Name: Vishal Kumar Gupta
UFID: 38159301 | Email: vishal191

COP 5536: Advance Data Structures — Project Report: Huffman Coding

**Language: Java 1.8.0_121**

**View codes here: https://github.com/vishalkg/huffman_coding**

**Code prototypes are in this color**

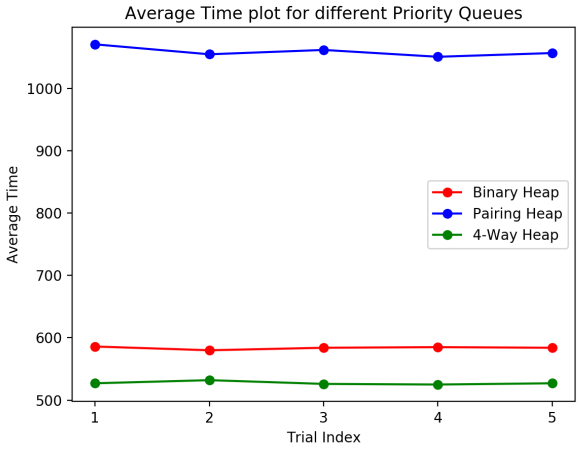
Three types of priority queues are implemented:

1. Binary Heap
2. Pairing Heap
3. 4-way cache optimized Heap

A detailed analysis has been done to check the performance of above mentioned priority queues. To be precise, with the same input, Huffman Tree is constructed using all the three priority queues and running time is averaged over 10 runs. The below graph shows the performance measure for 5 trials.

Key points from performance analysis:

- ✓ 4Way heap is always better than binary heap. An obvious explanation is: in 4Way heap we make comparisons as half of binary heap
- ✓ Height of cache-optimized 4Way heap is half of that of binary heap i.e $\log_d n$, here d = 4.
- ✓ Pairing heap performs the worst, one reason is the way meld operation is implemented: here the simpler but costly version is implemented. Details in code description!




Machine: Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz

It can be observed that, 4-way cache optimized heap performs the best with binary heap as next best choice. So, for Huffman coding in subsequent sections, 4-way cache optimized heap is used.

**Code Description:**

**1.** Node.java: represents one node of Huffman tree.

Class object has the following fields:

freq: to store frequency of message (if internal node, it is 0)
message: to store messages (if internal node, it is -1)
l: pointer to left child
r: pointer to right child

Class has following methods:

get_left, set_left, get_right, set_right, get_msg, set_msg, get_freq, set_freq

**2.** BinaryHeap.java: Binary min heap implementation using arrays

Class object has following fields:

data: array of Node objects
heap_size: to keep track of active heap elements

Class has following main methods:

insert: to insert a new element in the priority queue
del_min: to extract min element of min heap
manage_heap_upwards: used to assist del_min

manage_heap_downwards: used to assist insert

3. D_aryHeap.java:
   Class object has following fields:
   data: array of Node objects
   d: in our case it is 4, however the code is generic for other d also
   heap_size: to keep track of active heap elements

   Class has following main methods:
   insert: to insert a new element in the priority queue
   del_min: to extract min element of min heap
   manage_heap_upwards: used to assist del_min
   manage_heap_downwards: used to assist insert

4. PairNode.java: represents one node of pairing heap
   Class object has following fields:
   n: Node object to store pointer to existing trees in tree collection
   l: pointer to left sibling
   next: pointer to next node in the doubly linked list of siblings
   prev: pointer to previous node in the doubly linked list of siblings (points to parent in case of left sibling)

   Class has following methods:
   get/set_freq/msg: to get/modify frequency/message of root of tree stored in that node
   Other methods include get_left, set_left, get_prev, set_prev, get_next, set_next

5. PairingHeap.java: to implement priority queue based on PairNode
   Class object has following fields:
   root: pointer to root of pairing heap

   Class has following main methods:
   insert: to insert new pair node in pairing heap
   del_min: to extract minimum from min heap priority queue
   merge: to merge to PairNodes into one (used in both del_min and insert)

   Merge Scheme:

```
CurrentTree = first subtree
for (each of the remaining trees):
        CurrentTree = merge(CurrentTree, nextTree)
```

6. Gen_huffman_code.java: used to test the performance of different priority queues using single input file. Prototype:

```
class Gen_huffman_code {
    gen_codes (CurrentNode, code) {
        /* Recursive function for inorder traversal of huffman tree*/
        if external node
                record code
        else
                call gen_codes on left and right subtrees respectively with code+”0” or
                code+”1”
    }

    build_tree_using_binary_heap (freq_table) {
        min_heap = initialize heap with single node trees for all the messages with
        positive frequencies
```

```
            while (heap_size != 1) {
                    extract min and 2nd min
                    x = merge (min, 2nd min)
                    insert (x)
            }
            gen_codes(mean_heap.get_root(), empty string “”)
            return min_heap.get_root()
    }

    Same is done for build_tree_using_pairing_heap & build_tree_using_4way_heap

    build_freq_table (freq_data) {
            build an array where each index is a message and each index entry is its frequency
    }

    main {
            freq_data = read input file in an ArrayList
            freq_table = build_freq_table (freq_data)

            call build_tree_using_binary_heap (freq_table) 10 times and average the runtime
            of each iteration (Same for pairing heap and 4way heap)

            conclude which performs better
    }
}
```

**7.** encoder.java: takes in input set of messages, processes them and generates code_table.txt and encoded.bin. Prototype:

```
class encoder {
    fillNwrite_code_table (huffman_tree, empty code “”, code_table, BufferedWriter) {
            recursive function to fill code_table by traversing huffman_tree
            if external node
                    code_table[node.message] = code
            else
                    call fillNwrite_code_table on left and right subtrees respectively with
                    code+”0” or code+”1”

    }

    get_binary_data (freq_data, code_table) {
            buff = initialize a BitSet object
            bitIndex = 0
            for each entry in freq_data {
                    symbol = get entry code from code_table
                    for each bit in symbol {
                            set/clear buff at bitIndex
                            bitIndex++
                    }
            }
            return buff.toByteArray()
    }

    main {
            freq_data = read input file
            freq_table = Gen_huffman_code.build_freq_table (freq_data)
            huffman_tree = Gen_huffman_code.build_tree_using_4way_heap

            code_table = initialize empty array to store codes in a table
            fillNwrite_code_table (huffman_tree, empty code “”, code_table, BufferedWriter)

            generate binary data using get_binary_data(freq_data,code_table)
            write binary data to encoded.bin using FileOutputStream
```

```
    }
}
```

8. decoder.java: takes in input encoded.bin and code_table.txt to generate decoded.txt. decoded.txt should be same as sample input given in encoder.

**Building Huffman Tree:** Logic is simple, take a symbol and traverse the tree, starting at root, dropping one level for each bit of symbol. In case a node doesn't exists add a node. If the last bit of symbol is being processed, set the message field of node to the message corresponding to current symbol else, set the message field of the node to -1.
**Complexity analysis:** Let's say there are $n$ unique symbols in code_table.txt. This implies while loop in `build_huffTree_using_codeTable_txt` will run $n$ times. Also, if the maximum length of symbol present in code_table.txt is $m$, then `add_node` function will be called at max $m$ times. Also observe that, the height of Huffman tree is $m$ only. Therefore, $T(n) = O(nm)$

**Decoding encoded.bin:** After the Huffman tree is constructed from code_table.txt, the next task is to decode the encoded.bin back to original messages. Here also algorithm is simple: since the file is in binary format, reading all the data in one go in a byte array makes sense. Once the byte array is available, running a *for loop* once over all the bytes will be enough to recover all the messages. So, for every byte, we take one bit at a time using masking and descend the tree. Once an external node is encountered, we retrieve the message stored in that node, write it to decoded.txt and set our pointer to root. If we run out of bits for current byte, we take another byte and continue to traverse the tree.
**Complexity Analysis:** Here also, let's say the height of the tree is $m$, which is same as maximum length of symbol present in out code table. So, we will make at most m comparisons (to check if we encountered an external node) for retrieving a symbol. If the encoded.bin encodes $p$ messages (need not be unique), then our time complexity comes out to be, $T(n) = O(pm)$.

Prototype:

```
class decoder {
    build_huffTree_using_codeTable_txt (code_table.txt){
            huff = empty tree
            while (line from code_table.txt is not null) {
                    extract message and code from line
                    add_node(huff,message,code)
            }
    }

    add_node (huff,message,code) {
            /* Recursive function for adding nodes to huff on the way */
            if code.length == 1          //last bit of symbol being processed
                    add a new node as left/right if code = 0/1
                    return
            else
                    if code.at0Index == 0
                            if huff.left == null
                                    add a new node
                            // bits except 1st
                            add_node (huff.left,message,code[1:end])
                    else
                            if huff.right == null
                                    add a new node
                            //bits except 1st
                            add_node (huff.right,message,code[1:end])
    }

    main {
            huffman_tree = build_huffTree_using_codeTable_txt(code_table.txt)
            code_bin = read all bytes from encoded.bin
```

```
            curr = huffman_tree
            for each byte in code_bin
                  for each bit in currByte
                        if curr.message != -1        //external node
                              write message to decoded.txt
                              reset curr to root
                        else
                              get nextBit from currByte
                              if nextBit == 0
                                    curr = huffman_tree.left
                              else
                                    curr = huffman_tree.right
      }
}
```

A sample trial run of all the codes is as below:

```
vishalgupta:src $ make
javac -d . -classpath . Node.java
javac -d . -classpath . PairNode.java
javac -d . -classpath . BinaryHeap.java
javac -d . -classpath . PairingHeap.java
javac -d . -classpath . D_aryHeap.java
javac -d . -classpath . Gen_huffman_code.java
javac -d . -classpath . encoder.java
javac -d . -classpath . decoder.java
vishalgupta:src $ java Gen_huffman_code ../sample_input_large.txt
Reading input file ... Building Freq Table ... Done.
Building Huffman Tree using Binary Heap ...
Run:1.. 2.. 3.. 4.. 5.. 6.. 7.. 8.. 9.. 10..  Done
Average Time: 1078.0
Building Huffman Tree using Pairing Heap ...
Run:1.. 2.. 3.. 4.. 5.. 6.. 7.. 8.. 9.. 10..  Done
Average Time: 1718.0
Building Huffman Tree using 4-ary Heap ...
Run:1.. 2.. 3.. 4.. 5.. 6.. 7.. 8.. 9.. 10..  Done
Average Time: 695.0
vishalgupta:src $ java encoder ../sample_input_large.txt
Reading input file ...
Building Freq Table ...
Building huffman tree... Done.
Generating code_table.txt .. Done.
Generating encoded.bin .. Done.
vishalgupta:src $ java decoder encoded.bin code_table.txt
Building huffman tree from code_table.txt ..
Reading code_table.txt .. Done.
Reading encoded.bin .. Done.
Generating decoded.txt .. Done.

vishalgupta:src $ diff decoded.txt ../sample_input_large.txt
vishalgupta:src $
```