose significant amount of points if your C++ code does not compile or execute in a Linux environment.
- Please make sure that you are aware of the homework grading policy that is explained in the rubric for homeworks
- This homework will be graded by your TA, Alihan Okka. Thus, please contact him directly for any homework related questions.

Attention: For this assignment, you are allowed to use the codes given in our textbook and/or our lecture slides. However, you ARE NOT ALLOWED to use any codes from other sources (including the codes given in other textbooks, found on the Internet, belonging to your classmates, etc.). Furthermore, you ARE NOT ALLOWED to use any data structure or algorithm related function from the C++ standard template library (STL). **Do not forget that plagiarism and cheating will be heavily punished. Please do the homework yourself.**

Part 1 – 25 points

(a) [*5 points*] Insert the values $13, 9, 10, 21, 11, 16, 4, 12, 6$ into an initially empty AVL tree in order. Show **only the final tree** after all insertions. Then, delete $10, 9, 6$ in given order. Show **only the final tree** after all deletion operations.

(b) [*5 points*] Insert $15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1, 3$ into an empty min-heap. Show **only the final heap as a tree** (not as an array) after all insertions.

(c) [*15 points*] Let T_1 and T_2 be AVL trees such that the largest key in T_1 is less than the smallest key in T_2. Give an algorithm (in psuedocode) for procedure JOIN-AVL-TREES(T_1, T_2) that joins these two AVL trees. The runtime should be $\mathcal{O}(\log n)$, where n is the size of the resulting AVL tree. Justify in a few of sentences the correctness and efficiency of your algorithm.

Part 2 – Running Median (35 points)

Use the given file names and function signatures during implementation. Feel free to write helper functions to accomplish certain tasks but remember to give meaningful function names.

(a) [*7,5 points*] Implement an array-based max-heap data structure named as `MaxHeap` for maintaining a list of integer values that supports the following operations:

Listing 2: MaxHeap

```
void insert(int value); // inserts integer into heap
int peek(); // returns the value of the max element
int extractMax(); // retrieves and removes the max element
int size(); // returns the number of elements in the heap
```

Put your code into `MaxHeap.h` and `MaxHeap.cpp` files.

(b) [*7,5 points*] Implement an array-based min-heap data structure named as `MinHeap` for maintaining a list of integer values that supports the following operations:

Listing 3: MinHeap

```
void insert(int value); // inserts integer into heap
int peek(); // returns the value of the min element
int extractMin(); // retrieves and removes the min element
int size(); // returns the number of elements in the heap
```

Put your code into `MinHeap.h` and `MinHeap.cpp` files.

(c) [*20 points*] Design a `MedianHeap` data structure reusing `MaxHeap` and `MinHeap`, which supports getting median of the elements in $\mathcal{O}(1)$ time and inserting an element in $\mathcal{O}(\log n)$ time. If the length of the list is even, choose the larger value of the two middle elements as the median.

Listing 4: MedianHeap

```
void insert(int value); //inserts an element into MedianHeap
int findMedian(); //returns the value of the median
```

Put your code into `MedianHeap.h` and `MedianHeap.cpp` files.

Part 3 – Huffman Coding (40 points)

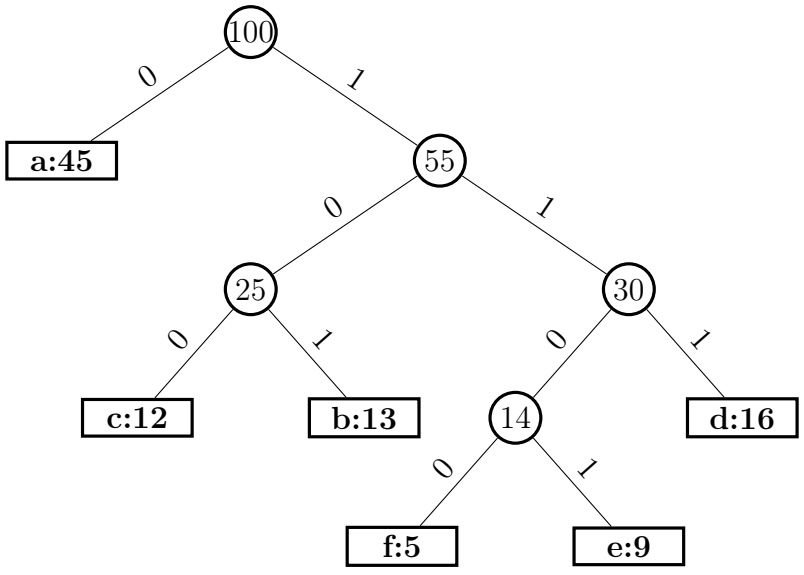
Huffman coding, in essence, is a lossless data compression algorithm. Consider the data which is a sequence of characters encoded in ASCII. In ASCII, every character is encoded with the same number of bits: 8 bits per character. The common characters, e.g., alphanumeric characters, punctuation, control characters, etc., use only 7 bits; there are 128 different characters that can be encoded with 7 bits. Huffman coding on the other hand, compresses data by using fewer bits to encode more frequently occurring characters so that not all characters are encoded with 8 bits.

Suppose that we have a 100,000 character data file that we wish to store. We observe that the characters in the file occur with the frequencies given by Table 1. That is, only 6 different characters appear, and the character 'a' occurs 45,000 times. If we store it without compression we would need 700,000 bits. However with Huffman coding, we can compress the file to $(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1{,}000 = 224{,}000$ bits.

Table 1: A character-coding

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
ASCII	1100001	1100010	1100011	1100100	1100101	1100110
Variable-length codeword	0	101	100	111	1101	1100

The main goal is to create a tree that corresponds to coding schemes that will be used to encode/decode a file. **Huffman trees** are a type of binary tree used in performing this kind of conversion. A Huffman tree includes all the characters of your data on the leaf nodes, with most frequent characters being closer to the root. The distance and path away from the root is used to determine what bits (what 0s and 1s) are used to represent each character. The Huffman tree of Table 1 is as follows.



In the pseudocode that follows, we assume that C is a set of n characters and that each character c in C is an object with an attribute $c.freq$ giving its frequency. The algorithm uses a min-priority queue Q, keyed on the *freq* attribute, to identify the two least-frequent objects to merge together.

Algorithm 1: HUFFMAN(C)

```
1  n = |C|;
2  Q = C;
3  for i = 1 to n - 1 do
4  |   allocate a new node z;
5  |   z.left = x = EXTRACT_MIN(Q);
6  |   z.right = y = EXTRACT_MIN(Q);
7  |   z.freq = x.freq + y.freq;
8  |   INSERT(Q, z);
9  end
10 return EXTRACT_MIN(Q)
```

The steps of creating the Huffman tree of Table I is given in the next page as an example [1]. Your first objective should be to implement the min-heap data structure as an array of pointers in which each pointer points to a `MinHeapNode`.

Listing 5: MinHeapNode

```
struct MinHeapNode
{
    char character; // An input character
    int freq; // Frequency of the character
    MinHeapNode *left, *right; // Left and right child
};
```

Put your min-heap code into `HuffmanHeap.h` and `HuffmanHeap.cpp` files. After you are finished, transform your heap into a min-priority queue implementing Huffman coding and put the codes into `HuffmanQueue.h` and `HuffmanQueue.cpp` files. Design the Huffman coding algorithm and put your code into `HuffmanCode.h` and `HuffmanCode.cpp` files.

Add a main function to the `main.cpp` file which reads an input file of characters and their frequencies (i.e., e 9), and writes their coding schemes (i.e., e 1101) on an output file. Take input and output file names as arguments. At the end, write a basic `makefile` which compiles all your code and creates an executable file named `hw3`.

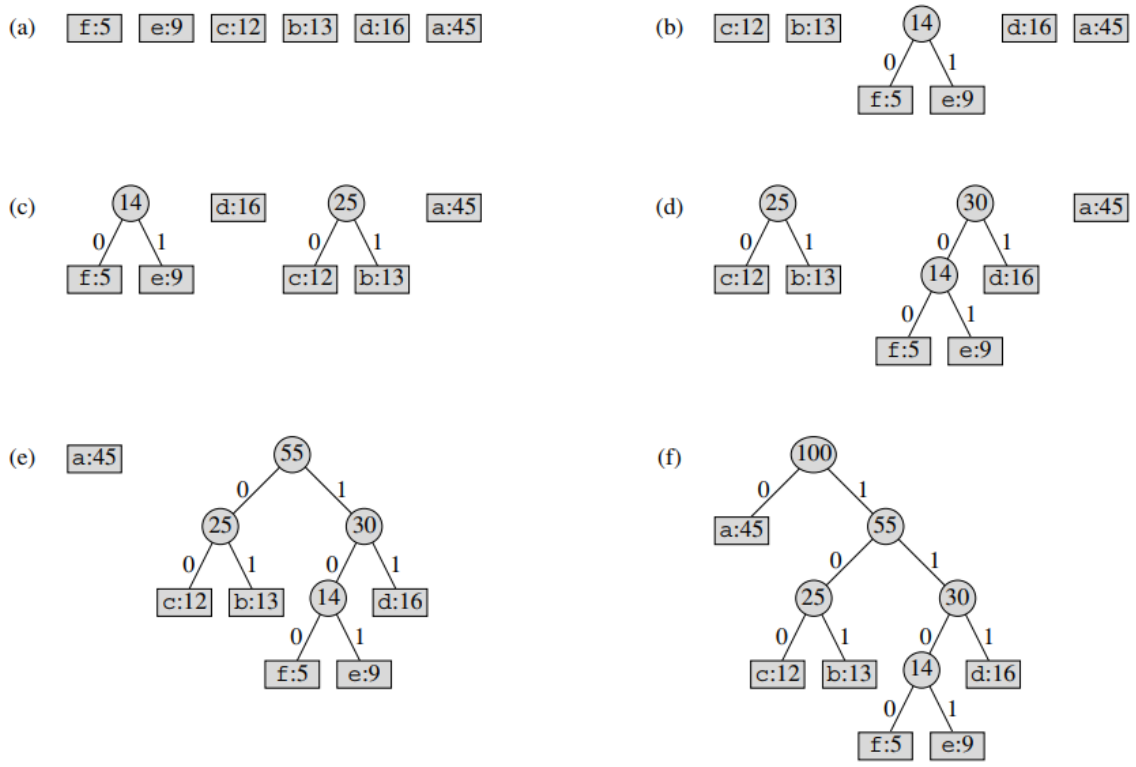


Figure 16.5 The steps of Huffman's algorithm for the frequencies given in Figure 16.3. Each part shows the contents of the queue sorted into increasing order by frequency. At each step, the two trees with lowest frequencies are merged. Leaves are shown as rectangles containing a character and its frequency. Internal nodes are shown as circles containing the sum of the frequencies of its children. An edge connecting an internal node with its children is labeled 0 if it is an edge to a left child and 1 if it is an edge to a right child. The codeword for a letter is the sequence of labels on the edges connecting the root to the leaf for that letter. **(a)** The initial set of $n = 6$ nodes, one for each letter. **(b)–(e)** Intermediate stages. **(f)** The final tree.

References

[1] T.M. Cormen, C. E. Leiserson, R. L. Rivest & C. Stein. *Introduction to Algorithms*, 3rd edition. The MIT Press. 2001. 432