- ***main(String args[])***
  Main function to read the input file using args[0], call build frequency table, parse to the build_Tree function, generate a encoded string map and encoded data to print into a output file (encoded.bin)
  Input:
  args[]: command line arguments: input file.

- ***BuildFrequencyTable(String filename)***
  Function to load data from a file, initialize a frequency table and generate a Heap.
  Input:
  Filename: Filename with input file.

- ***BuildTree(Heap)***
  Build Huffman tree using Heap.
  Input:
  Heap

- ***BuildEncodedMap(Map, node, string)***
  Build a hashmap to contain <Key = String to encode, Value = Huffman code/>
  Input:
  Map: empty map that needs to be filled.
  Node: Root node of the Huffman tree./ and child node to be used for recursive DFS Huffman code.
  String: Huffman code halfway build during DFS traversal

Programming Assignment - I
Huffman Tree Implementation

Author: Ananda Kishore Sirivella | UFID: 9951-5080

<u>Language and Compiler Information:-</u>

- Java – version 1.8.0_65
- Laptop specifications - Core i5 (2.3GHz), 8GB RAM & windows 10
- Tested for input size of 1 million.
- Directory structure:
- Step to compile:
  + Javac encoder.java
  Creates the following files:
    - encoder.class
    - Node.class
    - Heap.class

- TestHeap.class
- MinHeap.class
- FourWayHeap.class
- PairingHeap.class
- PairingHeap$TreeNode.class

+ Javac decoder.java
Creates the following files:

- decoder.class

- Steps to execute:
    - Java encoder <input_Filename/>
    - Java decoder <Code_table_filename/> <encoded_filename/>
    - Input_Filename: File with initial configuration for building a Huffman tree and encoding the contents accordingly.
    - Code_Table_filename: File with values and Huffman encoded value, space separated
    - Encoded_fileName: Binary file which contains all the value Huffman encoded.

Function Prototypes and Description:-

Encoded.java:-

- ***main(String args[])***
  Main function to read the input file using args[0], call build frequency table, parse to the build_Tree function, generate a encoded string map and encoded data to print into a output file (encoded.bin)
  Input:
  args[]: command line arguments: input file.

- ***BuildFrequencyTable(String filename)***
  Function to load data from a file, initialize a frequency table and generate a Heap.
  Input:
  Filename: Filename with input file.

- ***BuildTree(Heap)***
  Build Huffman tree using Heap.

Input:
Heap

- ***BuildEncodedMap(Map, node, string)***
  Build a hashmap to contain <Key = String to encode, Value = Huffman code/>
  Input:
  Map: empty map that needs to be filled.
  Node: Root node of the Huffman tree./ and child node to be used for recursive DFS Huffman code.
  String: Huffman code halfway build during DFS traversal

<u>Node class:-</u>

- ***Node***
  Class utilized in creation of Huffman tree node with its data (data & freq value), left & right child linkage.

<u>Heap classes:-</u>

- **Heap:-**
  Abstract class to hold compulsory functions like meld(), removeMin() & insert(Node n).

- ***TestHeap:-***
  A dummy heap implemented using Priority queue to cross the accuracy of the other Heaps.
  - Meld(): Does nothing here;
  - RemoveMin(): get front of the queue.
  - Insert(node n): insert an element to the queue.
- ***MinHeap:-***
  A Binary heap implemented using arrays where 0th element is the head node.
  - Parent() : get parent Node;
  - leftChild(): get leftchild node;
  - rightChild(): get rightchild node;
  - Meld(): Does nothing here;
  - RemoveMin(): get Head of the heap and replace with the last element of the heap then => pushdown to heapify.
  - PushDown(int): to push down the root to its correct valid place.
  - Insert(node n): insert an elements in the end of the heap and => shiftUp(int)
  - shiftUp(int): to shift up the last element to the its correct valid place.

- ***MinHeap:-***
  A 4-ary heap implemented using arrays where 0th element is the head node.
  - Parent() : get parent Node;
  - Child(): get child node based on the index.
  - Meld(): Does nothing here;

- RemoveMin(): get Head of the heap and replace with the last element of the heap then => pushdown to heapify.
- PushDown(int): to push down the root to its correct valid place.
- Insert(node n): insert an elements in the end of the heap and => recursively comparing with the parent to check and swap if necessary.

- ***PairingHeap:-***

  A Pairing heap implemented using TreeNode object.
  - TreeNode: Node class to hold

    + Data: Node

    + children: List of TreeNodes
  - Meld(): To heapify the Heap
  - RemoveMin(): get Head of the heap and executed meld.
  - Insert(node n): insert an elements in the end of the heap and executed meld.

Encoded.java:-

- ***main(String args[])***

  Main function to read the encoded file & code table text, build an encoded string map and decode the data to print into an output file (decode.txt)

  Input:

  args[0]: input file . generally *encoded.bin*

  args[1]: code table file. Generally *code_table.txt*

- ***constructDecodetree(String)***

  Function to code_table data from a file and building a Huffman tree.

  Input:

  Filename: Filename with input file. Generally *code_table.txt*

Preformance analysis:-

Output(for 1 million records)

Heap	**Huffman tree generation Time**	**Encoded.bin generation time**
TestHeap (using Java's Priority Queue)	9.43 seconds	23.8 seconds
BinaryHeap	10.146 seconds	24.567 seconds
4-ary Heap	10.665 seconds	25.568 seconds
Pairing Heap	40 seconds	57.209 seconds

Decoding part is same for all the types of heaps.

Decode time for generating the tree: **3.441 seconds**

Decode time to decode whole file: **34.527 seconds**

+ Encoding Algorithm:

- Using DFS to reach each node and generating a hashmap with key as data and value as Huffman code
- O(1): simple HashMap look-up

+ Decoding Algorithm:

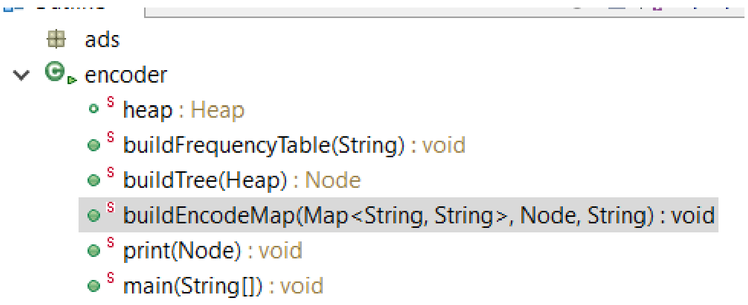
1. Read a file to byteArray
2. Read a byte by byte, start from root => read till a leaf is reached **&&** print to decode.txt file.
3. Reset the node to root and redo the whole till the end of byte array is reached.

- Complexity of O(k * log n) where k is number of value to decode and n is the number nodes in the Huffman tree.

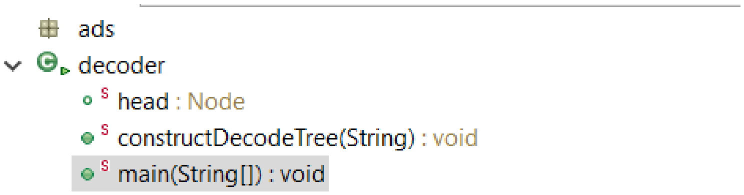
<u>Observations/ Analysis:-</u>

- Theoritically the 4-ary heap is expected to run at better complexity than 2-ary Heap (binary).
- But Practically the 4-ary heap and 2-ary seem to work really close.
- The array implementation seems to compete with Java’s Priority Queue well.

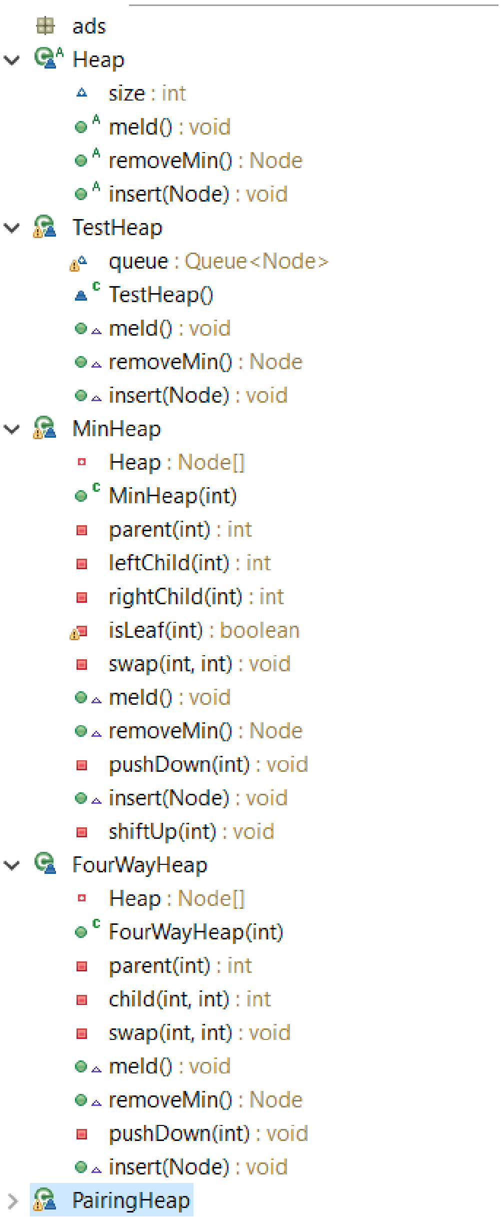
Class diagram:-



```
ads
encoder
    heap : Heap
    buildFrequencyTable(String) : void
    buildTree(Heap) : Node
    buildEncodeMap(Map<String, String>, Node, String) : void
    print(Node) : void
    main(String[]) : void
```



```
ads
decoder
    head : Node
    constructDecodeTree(String) : void
    main(String[]) : void
```



```
ads
Heap
    size : int
    meld() : void
    removeMin() : Node
    insert(Node) : void
TestHeap
    queue : Queue<Node>
    TestHeap()
    meld() : void
    removeMin() : Node
    insert(Node) : void
MinHeap
    Heap : Node[]
    MinHeap(int)
    parent(int) : int
    leftChild(int) : int
    rightChild(int) : int
    isLeaf(int) : boolean
    swap(int, int) : void
    meld() : void
    removeMin() : Node
    pushDown(int) : void
    insert(Node) : void
    shiftUp(int) : void
FourWayHeap
    Heap : Node[]
    FourWayHeap(int)
    parent(int) : int
    child(int, int) : int
    swap(int, int) : void
    meld() : void
    removeMin() : Node
    pushDown(int) : void
    insert(Node) : void
PairingHeap
```

References:-

- Introduction to Algorithms, 3rd Edition (MIT Press) 3rd Edition by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein
- Google.com
- Wikipedia.org
- http://www.cise.ufl.edu/~sahni/cop5536/