

Advanced Data Structures

Programming Project Report

Spring 2017

Jeni Joe

UFID: 4995-9419

jjoe@ufl.edu

Solution Description:

The problem statement was to perform Huffman encoding and decoding using one of the three priority queue data structures from pairing heap, 4 way cache optimized heap and binary heap after checking which one of them takes the least time for heap creation. First a frequency table is created consisting of the unique values with their respective frequencies. This data is used for implementing all the three data structures and the runtime for heap creation is measured. Since the least time was taken by the 4 way cache optimized heap, it was used for providing minimum values for creating the Huffman tree for encoding.

Program Structure with Function Prototypes:

Huffman Encoding:

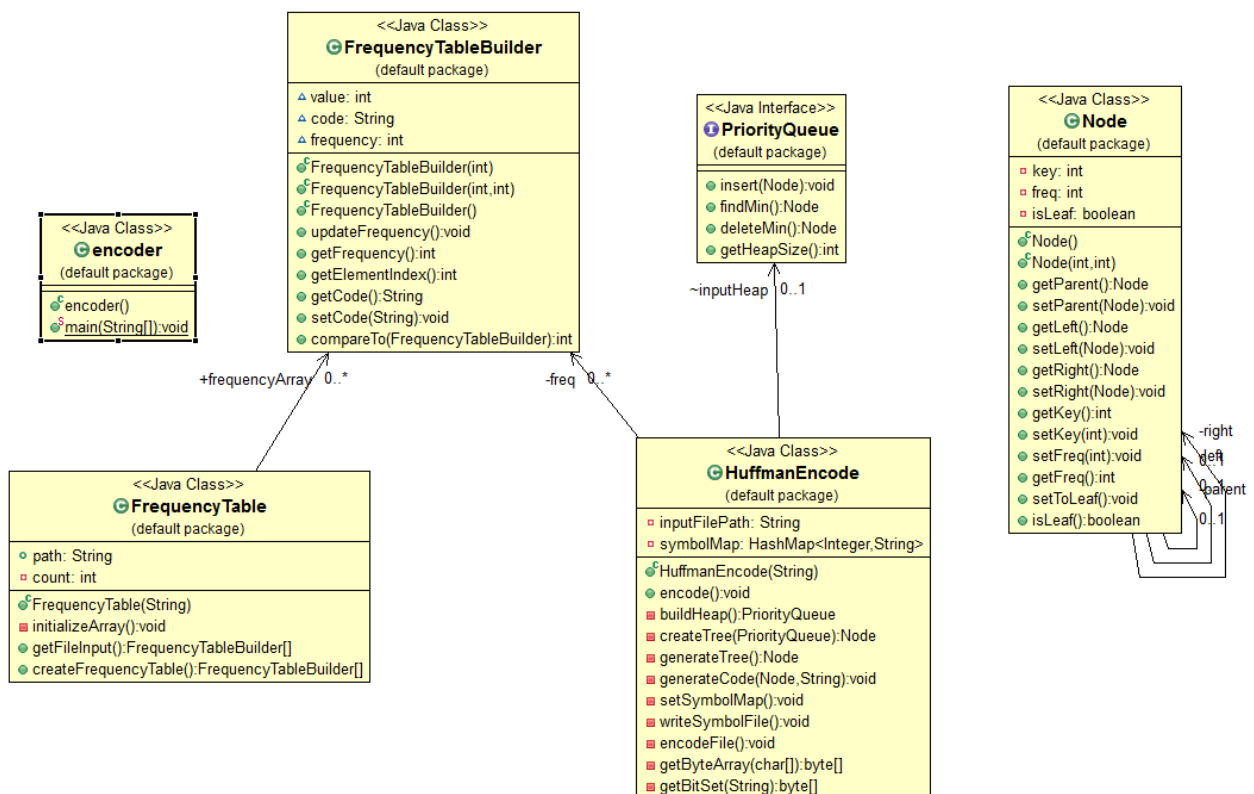


Fig 1: Class diagram of Huffman Encoding

public class encoder.java :

Takes input of the file to be encoded and instantiates an object of HuffmanEncode class and uses it to invoke encode().

public class HuffmanEncode.java :

Class that contains the methods that perform the Huffman Encoding.

private String inputFilePath

private FrequencyTableBuilder[] freq

private HashMap<Integer, String> symbolMap

PriorityQueue inputHeap;

public void encode(): Method that invokes the other methods of HuffmanEncode.java driving the encoding process. Instantiates frequencyTable, an object of FrequencyTable.java which is used for creating freq, an array of FrequencyTableBuilder.java

private PriorityQueue buildHeap(): Builds the minimum heap using the chosen priority queue data structure. Instantiates each Node object with the value and corresponding frequency and inserts the created Node objects into the heap. Returns the created minimum heap.

private Node generateTree(): Invokes createTree with the heap generated by the buildHeap method.

private void generateCode(Node node, String code): Generates symbolMap which is the code table.

private void encodeFile(): Generates the encoded.bin file by using the input text file and the symbolMap

private void writeSymbolFile(): Writes the code_table.txt file using the symbolMap

public class FrequencyTable.java:

public frequencyArray: Array of type FrequencyTableBuilder

public String path: Path of input file to be encoded.

private int count

public FrequencyTable(String path): Constructor that creates frequencyArray and invokes initializeArray().

private void initializeArray(): Invokes the FrequencyTableBuilder constructor on each of the elements in frequencyArray.

public FrequencyTableBuilder[] createFrequencyTable(): Invokes getFileInput() to populate a FrequencyTableBuilder array. Sorts the array and sets the FrequencyTableBuilder objects with the array values.

public FrequencyTableBuilder[] getFileInput(): Parses the input text file to create and return the FrequencyTableBuilder array with values and their corresponding frequency of occurrence in the textfile.

public class FrequencyTableBuilder.java:

int value

String code

int frequency

public FrequencyTableBuilder(int value): Sets the value field of the object to the integer argument passed and sets the corresponding frequency to 0.

public void updateFrequency(): Increments frequency by 1

public int getFrequency(): Return the frequency

public int getElementIndex(): Return the element's value

public String getCode(): Returns the element's code

public void setCode(String code): Sets the code field with the String argument passed.

public int compareTo(FrequencyTableBuilder o): Comparator method for the class.

public interface PriorityQueue : Implemented by the three data structures

public void insert(Node x);

public Node findMin();

public Node deleteMin();

public int getHeapSize();

public class Node:

private Node parent

private Node left

private Node right

private int key = -1

```
private int freq = -1
```

```
private boolean isLeaf = false
```

Getter and setter methods for fields.

4 way Cache Optimised Heap:

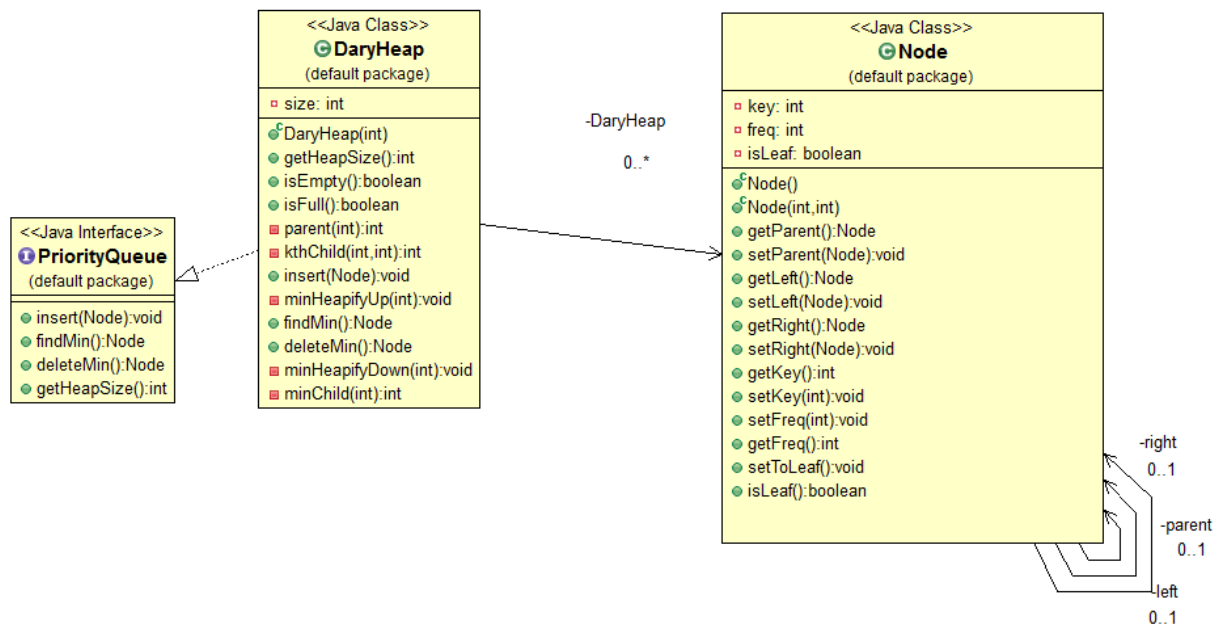


Fig 2: 4way Cache Optimized Heap (d-ary)

```
public class DaryHeap implements PriorityQueue
```

```
private Node[] DaryHeap;
```

```
private int size = 0
```

```
public void insert(Node element)
```

Insert the Node as a leaf. Invoke minHeapifyUp.

```
private void minHeapifyUp(int childInd)
```

Preserves the minimum heap condition in the heap that the parent node will always have a value lesser than or equal to its child nodes

```
public Node findMin()
```

Returns the minimum element, that is the root.

public int getHeapSize()

Returns the heap size

public Node deleteMin()

Removes and returns the root. Invokes minHeapifyDown

private void minHeapifyDown(int ind)

Sets the last leaf node at root. Finds the smallest child of the node. This node is swapped with its smallest child if needed. This, except the setting last leaf node at root is done until no more swaps are needed.

Binary Heap:

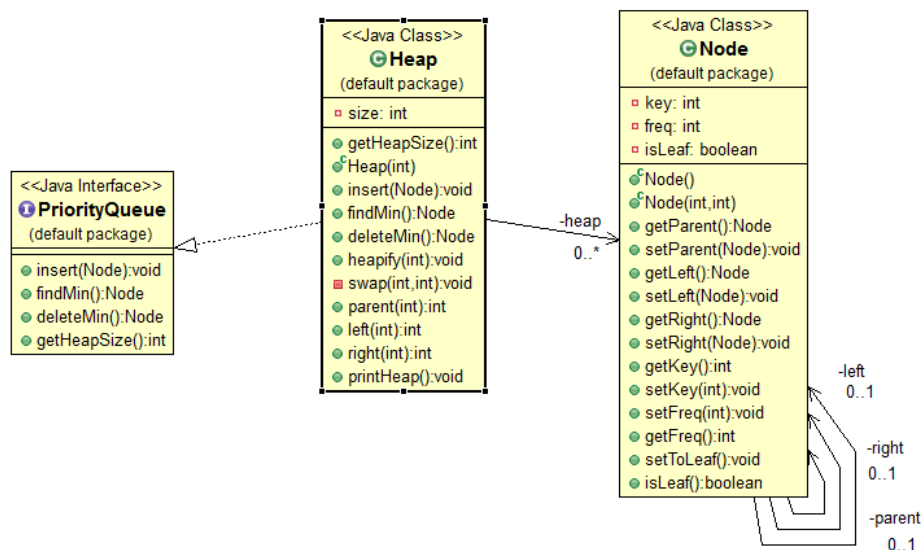


Fig 3: Binary Heap

public class Heap implements PriorityQueue

private Node[] heap

private int size = 0

public int getHeapSize(): Returns size

public Heap(int i) : Sets heap with Node of value passed

public void insert(Node k): Inserts a Node with value k into the heap while preserving the minimum binary heap property of keeping the root node's values always lesser than or equal to the child nodes.

public Node findMin(): Returns the Node with minimum value, that is stored at heap[1].

public Node deleteMin(): Deletes and returns the root and then calls heapify method.

public void heapify(int i): Maintains the Minimum Heap Property $A[\text{PARENT}(i)] \leq A[i]$

private void swap(int i, int l): Swaps the nodes with values passed.

public int parent(int i): Returns the parent value

public int left(int i): Returns the value of the left child of the Node whose value is given as argument.

public int right(int i): Returns the value of the right child of the Node whose value is given as argument

public void printHeap(): Prints heap if required

Pairing Heap:

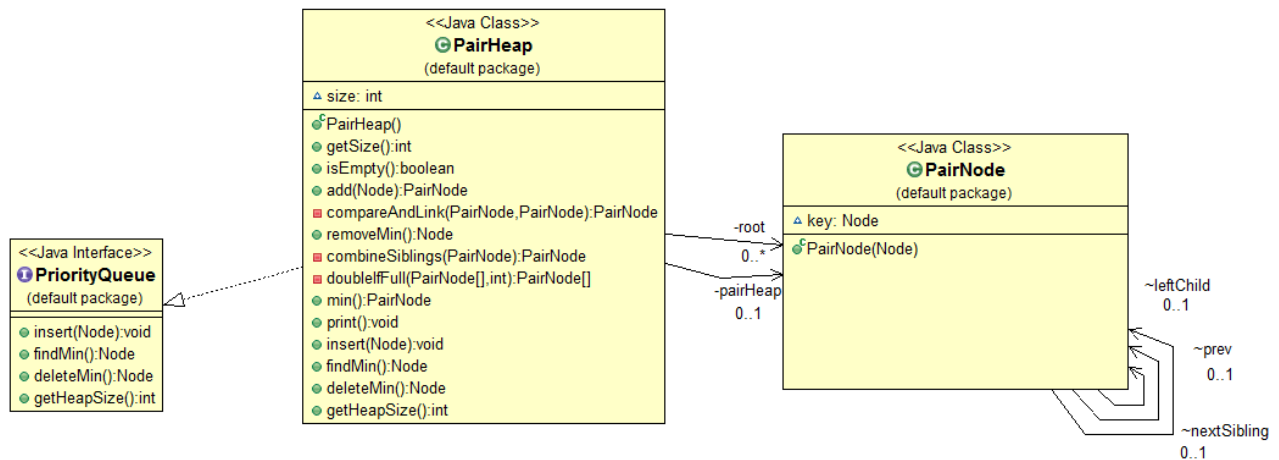


Fig 4: Pairing Heap

class PairNode

Node key

PairNode leftChild

PairNode nextSibling

PairNode prev

public PairNode(Node x) : Sets key to x and leftChild, nextSibling and prev to null

public class PairHeap implements PriorityQueue

private PairNode root

```
private PairNode[] pairHeap
int size
```

public PairHeap(): Sets the root to null

public void insert(Node x): Create a pairing heap with x and meld it with the existing pairing heap

public Node findMin(): Returns the node value of root of the heap

public Node deleteMin(): Removes the root of the heap. Melds the min trees that are formed when the root is removed until only one tree remains.

public int getHeapSize(): Returns the heap size

Huffman Decoding:

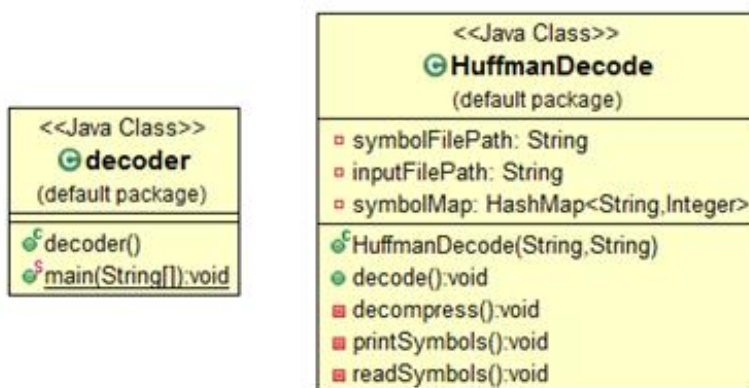


Fig 5: Huffman Decoding

public class decoder.java :

Takes input of the file to be encoded and instantiates an object of HuffmanDecode class and uses it to invoke decode().

public class HuffmanDecode.java :

private String symbolFilePath: Path for the code_table

private String inputFilePath: Path for encoded.bin

private HashMap<String, Integer> symbolMap: Stores the code_table

public HuffmanDecode(String inputFilePath, String symbolFilePath): Sets the inputFilePath and symbolFilePath.

public void decode(): Invokes readSymbols() and decompress()

private void readSymbols(): Reads the symbolFilePath (code_table) line by line and populates the Hashmap symbolMap with the values and the corresponding String Huffman code.

private void decompress(): Reads the encoded file input. Uses the symbolMap created by readSymbols and the encoded file to generate the decoded output file.

Performance Analysis Results and Explanation:

Results:

Performance analysis was done by running the createTree method for creating the Huffman tree 10 times with values provided by the minimum heap and taking average of the time.

This was performed for all the three data structures, namely pairing heap, 4 way cache optimized heap and binary heap.

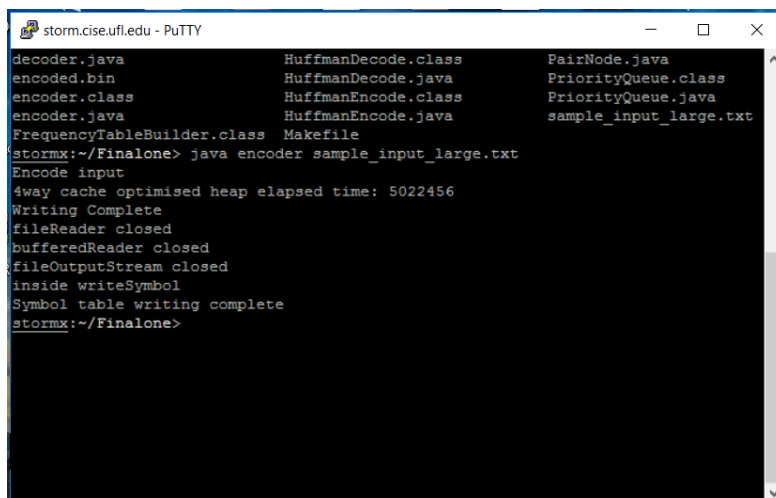
The elapsed time with each heap was noted.

Binary Heap: 5815080 microseconds

4 way Cache optimized heap: 5022456 microseconds

Pairing Heap: 8670122 microseconds

Since the time taken by the 4way cache optimized heap was the least, this data structure was chosen to be used.



```
storm.cise.ufl.edu - PuTTY
decoder.java      HuffmanDecode.class  PairNode.java
encoded.bin      HuffmanDecode.java   PriorityQueue.class
encoder.class     HuffmanEncode.class  PriorityQueue.java
encoder.java      HuffmanEncode.java   sample_input_large.txt
FrequencyTableBuilder.class  Makefile
stormx:~/Finalone> java encoder sample_input_large.txt
Encode input
4way cache optimised heap elapsed time: 5022456
Writing Complete
fileReader closed
bufferedReader closed
fileOutputStream closed
inside writeSymbol
Symbol table writing complete
stormx:~/Finalone>
```

```
storm.cise.ufl.edu - PuTTY
decoder.java      HuffmanDecode.class  PairNode.java
encoded.bin       HuffmanDecode.java  PriorityQueue.class
encoder.class     HuffmanEncode.class  PriorityQueue.java
encoder.java      HuffmanEncode.java    sample_input_large.txt
FrequencyTableBuilder.class  Makefile
stormx:~/Finalone> java encoder sample_input_large.txt
Encode input
Binary Heap elapsed time: 5815080
Writing Complete
fileReader closed
bufferedReader closed
fileOutputStream closed
inside writeSymbol
Symbol table writing complete
stormx:~/Finalone>
```

```
storm.cise.ufl.edu - PuTTY
decoder.java      HuffmanDecode.class  PairNode.java
encoded.bin       HuffmanDecode.java  PriorityQueue.class
encoder.class     HuffmanEncode.class  PriorityQueue.java
encoder.java      HuffmanEncode.java    sample_input_large.txt
FrequencyTableBuilder.class  Makefile
stormx:~/Finalone> java encoder sample_input_large.txt
Encode input
Pair Heap elapsed time: 8670122
Writing Complete
fileReader closed
bufferedReader closed
fileOutputStream closed
inside writeSymbol
Symbol table writing complete
stormx:~/Finalone>
```

Explanation:

The 4 way cache optimized heap performed best because of the mapping into the cache aligned array where Siblings are in same cache line.

This leads to $\sim \log_4 n$ cache misses for average remove min.

Cache optimization is achieved by shifting the first 3 elements.

Decoding algorithm used with complexity:

Complexity: $O(n*k)$ where n is the number of inputs in the code table and k is the height of the Huffman Tree formed (i.e. maximum length of code)

Decoding algorithm:

1. Bits from the stream are checked. If 0 obtained, go left. If 1 obtained, go right in the decoding tree
2. When tree leaf is reached, it means that a character has been decoded. This character is then placed into the output stream.
3. Similar procedure is followed with the remaining bits in the input stream. The immediate next bit will be the first bit representing next character.