FUNCTION PROTOTYPES AND STRUCTURE:

Project consists of 9 files:

1) MinHeap.java: Interface for min heaps. Interface has type parameter E, a comparable generic class
   a) boolean isEmpty(): check whether the heap is empty
   b) E removeMin(): Extract the minimum element from the heap
   c) int getSize(): Returns number of elements in the min heap
   d) void insert(E elem): Insert element to min heap
2) BinaryMinHeap.java: consist of implementation of binary min heap. Stores comparable objects in an array.
   a) Object[] arr: stores nodes of the binary heap
   b) int size: number of elements in the heap
   c) void bottomUpHeapify(int index): compares child at 'index' with its parent and if it is smaller then, it swaps the entries and recursively called for the parent's index
   d) void topDownHeapify(int index): compares parent at 'index' with child having minimum value and if it is greater, then it swaps the entries and recursively called for the corresponding child's index
   e) E removeMin(): returns first element in the array and swaps it with last element and calls topDownHeapify at index 0. Decreases size value.
   f) void insert(E elem): inserts element at the end of the array and calls bottomUpHeapify at the last index. Increments size value.
3) FourWayMinHeap.java: consist of implementation of 4-way cache optimized min heap.
   a) Object[] arr: stores nodes of the heap
   b) int size: number of elements in the heap
   c) static final int d: This specifies the degree of the heap. This value is set to 4.
   d) int getLeftChildIndex(int index): returns index of the left most child
   e) int getParentIndex(int index): returns index of the parent of the child at *index*.
   f) E getParent(int index): returns parent object of child at *index*.
   g) void bottomUpHeapify(int index): compares child at *index* with its parent and if it is smaller then, it swaps the entries and recursively called for the parent's index
   h) void topDownHeapify(int index): compares parent at *index* with child having minimum value and if it is greater, then it swaps the entries and recursively called for the corresponding child's index
   i) E removeMin(): returns first element in the array and swaps it with last element and calls topDownHeapify at index 0. Decreases size value.
   j) void insert(E elem): inserts element at the end of the array and calls bottomUpHeapify at the last index. Increments size value.
4) PairingHeapNode.java: class for pairing heap nodes. This class is defined for generic type E
   a) E data: value of the node of type E.
   b) PairingHeapNode<E> child: child of the node
   c) PairingHeapNode<E> left: left sibling of the node
   d) PairingHeapNode<E> right: right sibling of the node
5) PairingMinHeap.java: consist of implementation of pairing min heap.

    a) PairingHeapNode<E> root: Root of the pairing heap tree. Contains the minimum value.
    b) int size: Size of the heap structure.
    c) PairingHeapNode<E> meld(PairingHeapNode<E> node1, PairingHeapNode<E> node2): Performs meld operation on pairing heap nodes *node1* and *node2* and returns the root of the resulting tree.
    d) PairingHeapNode<E> twoPassMerge(PairingHeapNode<E> child): Performs two pass merging leftmost child *child* and its siblings. Returns the root of the resulting tree.
    e) E removeMin(): returns the value of the root element. Calls twoPassMerge on its child and sets the root to the return value of the twoPassMerge. Decrements size variable.
    f) void insert(E elem): creates new pairing heap node with value 'elem' and calls meld with root node. Increments size variable.

6) Node.Java: class for Huffman tree nodes
    a) String value: stores value of the node if it is a leaf node. Otherwise, this will by empty
    b) int frequency: stores frequency of the value if it is a leaf node. Otherwise, this value will be equal to the sum of the frequencies of the children nodes.
    c) PairingHeapNode<E> left: left sibling of the node
    d) PairingHeapNode<E> right: right sibling of the node
    e) Node(Node left, Node right): Constructor sets left and right children provided in the argument. Frequency is set as the sum of the sum of the frequencies of the children nodes.
    f) int compareTo(Node node): Overrides default method for Comparable<Node> class. Returns negative value if current node has less frequency than 'node', return positive value if greater and return 0 when has equal frequencies.

7) PerformanceCheck.java: code for checking performance of different heap implementations. Generates HashMap *frequencyMap* by iterating through the input file. Map will have a String entry as the key and Integer value corresponding to number of occurrences. Then builds Huffman tree using methods described below and prints time taken.
    a) Node buildHuffmanTree(Map<String, Integer> frequencyMap, MinHeap<Node> minHeap): Takes *frequencyMap* and *minHeap* (an object of class implementing MinHeap interface) as input arguments. Converts each key-value pair in frequency map into a Node object and insert into *minHeap*. Then takes two minimum nodes from the heap and constructs a new Node parent object with 2 min nodes as children and inserts into the *minHeap*. This process is repeated until only one node remains in *minHeap*. Returns the remain node in the *minHeap*.
    b) Node buildTreeUsingBinaryHeap(Map<String, Integer> frequencyMap): Creates a BinaryMinHeap object with the size of *frequencyMap* as capacity. Calls and returns buildHuffmanTree using *frequencyMap* provided in the argument and newly created BinaryMinHeap instance as arguments.
    c) Node buildTreeUsing4WayHeap (Map<String, Integer> frequencyMap): Creates a FourWayMinHeap object with the size of frequencyMap as capacity. Calls and returns buildHuffmanTree using *frequencyMap* provided in the argument and newly created FourWayMinHeap instance as arguments.
    d) Node buildTreeUsingPairingHeap (Map<String, Integer> frequencyMap): Creates a PairingMinHeap object. Calls and returns buildHuffmanTree using *frequencyMap*

provided in the argument and newly created PairingMinHeap instance as arguments.

8) encoder.java: code for Huffman encoding and writing outputs to files. Generates HashMap *frequencyMap* by iterating through the input file. Map will have a String entry as the key and Integer value corresponding to number of occurrences. Then builds Huffman tree using methods described below.
   a) Node buildHuffmanTree(Map<String, Integer> frequencyMap, MinHeap<Node> minHeap): Takes frequencyMap and *minHeap* (an object of class implementing MinHeap interface) as input arguments. Converts each key-value pair in frequency map into a Node object and insert into *minHeap*. Then takes two minimum nodes from the heap and constructs a new Node parent object with 2 min nodes as children and inserts into the *minHeap*. This process is repeated until only one node remains in *minHeap*. Returns the remain node in the *minHeap*.
   b) Node buildTreeUsingPairingHeap (Map<String, Integer> frequencyMap): Creates a PairingMinHeap object. Calls and returns buildHuffmanTree using frequencyMap provided in the argument and newly created PairingMinHeap instance as arguments.
   c) buildCodeTable(Node root, Map<String, String> codeTableMap, String code): This is a recursive method takes root of the Huffman Tree as the argument and iterates through all of its children filling *codeTableMap*. When node traverses to left child String "0" will be appended to the 'code' and "1" will be appended for right child traversal. Whenever the method reaches a leaf node, the leaf node's value and 'code' will put as a key-value pair in the *codeTableMap*. When the buildCodeTable method returns *codeTableMap* will contain all entry to code mappings for all the leaf nodes in the Huffman tree of *root*.
   d) void writeCodeTableToFile(Map<String, String> codeTable): Writes the mappings in the HashMap *codeTable* to a file
   e) void writeToBinaryFile(StringBuilder sb): This method takes a StringBuilder object *sb* containing encoded strings as input. BitSet utility provided by java.util package is used for building *byteArray* from the StringBuilder object. We iterate through all characters in the 'sb' and set the corresponding indices in the BitSet object if the character is '1'. BitSet object Is converted into a *byteArray*. If the last bit of the BitSet object was '0', then we are putting one more redundant true bit in the end and removing that from the byte array, in order to obtain a correct *byteArray* instance. The *byteArray* is written to an output file using java.io.FileOutputStream.

9) decoder.java: This class builds Huffman tree from the code-table file, decodes the encoded file and writes to the out file using following methods.
   a) Node buildHuffmanTree(String codeTableFile): Creates an empty Huffman tree root node. Reads the input *codeTableFile* and for each line (corresponding to a value and code pair) in the file does the following operation: Build tree corresponding to the code by traversing to the left child (if it is empty creates a new child) if the character is 0 and similarly to the right child otherwise. Set the value of the resulting leaf node as the value in the line.
   b) byte[] readEncodedFile(String encodedFile): Reads the encoded binary file and returns the corresponding bye array values by using java.io.InputStream.

c) void decodeAndWriteToFile(byte[] byteArray, Node root): Takes the byte array in the encoded file and root node of the Huffman tree as input arguments. We use BitSet instance of the byte array and iterate though each bit one by one. We start from the root and if the bit is 'set' we go to the right child otherwise traverse to the left child. Whenever we reach a leaf node, we write the value of the node to the output file and go back to the root node.

PERFORMANCE MEASUREMENT:

Following measurements are taken using the *PerformanceCheck* class mentioned above. We build Huffman tree using 3 different heap implementations and the time taken by each is measured. Following experiment is conducted in thunder.cise.ufl.edu server.

Time taken in milliseconds for building Huffman Tree using different min heaps:

Binary Heap	**4-way cache optimized heap**	**Pairing Heap**
1254	973	3449
1545	1405	2859
1479	1149	2537
1482	1171	2683
1451	1060	3661
1410	1119	3940
1354	1027	1904
1343	1079	1936
1610	1166	3001
1456	1217	3253

Average time taken in **milliseconds**:

Binary Heap	**4-way cache optimized heap**	**Pairing Heap**
1438	1136	2922

As we can see 4-way cache optimized heap has the best performance followed by the binary heap. Pairing heap has shown the least performance. The reason behind this performance difference is the cache misses. In binary heap and 4-way heap, I have used array for storing the values. So, when I access the children of a node, in binary heap 2 nodes in 4-way heap 4 nodes are going to be available in the cache. So, the average cache miss for the removeMin operation in binary heap and 4-way heap are going to be ~ $\log_2 n$ and ~$\log_4 n$ respectively. In pairing heap memory addresses are not assigned sequentially and this leads to more cache misses than even binary heap.

DECODE ALGORITHM:

- Initially we begin the algorithm at the Root node of the Huffman tree and then for each bit in the encoded input we traverse through the tree.
- If we come across a 'false' bit we take the left child, otherwise we will go the right child.
- Eventually we will end up at the leaf node which will contain the decoded value for the corresponding 'bit path'.
- We write this value to the output file. We will begin traversing again at the Root node

This decode algorithm goes through each bit in the encoded file one time. So, **the complexity is O(m) where m is the number of bits in the encoded file**. We can also express this complexity as O(n*h) where n is the number of decoded values and h is the length of the maximum bit path. But the former expression is a tighter bound than the latter.

The decoder also has a step for building the Huffman tree from the code-table file (implementation detail is explained in the above section). It has a time complexity of O(k) where k is the total number of bits in all the codes. And the reading the input file into the byte array takes O(m) time complexity where m is the number of bits in the encoded file.