Programming Assignment
HUFFMAN CODING AND DECODING
Course: Advanced Data Structure
(COP 5536) Spring 2017

Submitted by: Manjary Modi
UFID :38408368
Email id: Manjary.modi@ufl.edu

Index:

1.Introduction: Huffman Encoding and Decoding

HUFFMAN Coding:
Huffman coding algorithm is lossless data compression algorithm that reduces the overall number of bits(size) of the data to be transmitted. The idea is to assign variable length code to the input character and length of code is based on frequency of the character in the input.

There are two major part of Huffman coding:
1.Build Huffman tree from input characters.
2. Traverse the Huffman tree to assign code to the characters.

Build Huffman Tree:
Huffman tree is built by creating a leaf node for each of the symbol and adding it to the priority queue, which first convert input file into frequency table. As per the requirement of the project I have used three structures for priority queue (Min Heap):
->Binary Heap
->4-way cache optimized heap
->pairing heap

Binary Heap: Binary Heap is complete binary tree that satisfies the heap ordering property (max heap and min heap). In huffman coding, min heap ordering is used to implement priority queue.

- Create leaf node of each of the unique input character with key as frequency of that character in the frequency table (Heap creation is done).
- Extract two nodes with min frequency from the heap. Create a new node whose key(frequency) is the sum of key(frequency) of these two nodes and its left and right child will be these extracted two nodes. Insert newly created node into the heap
- Repeat step 2 until only one node is remaining.

4-way cache optimized heap: 4way heap is the special case of the binary heap in which each node is having 4 children which satisfies the ordering property. Here too, min heap ordering is used to implement priority queue.

- Create leaf node of each of the unique input character with key of the node as frequency of that character in the frequency table (Heap creation is done). Create min 4way heap of these nodes.
- Extract two nodes with min frequency from the heap. Create a new node whose key(frequency) is the sum of key(frequency) of these two nodes and its left and right child will be these two nodes. Insert newly created node into the heap. This step has been to done to create the Huffman tree.

- Repeat step 2 until only one node is remaining.

Pairing Heap: Pairing Heap implementation of heap data structure having excellent amortized performance. It is a type of self-adjusting binomial heap.

- Create leaf of each of the unique input character with key at the node as frequency of that character in the frequency table (Heap creation is done).
- Now meld nodes in which Compare node with root and node (heap with root) with larger key will become leftmost subtree of the other node.
- Now extract min two nodes and sum up the key and form a new node with this key and insert the node and heapify it.

Once Huffman tree is created. I have created the Huffman code from Huffman tree (from the best performance priority queue) and written it into code_table.txt.

Encoding-In encoding, the input file is fed into the encoder which reads the input string and map it to its Huffman code (from code table) which in turn written into a encoded file in binary form**at.**

Decoding-In decoding step, first I have created a decoder tree and converted the encoded binary file into coded string, and then traverse the coded string on the decoder tree to generate decoded output.

Result: Input message should be same as output message.

2.Language and Compilation instruction:

Sr.No	Environment	Compiler	Java Version	Test Status
1	**Windows**	**Javac**	**1.8.0_121**	**Test Passed**
2.	**Unix(Thunder.cise.ufl.edu)**	**Javac**	**1.8.0_111**	**Test Passed**

Run Makefile for compilation

>> make -f makefile

Execute encoder class to encode file

>>java encoder <inputfile>

Excecute decoder class

>>java decoder <encoded file> <code_table.txt>

3.Node Structure of data structures used:

-**BinaryNode.java** define the structure of Binary Heap node which consist of its properties and getter and setter method:
1. freq: int 2. isLeaf: boolean 3. key: int 4. lefthild: BinaryNode 5. parent: BinaryNode 6. rightChild: BinaryNode

-**FourWayHeapNode.java** define the structure of Four Way Heap node which consist of its properties and getter and setter method:
1.child_four : FourWayHeapNode 2.child_one : FourWayHeapNode 3.child_three : FourWayHeapNode 4.child_two : FourWayHeapNode 5.freq : int 6.isLeaf : boolean 7.key : int 8.left_child : FourWayHeapNode 9.parent : FourWayHeapNode 10.right_child : FourWayHeapNode

-

PairHeapNode.java define the structure of Pair Heap node which consist of its properties and getter and setter method:
1. child : PairHeapNode 2. hLeftChild : PairHeapNode 3.hRightChild : PairHeapNode 4.isLeaf : boolean 5.key : int 6.nextSibling : PairHeapNode 7.prevSibling : PairHeapNode 8.value : int

Node.java: This node structure is used to form decoder tree node having below properties and its getter and setter methods:
1.code : String 2.left : Node 3.right : Node 4.value : int

4.Function prototype, structure and class diagram:

-Frequency.java:

- **read_file(String):** It returns the frequency table in the form of hash Map.
  Input- It takes the input as filepath of input file

Output-HashMap(Key,frequency)

-**BinaryMinHeap.java: Basic operation in binary heap**

- **BinaryMinHeap(int):** Initialises the BinaryMinHeap array
  Input-index of the array
- **delMin():**it deletes the minimum element (root) from the heap and copy the last element at the root and heapify at root
  Output-Binary Node
- **getHeap():** it returns the Binary heap array .
  Output -array of binary node.
- **getMin():** it returns the minmum element of the heap
  Output:BinaryHeapNode
- **heapify(int):**it min heapify the heap after deletion operation or any operation what changes the order of the heap
  Input- index of element at which heapify has to done
- **insert(BinaryNode):**it inserts the binary node in binaryNode array(Heap) and increase the size of the heap by 1
  Input- BinaryNode
- **left(int)-**it returns the index of the left child of the node
  Input-index of the node whose left child has to be returned
  Output-index of the left child
- **parent(int)-** it returns the index of the parent node
  Input-index of the node whose parent index has to be returned
  Output- index of the parent node
- **right(int)-**it returns the index of the right child of the node
  Input-index of the node whose right child has to be returned
  Output-index of the right child
- **size()-** It returns the size of the Binary Heap
  Output-Size of the binary yHeap
- **swap(int, int):**it swaps the two nodes of the binary heap
  Input: index of the two nodes

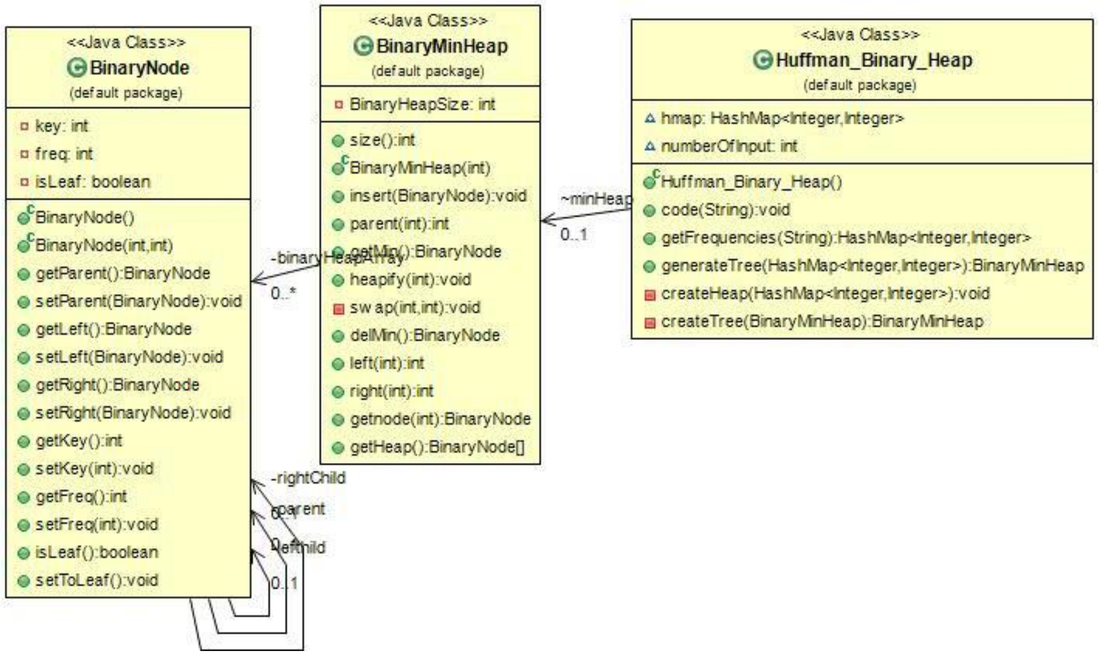
-HuffmanBinary_Heap.java: This class make the use of basic operations of binaryheap.java to generate binary min heap and Huffman tree.

- **createHeap(HashMap<Integer, Integer>):** It creates the binary min heap using frequency table
  Input:It take hashmap of frequency table with key and value as frequency
  Output:It returns the binary min heap in the form of binaryHeapNode array.

- **createTree(BinaryMinHeap):**It creates the tree by removing the two minimum from the tree, summing their frequency and form a new node , and insert it into the tree and repeat the process until only once node is remaining .
  Input: It takes binary heap (array of binary nodes)

- **generateTree(HashMap<Integer, Integer>):**It calls both create heap and create tree in order to generate the Huffman tree.

Class Diagram of binary min heap implementation



-FourWayHeap.java: Basic operations on 4-way Heap

- **FourWayHeap(int):**it initializes the four way heap
  Input-index of the node
- **delete_MinNode():** it delete the minimum node(root) and copy the last element at root and heapify it.
  Output:fourWayHeapNode
- **findMinChild(int, int);** it finds the index of smallest child of the node
  Input:start index and end index of children
  Output:Index of the smallest child
- **FourWayHeapSize():** It returns the size of the heap
  Output:size of the heap
- **getHeap():**It returns the array of four way heap node
  Outupt: FourWayHeapNode[] array
- **getMinNode():**It returns the minimum element of the fourwayHeap(root)
  Output:It returns the root of the fourway heap
- **heapify(int)**:It maintains the minheap order of the fourway heap when node is delete from the heap
  input-index at which heapify operations has to be applied
- **insertNode(FourWayHeapNode);I**nserts a new node in fourway heap
  Input- FourWayHeapNode
- **parent(int):**it returns the parent index of the fourway Heap node
  input-index of the node
  Output- index of the parent node of the input node
- **swapTwoNode(int, int):**It swaps the two node

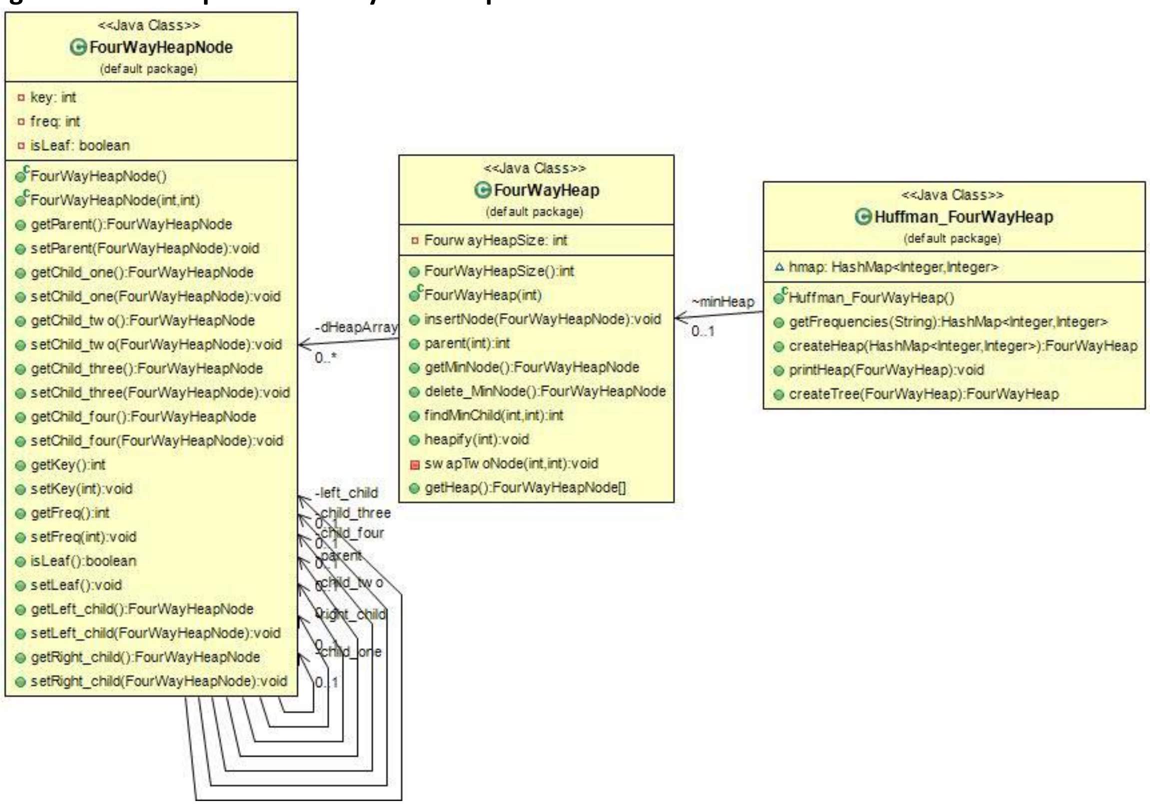
Input- indices of the nodes

Huffman_FourWayHeap.java: This class make the use of basic operations of FourWayHeap.java to generate cache optimized fourway heap and Huffman tree which is cache optimized by following the reference in lectures slides.

- **createHeap(HashMap<Integer, Integer>)** - It creates the fourway heap using frequency table
  Input:It take hashmap of frequency table with key and value as frequency
  Output:It returns the fourway min heap in the form of fourwayNode array.

- **createTree(FourWayHeap)**: It creates the tree by removing the two minimum from the tree, summing their frequency and form a new node , and insert it into the tree and repeat the process until only once node is remaining .
  Input: It takes FourWayHeap heap (array of fourway heap node)

Class diagram of cache optimized 4way min heap



- PairHeap.java-To do basic operation on pairHeap

- **PairHeap():**it initializes the pair heap
- **combineSiblings(PairHeapNode):**it stores the subtree in an array and combine subtree two at a time going left to right until all the trees are meld to one
  Input- PairHeapNode
  Output:PairHeapNode
- **compareAndConnect(PairHeapNode, PairHeapNode);**
  It connect firstchild and second child . It make the child with large value as leftmost child of the one with smaller value

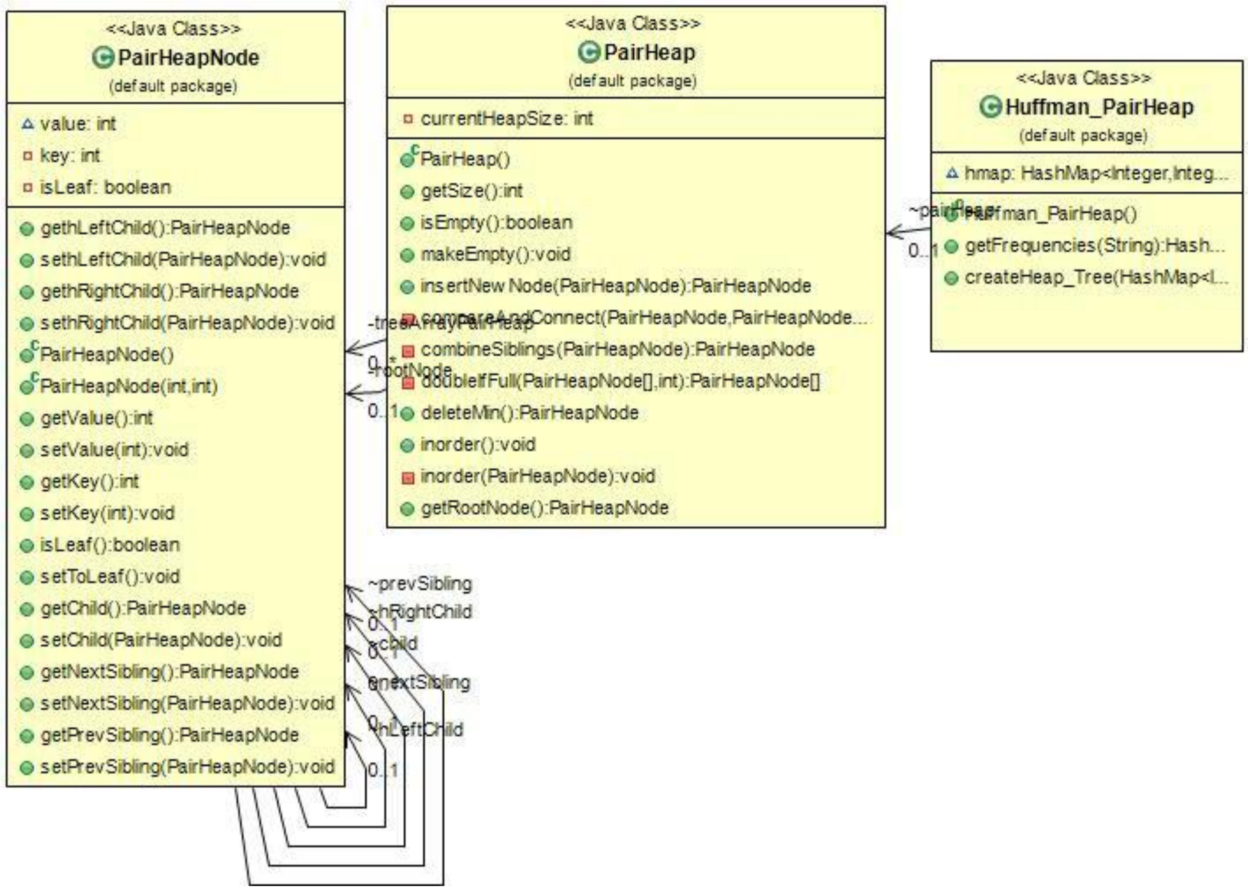
Input: Two pair Heap Node
Output :PairHeapNode(Resulting tree root) after Melding

- **deleteMin():**It delete the root the min tree and combine the subtrees to one tree again
  Output:Returns the root of heap
- **doubleIfFull(PairHeapNode[], int):**it Double the size of array if heap is full
  Output:PairHeapNode array
  Input:Old pairHeap node array and index
- **getRootNode():**It returns the root of the pair Heap
  input- PairHeap node
- **getSize():**It returns the size of the heap
  Output;integer
- **insertNewNode(PairHeapNode):** It inserts a new node into the Pair Heap
  Input: PairHeapNode
- **isEmpty():**it checks if the heap is empty or not.
  Output : boolean
- **makeEmpty():**it Make the root as null

-Huffman_PairHeap: This class make the use of basic operations of PairHeap.java to generate PairHeap heap and Huffman tree.

- **`createHeap_Tree(HashMap<Integer, Integer>):`** It creates the pair heap using frequency table and then create Huffman tree by removing the two minimum from the tree, summing their frequency and form a new node , and insert it into the tree and repeat the process until only once node is remaining .
  Input:Hashmap
  Output:PairHeap(array of PairHeap node)

Class diagram of pair Heap implementation:



Time_Calculation.java: This class calculates the running time of all the Huffman tree generation by all the heap structure as priority queue.

- **Binary_Min_Heap(String)**:It takes the input_file path as input. Run the generate tree method of Huffman binary heap class 10times and calculate the average running time for tree generation in micro seconds:
  Input:File path

- **FourWayHeap(String)**: It takes the input_file path as input. Run the generate tree(with heap creation and tree generation) method of Huffman fourway heap class 10times and calculate the average running time for tree generation in micro seconds:
  Input:File path

- **PairHeap(String):** It takes the input_file path as input. Run the generate tree along with heap creation method of Huffman pair heap class 10 times and calculate the average running time for tree generation in micro seconds:
  Input:File path
- main(String[]):It calls all the above three method and prints the average running time of all the priority queue implementation.

After the seeing the performance the cache optimized four way heap has the best running . So we will create encoder using that.

Encoder.java: This file take the input file path as input and build frequency table. Then Huffman tree (fourwayHeap)and code table,which in turn creates encoded.bin file.

- **generateCode_FourwayHeap (FourWayHeapNode, String):**It generates the Huffman code of huffman tree by tracing each node and reaching to the leaf node to generate code of that node.
  Input: FourWayHeapNode
  Return :Hashmp of the code and frequency
- **main(String[]):** it takes input file name as command line argument and calls all the methods to generate encoded.bin and code_table.txt
- **write_to_file_symbol(HashMap<Integer, String>, String):**It takes the hashmap(Huffman code) as input write it to code table file
  Input: Hashmap, String(code file path)
- **encode_data(String, String, String):**It take input file name, code filename and encoded.bin file name as input and encode the input using Huffman code in binary formate and write it into binary file encoded.bin

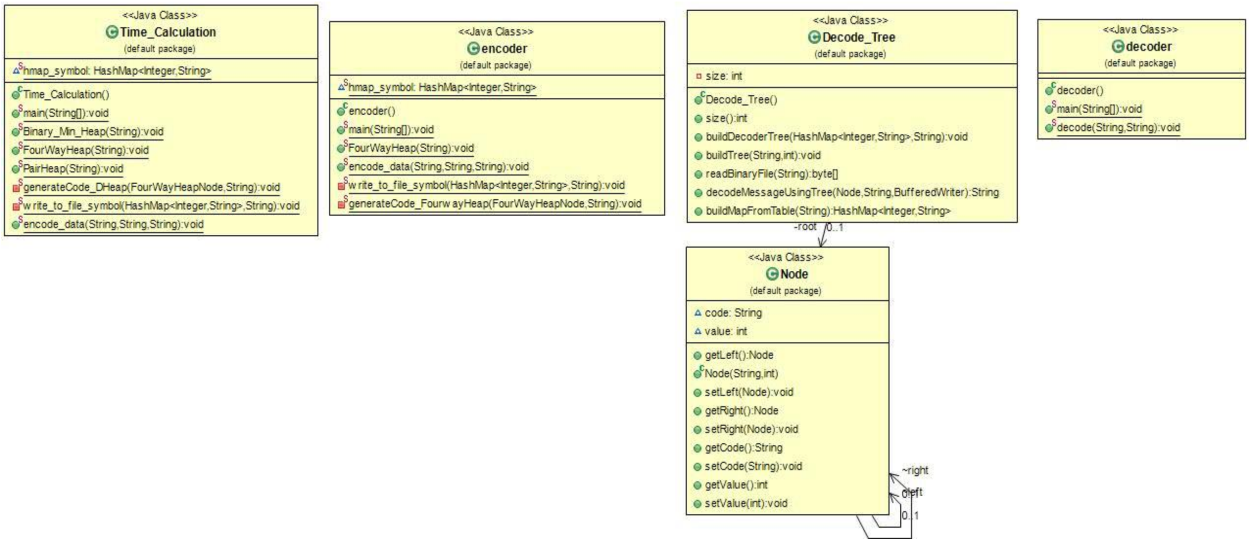
Decoder_tree.java: to perform decoding operations

- **buildMapFromTable(String):**This method take the input as code table file and returns the hashmap of code and value
- **buildTree(String, int):**This method takes code and input values and build the decoded binary tree.
- **decodeMessageUsingTree(Node, String, BufferedWriter):**This method takes the decoded tree,encoded message and buffer writer as input and traces the message over tree and once it reaches the leaf node ,it write the decoded code into decoded.txt file and merge the residual message string with next message.
- **readBinaryFile(String):**It read the binary encoded.bin file and convert it into string and return the byte array[]
  input: Binary file
  output:byte array.

Decoder.java : This class file take code table and input file as input and generate the decoded output.

- **decode(String, String)**:This method takes the input as encoded.bin and code table.txt path. This first create decoded tree using code table.txt. It reads the encoded.bin in string format after converting from binary format and then generate the decode output by traversing this encoded string in decoded tree.
- **Main method**: It takes encoded.bin and code_table.txt as input as commandline argument and call decode method to generate deoded message in file.

Encoding: In encoding process the input file will be fed to the encoder which first form the create the code_table.txt file from Huffman tree of 4way heap and then encode the input message into binary format in encoded.bin file



5.Performance of Huffman tree generation:

Average Running time of Huffman tree generation on given sample_input_large.txt file, using three different heap structure to implement priority queue over 10 iterations:

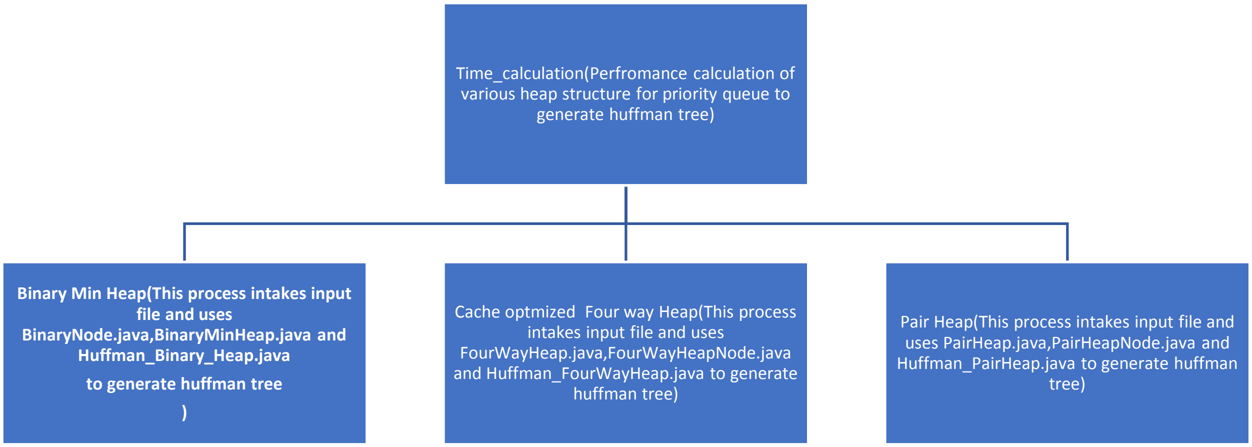
SR. No	Heap Structure	Time in microseconds
1	Binary Heap	715300
2	4-way cache optimized heap	639700
3	Pair Heap	915500

In above data structures, 4-way cached optimized heap is having the best performance. 4-way heap will be used to generate Huffman code.

Performance calculation of the various heap structure for priority queue implementation

```
Time to Generate Binary Min Heap tree --715300
Time to Generate 4way  Heap  tree --639700
Time to Generate Pair Heap  tree --915500
```

Class flow of tree generation by various heap data structure:



Cache optimized Fourway is having best performance amongst all.

Reasons of 4-Way heap is having better performance:
1. With 4-way implementation, the height of the tree is half(log_4n) as of the binary heap(log_2n) and better than pair heap too.

- At the time of insertion the node has to half as many levels as binary min heap
- At the time of remove min number of compare at each level will be four but number of level are halved.

2.The cache misses with the cache optimized 4-way heap implementation will reduce from log_2n to log_4n for remove min operation.

6.Encoding Implementation:

Once the Huffman tree is generated, out of above described three data structure, I have used the 4 way Huffman tree to generate the Huffman code. This code along with its value will be written to the code_table.txt file

Input file is fed into encoder class. Encoder class generates the code_table.txt file using Huffman code generated from Huffman tree. Now the input message will be read from input file one by one and and replaced with their code. Once the coded message is generated. This message is converted into bitwise format. And this binary format message will be written to the encoded.bin file.

7.Decoding Implementation:

Two steps of decoding
1.Build decoder tree.
2.Decode the message by traversing on decoder tree

a. Decoder Tree -

After the code is encoded in encoded.bin, as per the instruction to I have created decoder tree as Decoder tree is similar to binary Search tree, constructed by taking input of code_table. The decoder tree node will have following properties:
code : String- All the internal nodes will stores 0 or 1 values depending upon whether is right child(1) or left child(0). In case of leaf node it will have complete huffman code of each of the unique element of in code table
left : Node-It points to the left child of the node .
right : Node-It points to the right child of the node
value : int-It stores value as -1 for all the internal node and in unique element of code table in leaf nodes

Building of decoding tree: -The code will read the code table each element and its code. Then insert it into the tree traversing from the root**.**

Algorithm is as follows**:**

Build_Decoder_Tree(code, value)

```
      Node node =root
            char[] ch=code.toCharArray() //convert code to character array
         For each character in charater array
               if(ch =='0')
                     if(node.left == null)
                           node.left=new Node("0",-1);
                           node.left=node.left;
                     else
                           node=node.left;

               else if(ch=='1')
                     if(node.right == null){
                           node.right=new Node("1",-1);
                           node=node.right;

                     else
                           node=node.right;

          End of For loop
```

```
node.code=code;
node.value=value; //copy value and code from code table in leaf node
```

Complexity-

Let say n is the number of distinct elements in input.

Time to insert one element is O(logn), where logn is the height of 4-way array.

There are total of n insertion

This algorithm will take 0(nlogn).

Eg: 22 00110-

root ->traverse left (check for node and create if it isn't exists and make this as current node)

-> traverse left (check for node and create if it isn't exists and make this as current node)

-> traverse right (check for node and create if it isn't exists and make this as current node)

->traverse right (check for node and create if it isn't exists and make this as current node)

->traverse left (last character, create node with value 22 and code 001100).

b. Decode message by traversing the decoder tree

Steps

1.Read the binary file and which in turn return the array of byte.

2.Convert each byte into string.

3.Traverse the string character on decoder tree to get decoded message(decoded message will be at the leaf node which will be written into decoded.txt ,so at time some residual string will be return by this algorithm which will be appended to next byte message string).

```
decodeMessageUsingTree(DecoderTree root, message)
   while(message.length()>0){
                  for(int i=0; i<message.length();i++){
                      residual = residual + message.charAt(i);
                      if(message.charAt(i)=='0'){
                         n=root.left;
                         root=n;
                                    if(n.left==null && n.right==null ){
                                           bw.write(n.value + "\n");
                                 bw.flush();
                                 message = message.substring(i+1);
                                           root=utree;
                                           residual = "";
                                 break;
                                        }

                      }else  if(message.charAt(i)=='1'){
                         n = root.right;
                           root = n;
                                    if(n.left==null && n.right==null ){
                                           bw.write(n.value + "\n");
                                 bw.flush();
                                 message = message.substring(i+1);
                                 root=utree;
                                           residual = "";
```

```
                        break;
                                }
                    }
                }
                if(residual.trim().length()>0){
                    //System.out.println(residual);
                    break;
                }
            }
            return residual;
```

Append this residual string with message string of next byte.

Time complexity 0(nlogn)

Expected out: Input Message will be same as decoded message

8.Performance analysis of the solution:

In encoding each message traverse down to leaf node of the tree. So for encoding each message will be equal to the height of Huffman tree.Incase of 4-way heap the height of the tree is $\log_4 n$ where n is distinct element in the message. So total time for encoding n message will be **$O(n\log_4 n)$.**

In decoding first we build a decoding tree which in turn takes **$O(n\log_4 n)$.** After decoding tree, actual decoding take **$O(n\log_4 n)$.**

9.References:

1. http://www.cise.ufl.edu/~sahni/cop5536/index.html
2.Introduction to Algorithms, 3rd Edition (MIT Press) 3rd Edition by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein
3. https://www.wikipedia.org/
4. www.google.com