Huffman Encoder Decoder

Table of Contents

Introduction

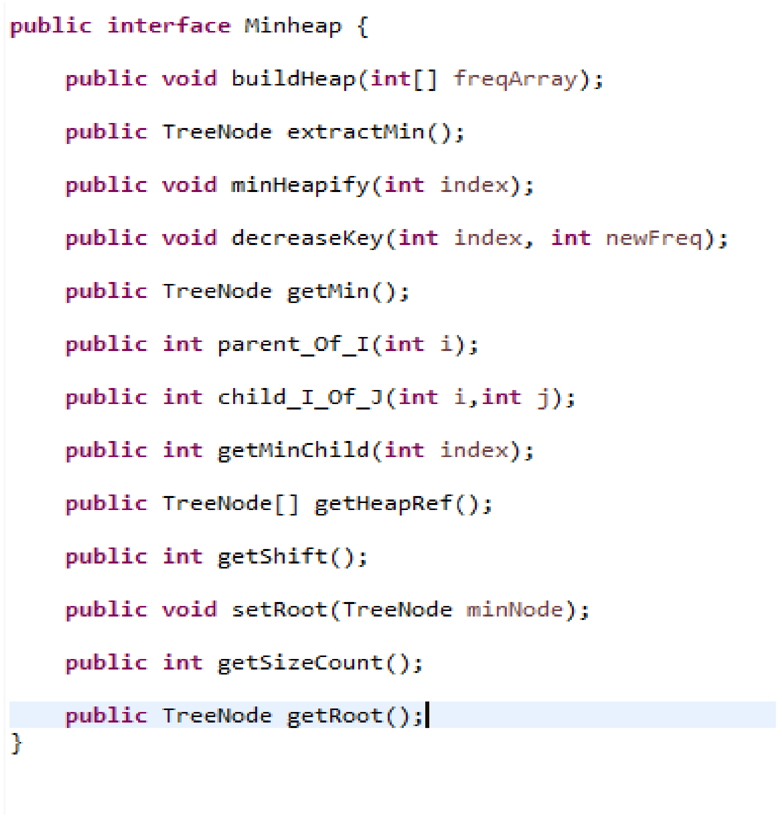
Transferring large amounts of data has always been a problem. Efficiently sending data has always been an area of research and many researchers came up with innovative algorithms to solve this problem.
The introduction of UNICODE, which increased the size of the total characters available, made the character set more robust. The robustness does come with a cost. The cost of transmission data has increased. Thus, an efficient encoding algorithm for is very important.
A famous encoding algorithm, Huffman encoding provides a very efficient way of data compression. This method helps us achieve an efficient encoding irrespective of the character encoding used.

Heaps (Priority queue)

Heaps are one of most efficient data structure for finding min/max elements in given data. Also a good method to sort elements in a list. Huffman encoding methods uses a Huffman tree for encoding. The process of building a Huffman tree is involves getting two items with minimum value. Min Priority Queue/ Heap can be used to implement the same.
I have implemented Minheap using a three data structures:

1. Binary Heap
2. Pairing Heap
3. 4-Way Cache Optimized heap

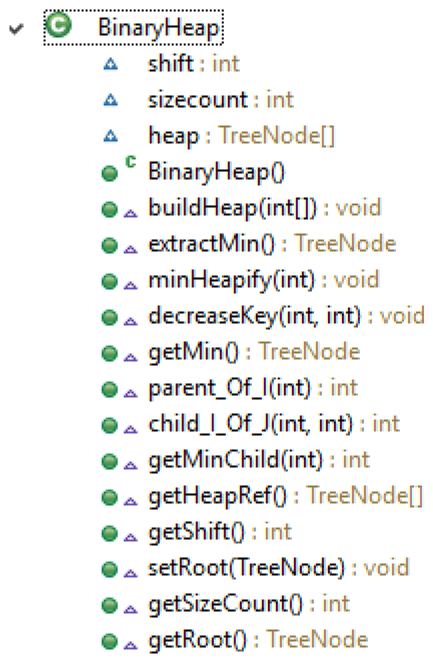
Have tried these three data stuctures to to understand which one runs the best to create a Huffman tree. I have created a interface of Minheap which is implemented by all the above data structures. Here's a snippet of the interface.



```java
public interface Minheap {

    public void buildHeap(int[] freqArray);

    public TreeNode extractMin();

    public void minHeapify(int index);

    public void decreaseKey(int index, int newFreq);

    public TreeNode getMin();

    public int parent_Of_I(int i);

    public int child_I_Of_J(int i,int j);

    public int getMinChild(int index);

    public TreeNode[] getHeapRef();

    public int getShift();

    public void setRoot(TreeNode minNode);

    public int getSizeCount();

    public TreeNode getRoot();
}
```

Binary Heap

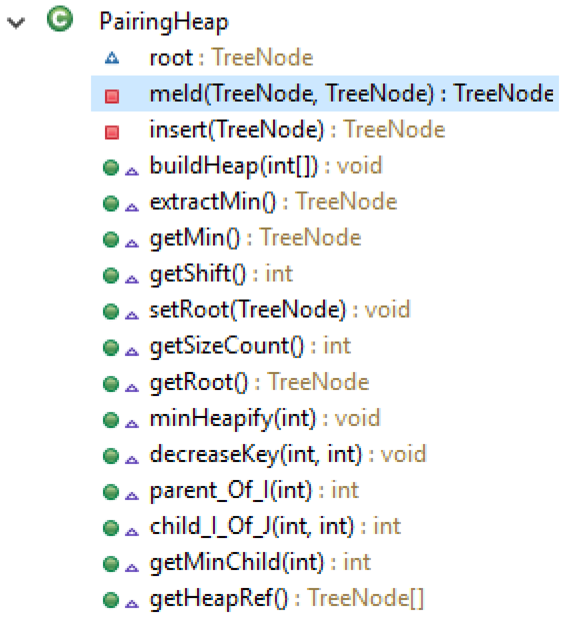
Binary heap can be viewed as a binary tree with certain properties. It is a complete binary tree with each root's value greater than or less than its children, based on whether it's a min heap or max heap.
There are four key functions: Min/Max Heapify O (log n), Extract Min/Max O (log n), Increase/Decrease Key O(log n) (includes taking care of possible violations).



```
BinaryHeap
    shift : int
    sizecount : int
    heap : TreeNode[]
    BinaryHeap()
    buildHeap(int[]) : void
    extractMin() : TreeNode
    minHeapify(int) : void
    decreaseKey(int, int) : void
    getMin() : TreeNode
    parent_Of_I(int) : int
    child_I_Of_J(int, int) : int
    getMinChild(int) : int
    getHeapRef() : TreeNode[]
    getShift() : int
    setRoot(TreeNode) : void
    getSizeCount() : int
    getRoot() : TreeNode
```

Pairing Heap:

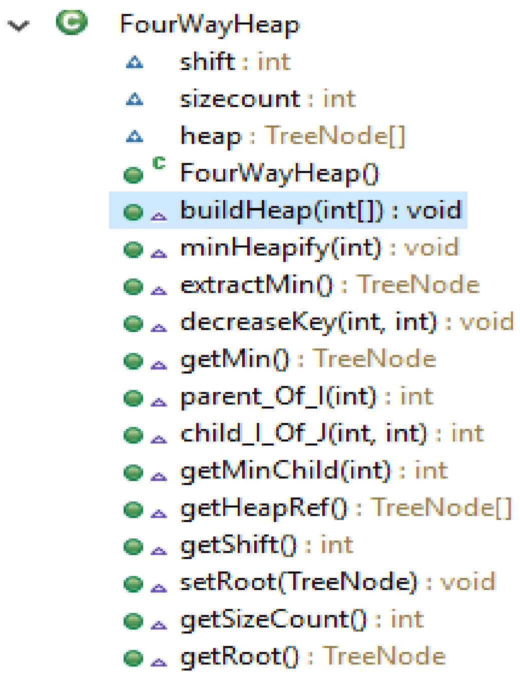
A min pairing heap in the one in which the operations are done in a specific manner. The key operation is the Meld operation. In the meld operation two nodes are selected and they are melded based on their values, here in Huffman encoder based on their frequency. In case of Min pairing heap, the node with greater value is appended to the left and is made as its child. The child points back to its parent. The nodes connected to the previous child remain intact and the new child points to the old child in the doubly linked list. The main operations are meld with complexity O(1), Remove Min/Max: O(n), Decrease/Increase Key O(1), BuildHeap: O(n)



```
PairingHeap
    root : TreeNode
    meld(TreeNode, TreeNode) : TreeNode
    insert(TreeNode) : TreeNode
    buildHeap(int[]) : void
    extractMin() : TreeNode
    getMin() : TreeNode
    getShift() : int
    setRoot(TreeNode) : void
    getSizeCount() : int
    getRoot() : TreeNode
    minHeapify(int) : void
    decreaseKey(int, int) : void
    parent_Of_I(int) : int
    child_I_Of_J(int, int) : int
    getMinChild(int) : int
    getHeapRef() : TreeNode[]
```

4-way cache optimized heap

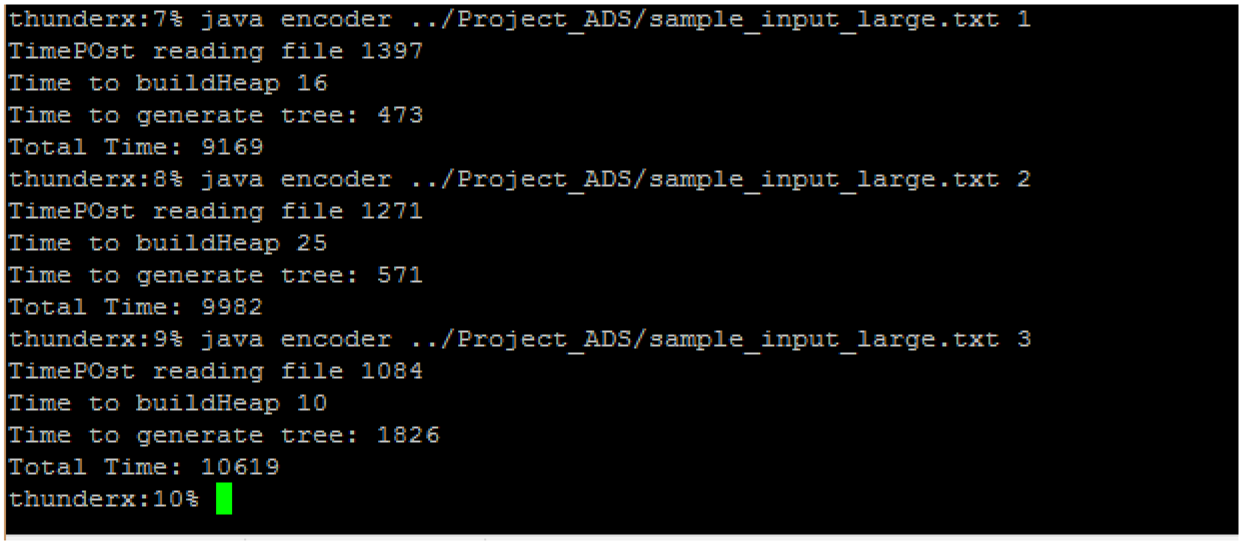
A 4-way heap is a special case of d-ary heap. The cache optimized case is the one where all the children of a node are accessed in a single cache hit. Thus, reducing the number of accesses and misses. Hence speeding up the overall time taken. Though the asymptotic complexity is unchanged the overall performance is better. Rest everything is similar to binary heap the methods and their algorithms.
Complexities: buildheap:O(n), minHeapify: O(log n), extractMin: O(log n), Decrease/increase Key: O(log n),



```
FourWayHeap
  shift : int
  sizecount : int
  heap : TreeNode[]
  FourWayHeap()
  buildHeap(int[]) : void
  minHeapify(int) : void
  extractMin() : TreeNode
  decreaseKey(int, int) : void
  getMin() : TreeNode
  parent_Of_I(int) : int
  child_I_Of_J(int, int) : int
  getMinChild(int) : int
  getHeapRef() : TreeNode[]
  getShift() : int
  setRoot(TreeNode) : void
  getSizeCount() : int
  getRoot() : TreeNode
```

Complexity Analysis:

To decide which heap implementation was better I ran 20 heap creation operation and Huffman tree creation operations on all the three data structures: - Binary Heap, Pairing Heap and 4-Way Cache Optimized Heap. I have attached a snapshot of my results below:


```
thunderx:7% java encoder ../Project_ADS/sample_input_large.txt 1
TimePOst reading file 1397
Time to buildHeap 16
Time to generate tree: 473
Total Time: 9169
thunderx:8% java encoder ../Project_ADS/sample_input_large.txt 2
TimePOst reading file 1271
Time to buildHeap 25
Time to generate tree: 571
Total Time: 9982
thunderx:9% java encoder ../Project_ADS/sample_input_large.txt 3
TimePOst reading file 1084
Time to buildHeap 10
Time to generate tree: 1826
Total Time: 10619
thunderx:10%
```

The second argument to the encoder explains which data structure is used
1: Four way Cache Optimized 2: Binary Heap 3. Pairing Heap

As seen from the above snapshot, the 4 way cache optimized heaps turns out to be faster than Binary and pairing heap. It is in line with my expectation as the creating a pairing heap is faster because of its O(1) insert operation compared to O(Log n) insert operations for Binary and 4-way heap cache optimized, but the overall time for Huffman tree creation is best for 4-Way Cache Optimized < Binary Heap < Pairing Heap.

Data Structure	Time for Build Heap	Time for Huffman Tree Generation
Four-Way Cache Optimized Heap	16 ms	**473** ms (Fastest)
Binary heap	25 ms	571 ms
Pairing Heap	**10** ms (Fastest)	1826 ms

The 4-way cache optimized works faster because of its cache friendly behaviour. I am using an array with size of 100,000 and using idea of counting sort to store a node in that. As node is an object type its reference is stored which is 8 byte of memory. The main aim to reduce the number of cache misses. It is ensured as siblings are stored in the same cache line, thus reducing the number of misses. This helps in faster construction of Huffman tree.
I have also implemented an optimization in Binary and 4-way cache optimized heap, where I actually perform one extract min and one readMin, thus reducing the number of heapify steps. From 3log n to 2 log n for each and every iteration in Huffman tree creation.

Huffman Algorithm

Introduction:

Huffman tree is a famous encoding/data compression algorithm. It uses the frequency of the input data to assign an encoding to it. The greater the frequency the smaller its encoding is. Following is the algorithm for it:

1. Huffman tree creation: **Input:** sample data file **output:** Huffman tree in the form of hashmap
2. Encoding using Huffman tree **Input:** sample data file and Huffman tree **Output:** encoded bin file and code table
3. Transmission of data (output from step 2) over a medium.
4. Decoding of data at the other user's end using the output from step 2. A decoder tree is constructed at the other users end and then the data is decoded using this tree.

Encoding:

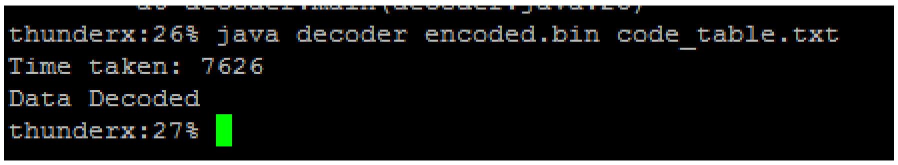
Following is the encoding algorithm- Considering 4-Way Cache Optimized heap:

1. Create the priority queue using the input data. Complexity: O(n) where n is no of entries
2. Create a Huffman tree by following the below steps: It uses a greedy algorithm whose complexity analysis is as follows: Sum cost of extract min and insert over the entire algorithm:

   (log n + log(n-1)) + (log (n-1) + log(n-2))…. => O(log(n!))

   Applying stirlings approximation: we get, => its bounded by O(nlogn) where n is the number of entries in the code table

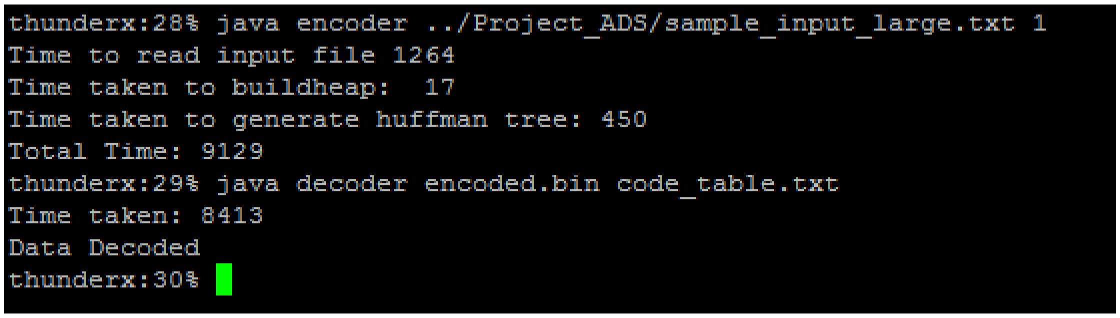
   a. Perform an extract min operation on the Heap, call this as min1
   b. Perform a readMin operation on the heap call this as min2
   c. Construct a new heapnode with null data and frequency as sum of min1 and min2.
   d. Perform an increase key at the root as there was the min.
   e. Then perform the heapify operation to make sure all data are consistent in the heap.
   f. Perform steps a to f till there is only one node left in the heap. Terminate there.
   g. Huffman tree created
3. Run through the input again and use the Huffman tree's data in stored in the form of a HashMap, the data – encoded value mapping Complexity: O(1) for each hashmap access and O(n) for writing data into the file where n is the size of the input.
4. The code once generated is appended to a bitset, because it's very efficient as it consumes less space and it is more advisable to store data in a buffer first rather than writing it continuously to file, it makes the code slower as the number of disk accesses increase.
5. The result of the above steps is an encoded .bin file and the code table is stored as a .txt file.



```
thunderx:26% java decoder encoded.bin code_table.txt
Time taken: 7626
Data Decoded
thunderx:27% 
```

Decoding:

1. The input is: code table and binary file.
2. Decoder tree generation:
   a. The code table is used to generate a Trie like structure using the encoded code information for that data.
   b. For every character in the string of encoded data in the table:
      If we get a 0 we go left, else we go right. If there is not a node already we add one else we go down to next and change the current node for next iteration.
      When we reach the end of the string we mark isEnd=true and put data value as the corresponding value of the data whose the encoded string it maps to.
   c. Do the same for every entry in the code table.
   d. Decoder tree is created.
3. The tree created is used to decode the stream of data and an entry is added whenever isEnd=true.
4. The property of Huffman codes is that no code is a prefix of another simplifies the possible scope of ambiguity in the decoder tree.
5. The procedure is done for the entire binary input data.



```
thunderx:28% java encoder ../Project_ADS/sample_input_large.txt 1
Time to read input file 1264
Time taken to buildheap:  17
Time taken to generate huffman tree: 450
Total Time: 9129
thunderx:29% java decoder encoded.bin code_table.txt
Time taken: 8413
Data Decoded
thunderx:30%
```

Complexity Analysis:

Building this trie speeds up the decoding process. The complexity to build the trie is O(M*N) where M is size of the largest encoded data string and N is the number of the entries in the code table, that is number of unique data.

The complexity of a single find is the size of the input, that is O(M). where M is the length of the code corresponding to a value.
The overall decoding complexity using decoder tree is O(M*N) where M is the length of the code corresponding to a value and N is the total number of times we insert data in the file. The above bound is loose as we multiply N with largest M. The tighter complexity would be the number of bits in the encoded.bin file.

Code Structure:

TreeNode: Structure of Huffman tree node/heap node
MinHeap Interface: defines the interface which is a skeleton of how all other heaps need to be implemented.
Binary Heap: Implements MinHeap Interface and has code for Binary Heap

FourWayHeap: Implements MinHeap Interface and has code for FourWayHeap
PairingHeap: Implements MinHeap Interface and has code for Pairing Heap
HuffmanTreeConstructor: Constructs the Huffman tree from the heap
CodeValueGenerator: Generates the code table for the data in the Huffman tree
EncoderMain: Is the driver for the encoder program
DecoderNode: Structure of decoder node
DecoderTree: Implementation of the decoder tree
DecoderMain: Is the driver of decoder program

References:

1. Thomas Cormen
2. Prof. Sartaj Sahni Lecture Slides