NAME: KISHAN RAO
UFID: 68568951

Objective: Build a Huffman Tree, Encoder and Decoder using the best of three priority queue structures, namely the Binary Heap, the 4-way cache optimized heap and the Pairing heap. After analysing the performance of each of the three structures, the one with the best performance was used to build the Huffman Tree. This Huffman Tree was then used to encode the input text file into a binary file having unique prefix codes for each key in the text file. This encoded binary file and the code table that was generated by the encoder was then used to build a decode tree from which the decoded input file was reconstructed.

Program Structure:
<ENCODING>

1. The Four Way Heap was used for building the Huffman Tree. The command *java encoder <input_file_path>* passes the input file with keys and frequencies to the main() method in the encoder class. This file is processed and inserted as <key,value> pairs into a HashMap.
2. The HashMap is then passed as an argument to the *buildTree(map)* function in the Four way heap. This function builds a Huffman tree from the map and returns the root of the tree.
3. The root of the Huffman Tree is passed as an argument to the *encode(PQNode node)* method in the encoder class.
4. The encoder class uses a helper function called *encodeRecursive()* to perform a DFS traversal of the tree and returns the encoded binary string for each key. This string is written to file to generate the code table for the given input file.
5. The code table is then read in the *encode()* method, and is then used to generate a StringBuilder object that is converted into a BitSet. The BitSet is then converted into a byte array and written to the binary file *“encoded.bin*”.

<DECODING>

6. The encoded.bin and code_table.txt files are then passed as command line arguments to the main method in the decoder.java class with the command *java decoder <encoded.bin> <code_table.txt>.*
7. The *"code_table.txt"* is then passed as an argument to the *decodeTree(String path)* function in the decoder class. This function reads the code table line by line and constructs a decode Huffman Tree. The root of this tree is then passed to the writeToDecodedFile(PQNode, String path) method in the decoder class.
8. The *"encoded.bin"* file generated in the encoding step is passed as the second argument to the writeToDecodedFile(PQNode, String path) method.
9. The method reads the file in bit by bit and uses it to traverse the decode tree and emits (writes) a key to the *"decoded.txt"* file when it hits a leaf.
10. Once the write is complete, the *"decoded.txt"* matches the input file and is fully reconstructed.

<DECODING ALGORITHM>

PART 1: Creating the Decode Tree

1. Create a dummy node called root. Set a temporary node to be equal to the root (this denotes the root of the Huffman Tree). current=root;
2. Read the code table in line by line.
3. For each line in the code table:
    a. Split the line on space and store the contents in a String[ ] array where array[0]=key and array[1]=Huffman Code
    b. Convert the array[1] to a Character array called code[ ]
    c. Iterate over the code array and do the following for each element:
        i. If a '1' is encountered and right child of current is null, create a right child and go to the right subtree
        ii. Else go to the right subtree

        iii. If a ‘0’ is encountered and left child of current is null, create a left child and go to the left subtree
        iv. Else go to the left subtree
4. Once you exit from the for loop you have hit a leaf so set current.key to array[0].
5. Reset current to root
6. Repeat steps 2-5 until EOF at which point, return the root of the decode tree.

PART 2: Writing to the decoded.txt

1. The root of the Decode tree and the *“encoded.bin”* file are passed to the writeToDecodedFile() method
2. Set a temp node called current to the root i.e current=root
3. Create a ByteArray whose size is the same as the encoded.bin file length
4. Use the DataInputStream to read in the entire *“encoded.bin”* file into the byte array.
5. Iterate over the byte array and for each byte do the following:
    a. Store the byte in a temp int variable.
    b. In a for loop starting at location 7 and running down to 0
        i. Do a bitwise AND of the temp int variable with 1 left shifted by “location” bits. This will extract the first bit in the byte
        ii. Right Shift the result of previous step by “location” number of bits to obtain the single bit value.
            1. If the single bit=1 move to the right by setting current=current.right
            2. Else if the single bit=0 move to the left by setting current=current.left
            3. Else check if you have hit a leaf(left and right pointers null) and so write the key of the current node to the decoded.txt file. Reset current to root i.e current=root
6. Once you exit the loop for the byte array, the file “*decoded.txt”* will contain the decoded input file.

Time complexity: *O(byte_array length * height of the Decode Tree)*

Function prototypes class wise:

1. *FourWayHeap.java*
    a. **public void** buildFourWayHeap(ArrayList<PQNode> contents){} // Initializes and creates a 4-way heap from inserted elements
    b. **private int** parent(**int** i){} // Returns parent of node at position ‘i’
    c. **private int** firstChild(**int** i){} //Returns child of node at position ‘i’
    d. **public void** heapify(**int** i){} //Performs the min-heapify operation
    e. **public void** insert(PQNode t){} //Inserts a node into the heap
    f. **public** PQNode remove{} //Removes and returns the root of the min heap
    g. **public** PQNode treeUsingFourHeap(HashMap<String,Integer> map){} //builds a Huffman Tree using the 4-way heap and <key, value> map
2. *Encoder.java*
    a. **static void** encode(PQNode root) **throws** IOException{} // builds code table and encoded.bin file
    b. **private static** HashMap<Integer, String> encodeRecursive(PQNode node, String code, HashMap<Integer, String> encoding) {} // Does a DFS traversal of the Huffman tree and returns codes for each key
    c. **public static void** reverseByteArray(**byte**[] arr){} //reverses byte array
3. *Decoder.java*
    a. **public static** PQNode decodeTree(String path) **throws** FileNotFoundException,IOException{ } // Builds decode tree

b. **public static void** writeToDecodedFile(PQNode root,String path) **throws** FileNotFoundException,IOException{} // Reads *"encoded.bin"*, traverses decode tree and writes to *"decoded.txt"*

Performance Analysis: Brief description of each type of heap with method signatures and performance analysis:

- *Binary Heap:*
    - Complete binary tree which when implemented as a min heap has every node smaller than its children at every level, with the root holding the minimum element in the heap in.

    - Time complexities for various operations:
        - Find min: $\Theta(1)$
        - Insert: O(log n)
        - Delete(Remove min): O(log n)
        - Decrease Key: O(log n)
    - Method signatures:
        - **public int** leftChild(**int** position){}
        - **public int** rightChild(**int** position){}
        - **public int** parent(**int** position){}
        - **public void** builBinaryHeap(ArrayList<PQNode> contents){}
        - **public void** insert(PQNode node)
        - **public void** heapify(**int** position)
        - **public** PQNode bremove()
        - **public** PQNode treeUsingBinaryHeap(HashMap<String,Integer> map)
        - Node class structure:

```
public class PQNode {
  PQNode left=null;
  PQNode right=null;
  int key;
  int val;
```

```
 PQNode(int key,int val){
     this.key=key;
     this.val=val;
  }

  PQNode(int val){
     this.val=val;
     this.key=-1;
  }
}
```

- Average time in milliseconds to build 10 Huffman Trees using large sample input: 1695 milliseconds

- *Pairing Heap:*
    - A pairing heap is a multiway, heap ordered structure which has a root and a list of pairing heaps. They satisfy the heap ordering property such that each pairing heap is smaller than the pairing heap node at the root. Pairing heaps rely on the meld operation wherein if the root is removed, all the children are recombined into a new pairing heap to determine the new min. This is usually implemented in a two pass or multipass scheme where heaps of equal degree are melded together to form the resulting heap.
    - Time complexities of various operations:
        - Find min: $\Theta(1)$
        - Insert: $\Theta(1)$
        - Delete(Remove min): O(log n)
        - Decrease Key: o(log n)
    - Method signatures:
        - **public** PHNode insert(**int** x){}
        - **public** PHNode pairCompare(PHNode one,PHNode two){}
        - **private** PHNode pairCombine(PHNode nextSibling){}
        - **public int** remove( ){}

        - **public void** treeUsingPairingHeap(HashMap<String,Integer> map){}
        - **public** PairingHeap(){}
        - **public boolean** isEmpty(){}
        - Node class used:

```
public class PHNode {
  PHNode leftChild;
  PHNode rightSibling;
  PHNode prev;
  int val;

 public PHNode(){}

  public PHNode(int val){
    this.val=val;
    leftChild=null;
    rightSibling=null;
    prev=null;
 }
}
```

    - Average time in milliseconds to build 10 Huffman Trees using large sample input: 890 milliseconds

- *Four way heap:*
    - The four way heap is a d-ary heap(d=4) which is a generalization of the binary heap structure where each node has 4 children instead of 2. The advantage of this is improved amortized performance on the decrease key operations at the cost of slower delete min operations. Four way heaps also have better memory cache performance compared to binary heaps.
    - Method signatures:
        - **public void** buildFourWayHeap(ArrayList<PQNode> contents){}
        - **private int** parent(**int** i){}
        - **private int** firstChild(**int** i){}

- public int size(){}
- public void heapify(int i){}
- public void insert(PQNode t){}
- public PQNode remove(){}
- public PQNode treeUsingFourHeap(HashMap<String,Integer> map){}
- Node class structure:

```
public class PQNode {
        PQNode left=null;
        PQNode right=null;
        int key;
        int val;

       PQNode(int key,int val){
              this.key=key;
              this.val=val;
               }

       PQNode(int val){
              this.val=val;
              this.key=-1;
               }
       }
```

- Average time in milliseconds to build 10 Huffman Trees using large sample input: 805 milliseconds