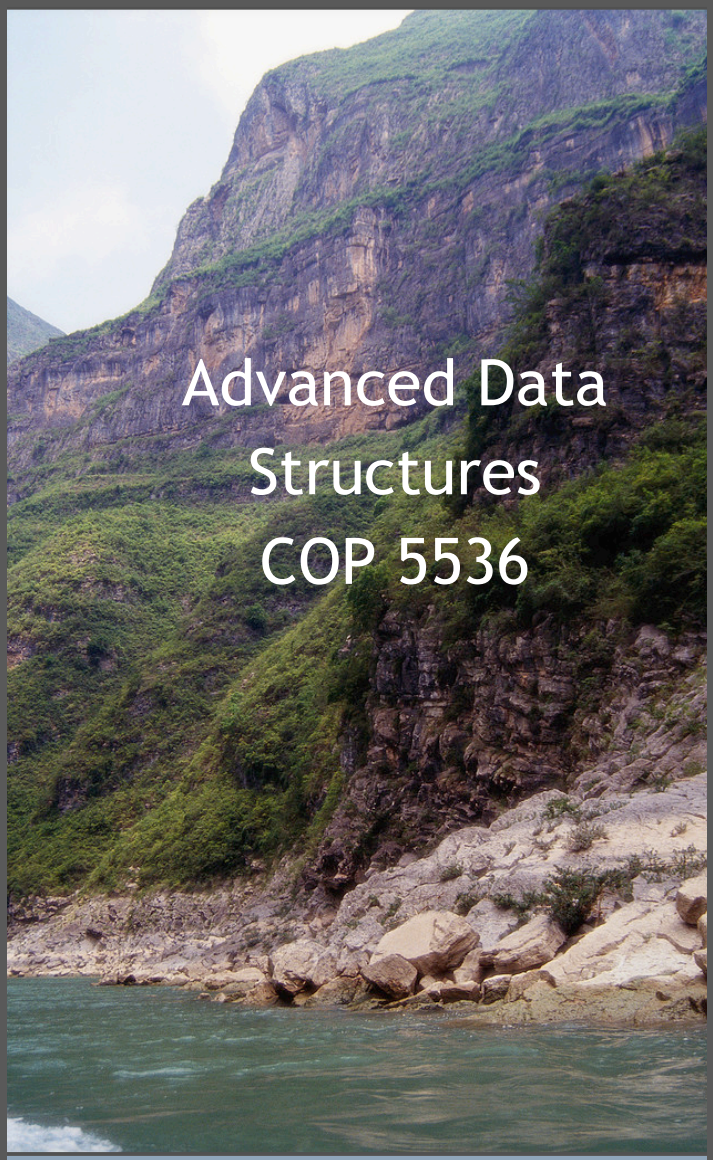
**Name:**
Akshay Chila
UFID: 19696721
E-Mail: achila@ufl.edu




# PROGRAMMING PROJECT

## Function Prototype and Structure of Program

The program involves Priority Queues implemented using three types of heaps, Binary Heap, 4 way cache optimized heap and Pairing Heap. As a large amount of data needs to be transferred, a Huffman code needs be built to reduce the data size.

The Huffman Tree was created using Binary Heap, 4 way Heap and Pairing heap. A run-time analysis was performed on the Huffman trees created with these three heaps. Based on the runtime analysis, the Huffman tree had the best runtime using Binary Heap for a large input file.

### Runtime analysis:

Time using BinaryHeap (milliseconds) = 658
Time using FourWayHeap (milliseconds) = 1215
Time using PairingHeap (milliseconds) = 2051

The four way cache optimized heap has a better runtime for remove min and insert operations, so has a faster execution time than the Binary Heap for building the heap. But since the greedy algorithm works faster for the binary heap, the overall runtime of the Binary Heap is better that the Four Way cache optimized heap for creating a Huffman tree.

There is a speed up of about 1.5 to 1.8 when sorting large number of elements using cache aligned 4 heap than a binary heap that begins at array position 0, as the siblings are in the same cache line for a cache aligned 4 way heap. Since the greedy scheme works faster on binary heap, it has a better overall time.

Time complexity with binary heap:

Initialize with n trees. Insertion takes O(n) time.

Remove 2 trees with least weight. 2(n-1) remove min operations take O(nlogn) time.

Insert new tree. (n-1) insert operations take O(nlogn) time

So, total time = O(nlogn)

Using the following class for the implementation:

```
public class BinaryHeap {

    private BinaryHeapNode[] heap;
    private BinaryHeapNode hroot;
    int size;


    public BinaryHeap(FrequencyTable ft){
         this.hroot = build_tree_using_binary_heap(ft);
    }
    public BinaryHeap(){
   }

        public static class BinaryHeapNode{
        private int data;
        private long freq;
    BinaryHeapNode left;
    BinaryHeapNode right;

        public BinaryHeapNode(int data, long freq){
                this.data = data;
                this.freq = freq;
        }

}
```

The Binary Heap includes a function that takes a frequency table as input and builds the required Huffman tree. It stores the heap nodes into an array and stores a reference to the Huffman tree root.

Each node contains data, its frequency and a reference to its left and right child.

**public void** minheap(**int** index)

The above function heapifies each element in the array so that it acts as a min priority queue which helps in building the huffman tree faster.

**public void** generateCode(BinaryHeapNode root,String st)

It also includes a function to generate code that is used to create a Code table for the data by traversing the Huffman tree. The code table is saved as a text file using the file writer. The code table along with the encoded data is sent to the Toggle server so that it can be used to decode the original data. This helps compress the data to a large extent by reducing it's size without any losses.

Encoder

The encoder takes a text file with integer as input and creates a frequency table based on the number of times a particular integer appears in the input file. The frequency table in then used to build the Huffman tree using the Priority queue implemented using Binary Heap. The Huffman tree is then traversed to build the code table that is sent to the Toggle server along with the encoded.bin file to interpret the encoded file. The encoder compresses the data so that it can be transferred at a faster rate.

## Decoder

The decoder uses the code table to create a Huffman tree on the server side. The Huffman tree can then be traversed to decode the encoded file received from the sender. Based on the above analysis the Huffman tree has a faster execution time. So using it on the server side would help in faster decoding.

Time complexity with binary heap:

Initialize with n trees. Insertion takes O(n) time.

Remove 2 trees with least weight. 2(n-1) remove min operations take O(nlogn) time.

Insert new tree. (n-1) insert operations take O(nlogn) time

So, total time = O(nlogn)

## Conclusion

Thus MyTube's goal of sending a large amount of data to the Toggle Server at a faster rate can be achieved using the Huffman tree encoding and decoding. The original data can be replicated on the server side at a much faster rate rather the sending the original text file.