COP 5536 Spring 2017
Advanced Data Structures
Programming Project

Name:	Amrita Surve
UF ID:	3556 9083
Email:	asurve@ufl.edu

1. OBJECTIVE

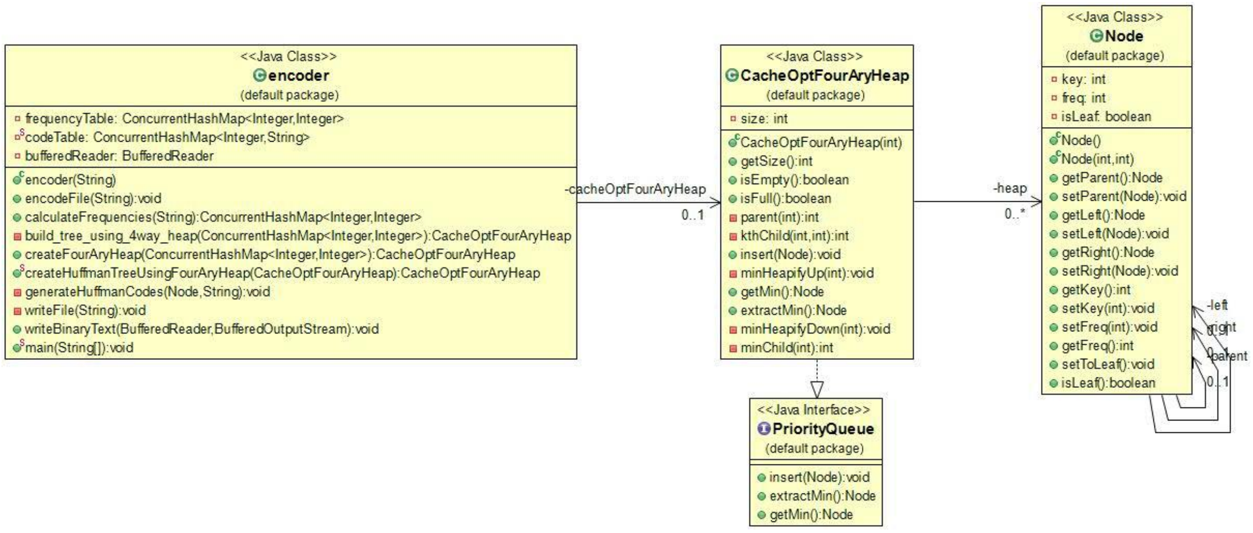
The intent of this project is to develop an efficient Huffman Compression scheme by comparing the run time for three data structures: Binary Heap, 4-way Cache Optimized Heap and Pairing Heaps and selecting the fastest data structure to implement efficient transfer of large amounts of data given in a specified format.

2. PROGRAM STRUCTURE AND EXPLANATION OF FUNCTIONS

The project can be divided into 'Encoding' and 'Decoding' phases. Following tasks are taken in each phase:

2.1 Encoding Phase:

Input:	Input file containing of integer values in range of 0-999999.
	Given samples: sample_input_large.txt, sample_input_small.txt
Output:	Encoded binary file: encoded.bin, Code Table file: code_table.txt



Class encoder: This class contains a methods to transform input file into binary 'encoded.bin' file.

Members:

- frequencyTable: Stores <value,frequency> from input file.
- codeTable: Stores <value, huffmanCode> calculated from the frequencyTable.
- cacheOptFourAryHeap: 4-way heap used to perform PriorityQueue functions

Functions:

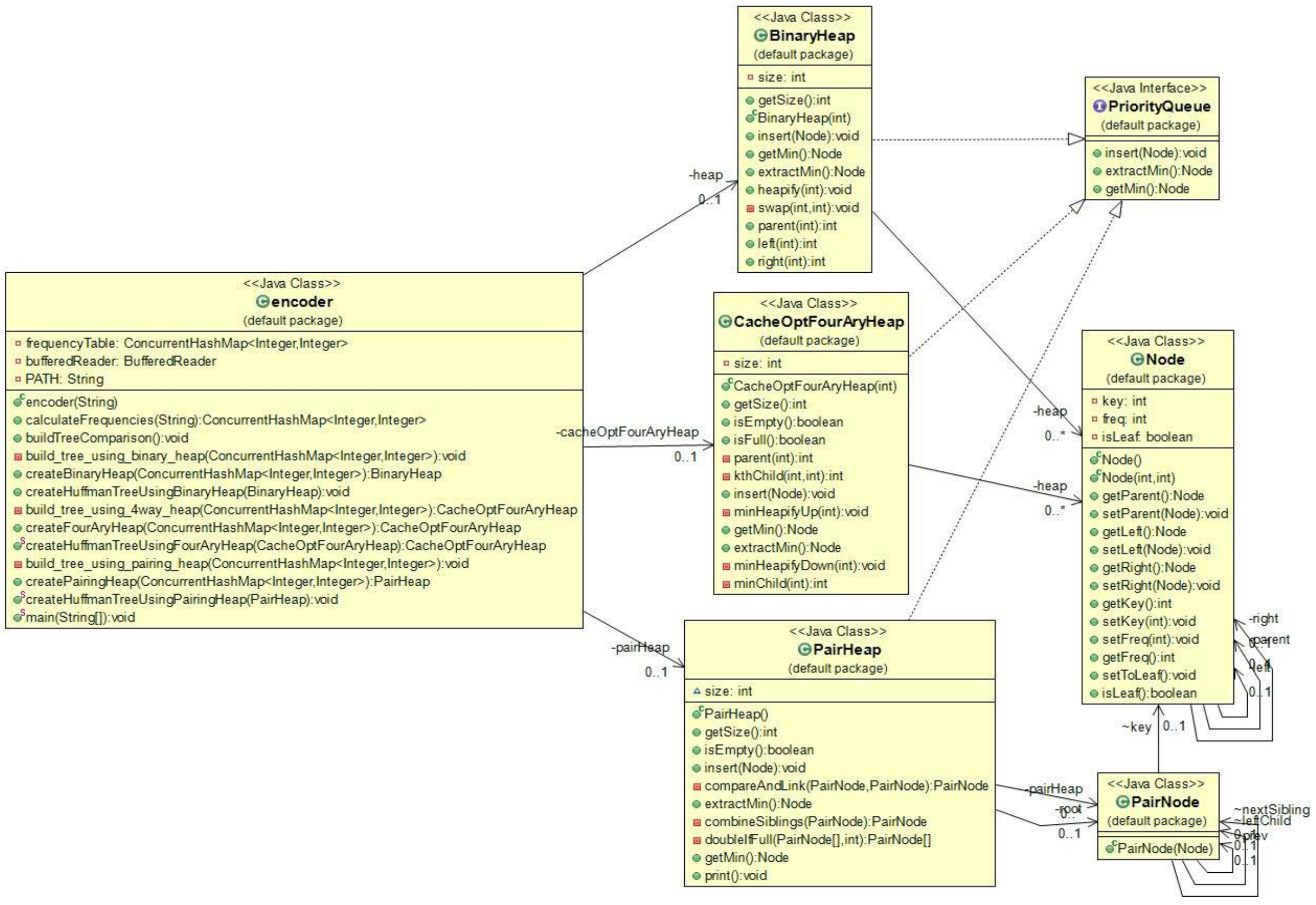
- **ConcurrentHashMap<Integer, Integer> calculateFrequency(String PATH):**
  Parameters: String Path of the input file
  Return type: ConcurrentHashMap<Integer, Integer>
  Description: Calculates the frequency of every input value and stores the <value, frequency> in a ConcurrentHashMap<Integer, Integer> data structure.

- Building the Huffman Tree:
  Build a binary Huffman tree from the using following algorithm[1]:

```
HUFFMAN($C$)
1  $n = |C|$
2  $Q = C$
3  for $i = 1$ to $n - 1$
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  return EXTRACT-MIN($Q$)      // return the root of the tree
```

  Implement Priority Queue functions(Q) functions – EXTRACT-MIN(Q) and INSERT(Q, node) by comparing the run time for the three data structures: Binary Heap, 4-Way Cache Optimized Heap and Pairing Heaps.

  Following Class Diagram represents the program for calculating the fastest data structure for implementing Priority Queue operations:



We compare time taken by:

- **BinaryHeap build_tree_using_binary_heap(ConcurrentHashMap<Integer, Integer> frequencyTable)**
- **CacheOptFourAryHeap build_tree_using_4way_heap (ConcurrentHashMap <Integer,Integer> frequencyTable)**
- **PairingHeap build_tree_using_pairing_heap(ConcurrentHashMap<Integer,Integer> frequencyTable)**

Input: ConcurrentHashMap<Integer, Integer> frequencyTable
Output: BinaryHeap/ CacheOptFourAryHeap/ Pairing Heap

Since, 4-Way Cache Optimized Heap gave best performance, it is used to build the Huffman Tree using 4-Way Cache Optimized Heap.

built_tree_using_4way_heap(): This method calls two methods:

- **CacheOptFourAryHeap createFourAryHeap(ConcurrentHashMap<Integer,Integer> frequencyTable) -**
  This method creates a Node object for each value in the ConcurrentHashMap<> frequencyTable and inserts it into a 4-way min heap.

- **CacheOptFourAryHeap createHuffmanTreeUsingFourAryHeap(CacheOptFourAryHeap cacheOptFourAryHeap) –**
  This method uses the following algorithm[1] to form the Huffman Tree:

```
HUFFMAN($C$)
1  $n = |C|$
2  $Q = C$
3  for $i = 1$ to $n - 1$
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  return EXTRACT-MIN($Q$)      // return the root of the tree
```

- **void generateHuffmanCodes(Node node, String code) -** Traverses the Huffman Tree from root to the leaf and assign a Code to each value in the leaf node based on the path. Store the <value, code> is stored in a ConcurrentHashMap<Integer, String> 'codeTable' data structure.
- **void writeFile(String PATH) –** Write file method writes the ConcurrentHashMap<Integer, String> 'codeTable' to the 'code_table.txt' file. It then calls the writebinaryText() method to write individual values in the input file as binary bits in encoded.bin file.
- **void writeBinaryText(BufferedReader br, BufferedOutputStream bos) -** Get the Huffman Code value for every value in the input file, convert the String containing of 0's and 1's to a BitSet data structure containing true(for 1) and false(for 0). Write encoded

byte stream to BufferedOutputStream using the BitSet.toByteArray() method. The encoded byte stream is stored in ‘encoded.bin’ file.

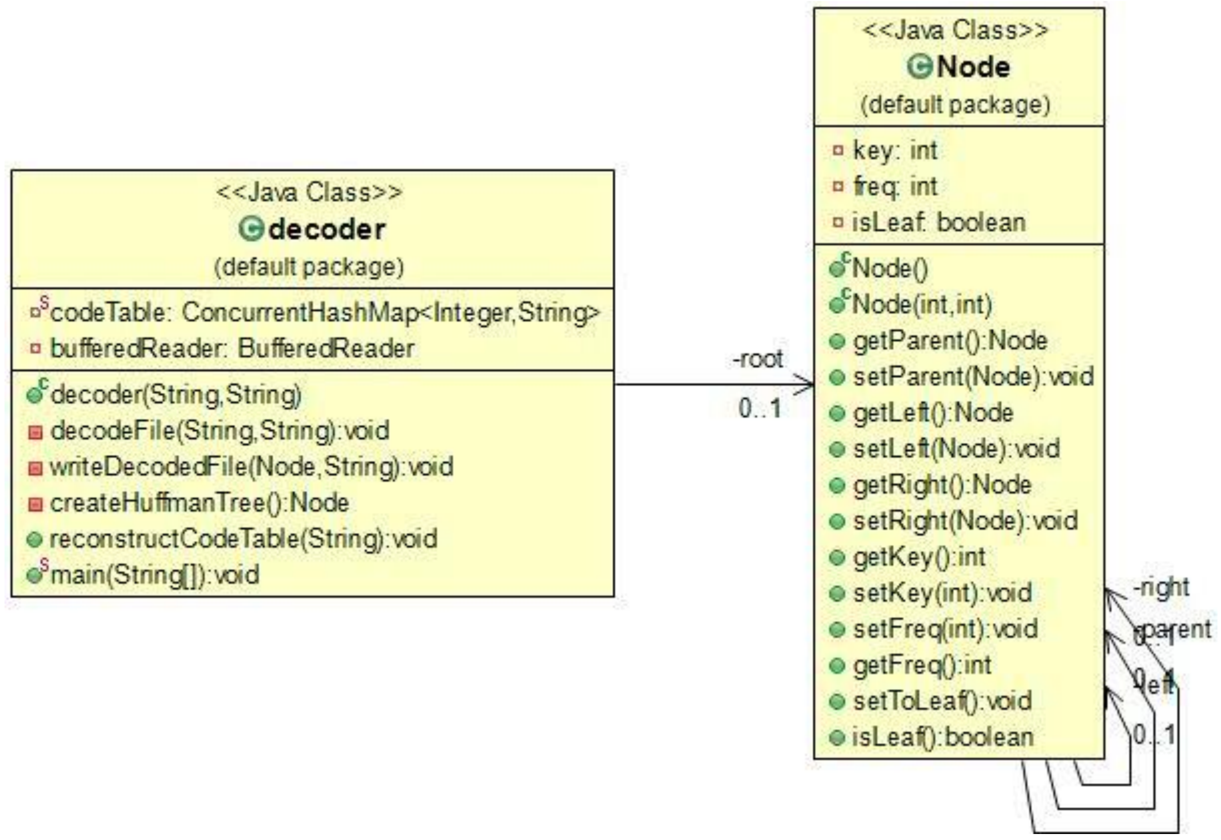
2.2 Decoding Phase:

Input:	Encoded binary file: encoded.bin, Code Table file: code_table.txt
Output:	Original Input File

Algorithm:

1. Reconstruct the code table using the code_table.txt file
2. Reconstruct the Huffman Tree using the Huffman codes in code table
3. We take each bit in the encoded.bin file and traverse through the Huffman Tree till we reach a root node. At this point the value in the leaf node is the decoded value.

Following is the Class Diagram for decoding the original input file from encoded.bin using code_table.txt



Class decoder: This class contains a methods to takes the binary ‘encoded.bin’ file and code_table.txt and generates input file.

Members:

- codeTable: Stores <value, huffmanCode> calculated from the frequencyTable.

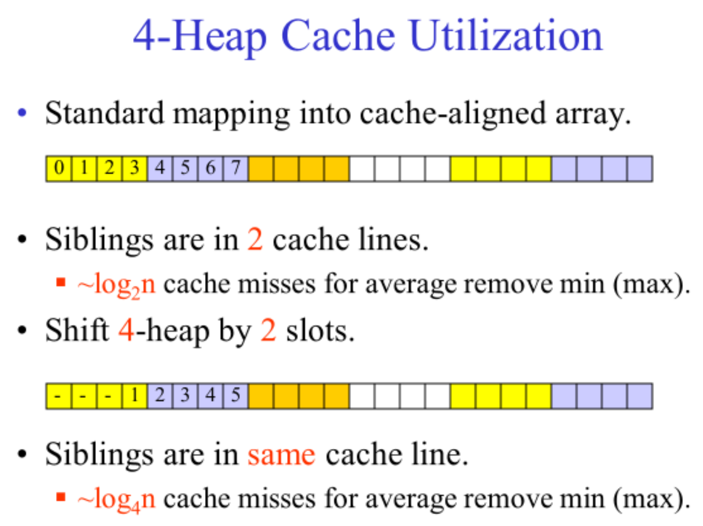
Functions:

- **void reconstructCodeTable(String CODE_TABLE_PATH):**
  Parameters: String Path of the input file
  Description: Reconstructs the ConcurrentHashMap<Integer, String> codeTable from code_table.txt
- **Node createHuffmanTree()**
  Description:
  Parses every value in the codeTable character by character.
  Every 0 is taken as a left path from the current Node and 1 is right path from current Node.
  Follow this over each character till we reach the last character in the value;
    If last character = 0, create a new left node as leaf and store the value at this Node.
    Else
    If last character = 1, create a new right node as leaf and store the value at this Node.
  Return Root Node
- **void writeDecodedFile(Node root, String ENCODED_FILE_PATH)**
  Description: We decode the String value by parsing each character through the Huffman Tree till we reach at the leaf node where we refer the actual value from the encoded value using the code table and append that value to the end of the previously decoded String.

2.3 Classes of Data Structures:

- **4-way Cached Optimized Heap:**
  Advantages of 4 way heap over binary heap:
  The height of the min heap becomes half of that of the min heap representation using Binary heap, this improves the speed of the PriorityQueue operations.
  e.g. ExtractMin operations do 4 compares per level rather than 2 (determine smallest child and see if this child is smaller than element being relocated).
  - But, number of levels is half.
  - Other operations associated with remove min are halved (move small element up, loop iterations, etc.)

  For cache optimization of the heap we insert into the heap from the 3rd index. The reason for doing this is so that children of first inserted node will be retrieved in one read of cache line.



4-Heap Cache Utilization

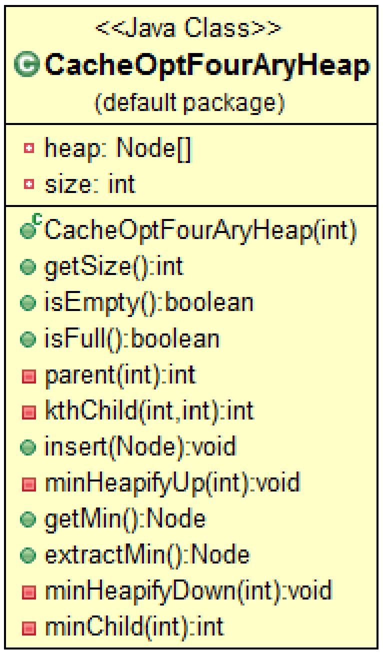
- Standard mapping into cache-aligned array.

0 1 2 3 4 5 6 7

- Siblings are in 2 cache lines.
  - ~$\log_2 n$ cache misses for average remove min (max).
- Shift 4-heap by 2 slots.

- - - 1 2 3 4 5

- Siblings are in same cache line.
  - ~$\log_4 n$ cache misses for average remove min (max).



<<Java Class>>
CacheOptFourAryHeap
(default package)

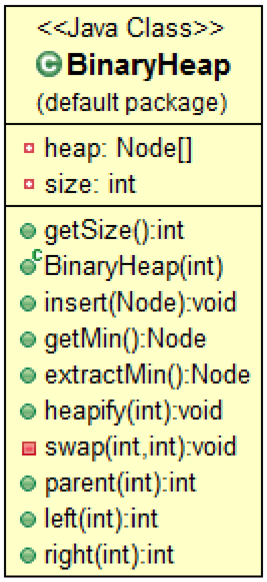
- heap: Node[]
- size: int

- CacheOptFourAryHeap(int)
- getSize():int
- isEmpty():boolean
- isFull():boolean
- parent(int):int
- kthChild(int,int):int
- insert(Node):void
- minHeapifyUp(int):void
- getMin():Node
- extractMin():Node
- minHeapifyDown(int):void
- minChild(int):int

Methods of CacheOptFourAryHeap:

- **void insert(Node element)**: Inserts the element at a leaf node and then uses the void minHeapifyUp(int childInd) method to compare the node frequency with the parent's frequency till the node reaches its correct place in the min heap.
- **int parent(int i):** returns (i/4)+2, '2' since we are inserting from the 3rd index.
- **Node extractMin():** returns the 3rd node in the min heap, replaces the 3rd node with the leaf node and calls void minHeapifyDown(int ind) to compare the 3rd node with its children and to place it at its correct position in the min heap.

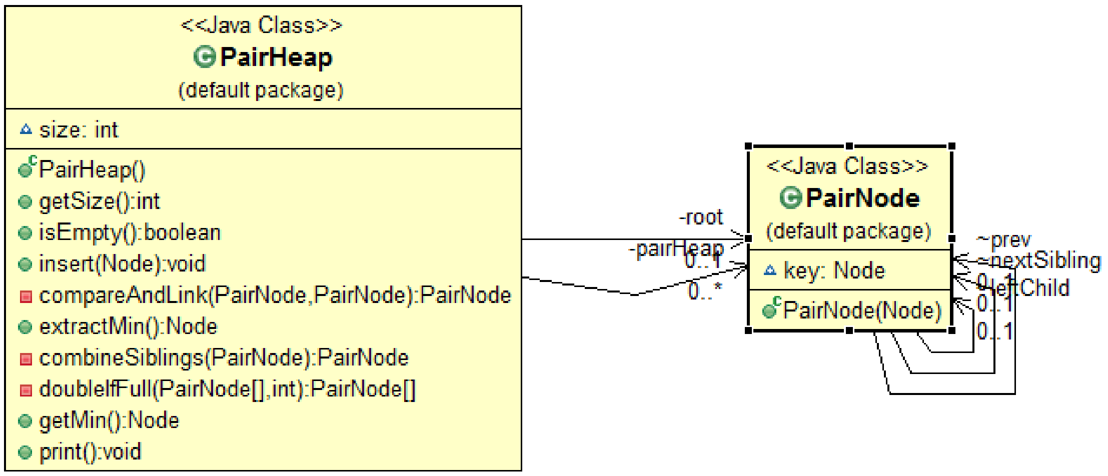
- **BinaryHeap:**



Methods of BinaryHeap:

- **void insert(Node element)**: Inserts the element at a leaf node and then compares the node frequency with the parent's frequency till the node reaches its correct place in the min heap.
- **int parent(int i):** returns (i/2) where 'i' is the child index.
- **Node extractMin():** returns the root node in the min heap, replaces the root node with the leaf node and compares the root node to its children to place it in its correct position in the min heap.

- **Pairing Heap:**

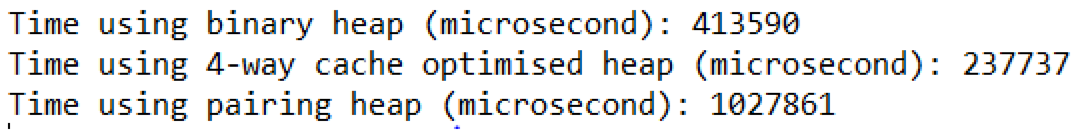


Methods of PairHeap:

- **void insert(Node element):** Creates a new PairNode and call the PairNode compareAndLink(PairNode first, PairNode second) to compare the root of the PairingHeap and the new PairNode and makes the smaller node of the two left child of the other.
- **Node extractMin():** Extract the root node and then calls PairNode combineSiblings(PairNode firstSibling) to Pairwise combine all the children of the root node to form the PairHeap

3. PERFORMANCE ANALYSIS

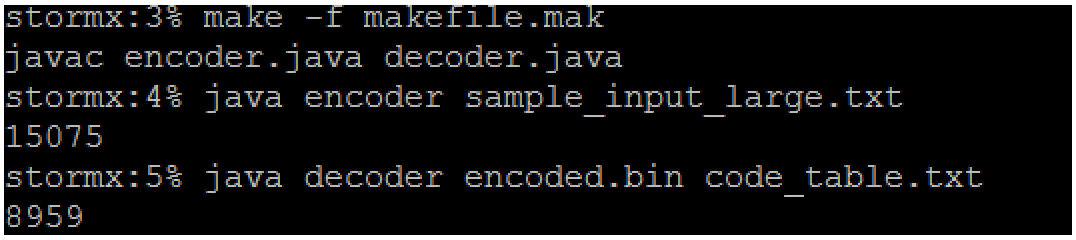
While comparing the three data structures on the storm server it was observed that 4-wayCache Optimized Pairing Heap took the least time:



```
Time using binary heap (microsecond): 413590
Time using 4-way cache optimised heap (microsecond): 237737
Time using pairing heap (microsecond): 1027861
```

The advantages of the 4-way heap is given in the section 2.3.

Encoding operation for 4-way cache optimized heap operation when tested on the Storm.cise.ufl.edu server takes approx. 15-16 secs and the decoding takes approx. 8-9secs. Following is a screenshot of the time taken in Milliseconds:



```
stormx:3% make -f makefile.mak
javac encoder.java decoder.java
stormx:4% java encoder sample_input_large.txt
15075
stormx:5% java decoder encoded.bin code_table.txt
8959
```

4. ENCODING AND DECODING ALGORITHM ANALYSIS

4.1 Encoding Algorithm Analysis: Algorithm for encoding is given in section 2.1

The total time required for Encoding includes:

1. Initializing 4-way Cached Optimized Heap: For a file of n distinct characters, we can initialize the 4-way Cache Optimized Heap in O(n).
2. Building Huffman Tree: To create the Huffman Tree the Extract-Min operation runs (n-1) times and every time takes O(log n) time.
3. Encoding values: Time required to encode a stream of m values of multiple frequencies of n distinct characters should take O(mlogn) time

4.2 Decoding Algorithm Analysis: Algorithm for decoding is given in section 2.2

The total time required for Decoding includes:

1. Building Huffman Tree: To create the Huffman Tree from Code Table containing set of n <key, value> pairs takes O(log n) time.
2. Decoding values: Time required to decode a stream of m values of represented as binary bits of max Huffman Tree height O(log n) should take O(mlogn) time

5. References

[1] Introduction to Algorithms, CLRS , 3rd Edition.