University of Florida

Project Report to COP5536

Spring 2017

Huffman Coding

ADS Project Report

Ramona Maria Juliet Vaz
UFID:8148-6697
rvaz@ufl.edu

April 6, 2017

1 Problem Statement

Software giant Toggle recently bought video streaming site MyTube. Now MyTube needs to send its data to Toggle server. As enormous amount of data will be transferred, they decided to use Huffman coding to reduce data size. You are a software engineer in MyTube and asked by your manager to write code for Huffman encoder and decoder.

2 Function Prototypes and Structure of Program

Huffman code is optimal in the sense that it encodes the most frequent characters in a message with the fewest bits, and the least frequent characters with the most bits. It is also a prefix-free code. There can be no code that can encode the message in fewer bits than this one. It also means that no codeword can be a prefix of another code word. This makes decoding a Huffman encoded message particularly easy: as the bits come by, as soon as you see a code word, you are done since no other code word can have this prefix. The general idea behind making a Huffman code is to find the frequencies of the characters being encoded, and then making a Huffman tree based upon these frequencies. Huffman codes are prefix-free codes - every character to be encoded is a leaf node. There can be no node for a character code at an internal node (i.e. non-leaf). Otherwise, we would have characters whose bit code would be a subset of another character's bit code, and we wouldn't be able to know where one character ended and the next began.

2.1 Encoder

2.1.1 Algorithm for the Huffman Tree

- Scan the message to be encoded, and count the frequency of every character
- Create a single node tree for each character and its frequency and place into a priority queue (4-way cache optimized heap) with the lowest frequency at the root
- Until the forest contains only 1 tree do the following:
  - Remove the two nodes with the minimum frequencies from the priority queue (we now have the 2 least frequent nodes in the tree)
  - Make these 2 nodes children of a new combined node with frequency equal to the sum of the 2 nodes frequencies
  - Insert this new node into the priority queue

2.1.2 Encoder in the current context

The encoder program at a high level makes use of

- HashMap containing the frequencies of the entries in the input file called ”freqTable”. The ”key,value” pair is nothing but the ”entry,frequency of the entry”

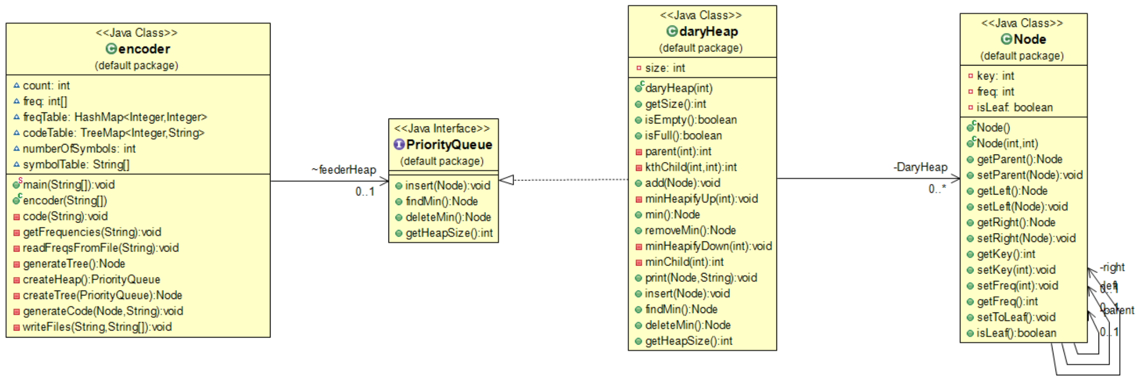


Figure 1: Class Diagram for the Encoder

- "freqTable" entries thus generated are used in the generation of the 4-way cache optimized heap called "feederHeap" from which the Huffman Tree is formed.
- Until the "feederHeap" contains only 1 tree do the following:
  - The two nodes with the minimum frequencies from the feederheap (we now have the 2 least frequent nodes in the tree) are extracted.
  - These 2 nodes become children of a new combined node with frequency equal to the sum of the 2 nodes frequencies
  - The new node is thus reinserted into "feederHeap".
- At the end of this operation the remaining node has the structure of the Huffman Tree which we will traverse and generate the code for by passing this node as an argument to the method "generateCode".

2.1.3 Function Prototype Description

- **private void readFreqsFromFile(String PATH)**: The method "readFreqsFromFile" takes the path at which the input file lies and processes the entries in the input file and creates the HashMap "freqTable" containing the frequencies.
- **private PriorityQueue createHeap()**:The "freqTable" is then consumed by the "createHeap()" method in which a new object of the "daryHeap.java" called "feederHeap" is created. A node object is then created with the "entry, frequency of the entry" which is inserted in the "feederHeap".
- **private Node createTree(PriorityQueue priorityQueue)**:The fully formed "feederHeap" is then consumed by the "createTree()" method in which until the "feederHeap" contains only one tree the following is done.
  - The two nodes with the minimum frequencies from the feederheap (we now have the 2 least frequent nodes in the tree) are extracted.

  - These 2 nodes become children of a new combined node with frequency equal to the sum of the 2 nodes frequencies
  - The new node is thus reinserted into "feederHeap".

- At the end of this operation the remaining node has the structure of the Huffman Tree which we will traverse and generate the code for by passing this node as an argument to the method "generateCode"

- **private void generateCode(Node node, String code)**: The code thus generated for each leaf node of the generated tree is then saved in HashMap called "codeTable" which we use to iterate over and write to a text file in the method "writeFiles"

- **private void writeFiles(String PATH)**:Each entry in the input file has now to be replaced by the Huffman code generated and written to the file encoded.bin, however these codes should be in bit form and therefore we have a BitSet buffer to hold all the bits of the encoded entries, we achieve file compression using this technique.

2.2 Decoder

The decoder program at a high level

- Reads the code table text file that was generated in the encoder.

- Create a root node and then for every character in the code encountered either traverse left if '0' or right if '1' and then form a new node if the node does not exist already and set the code equivalent in the node

- Now the encoded file has to be read and the tree that was constructed is traversed and the value at each leaf node is read which is the value that was encoded, essentially the decoded output.

2.2.1 Function Prototype Description

- **private void reconstructedTree(String CODETABLEPATH)**:The method "reconstructedTree" takes the path at which the code table file lies and processes the entries in the file and adds the key, code sequence to the root of the Huffman Tree either as a left leaf or right leaf node.

- **public void getDecodedMessage(String ENCODEDPATH)**:The method "getDecodedMessage" reads the encoded file and traverses the tree stopping, wherever leaf node values match the binary string and writes the output to the the decoded.txt file.

3 Performance Analysis Results and Explanation

The performance analysis results for creating the Huffman Tree for the three data structures are as follows:

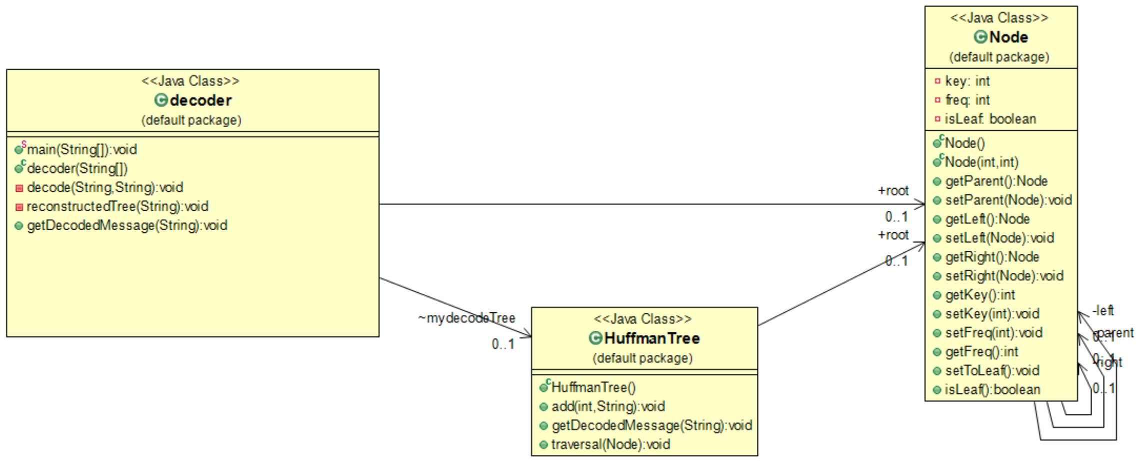


Figure 2: Class Diagram for the Decoder

- 4-way cache optimized Heap elapsed time: 5254290 μs
- Binary Heap elapsed time: 5816112 μs
- Pair Heap elapsed time: 8680138 μs

These results were obtained from the average time for creating the Huffman Tree over ten iterations. As we can see from the results the 4-way cache optimized heap is the best data structure to use as the underlying Priority Queue.

3.1 Data Structures competing to be Priority Queue

The data structures that were in consideration to be the Priority Queue:

- 4-way cache optimized Heap
- Binary Heap
- Pair Heap elapsed time

3.1.1 Binary Heap

Implements the PriorityQueue interface that has four methods primarily

- **public void insert(Node x)**: Put x at "next" leaf position.Percolate up by repeatedly exchanging node until no longer needed
- **public Node findMin()**:returns root of the heap.
- **public Node deleteMin()**:
  - Remove root (that is always the min!)

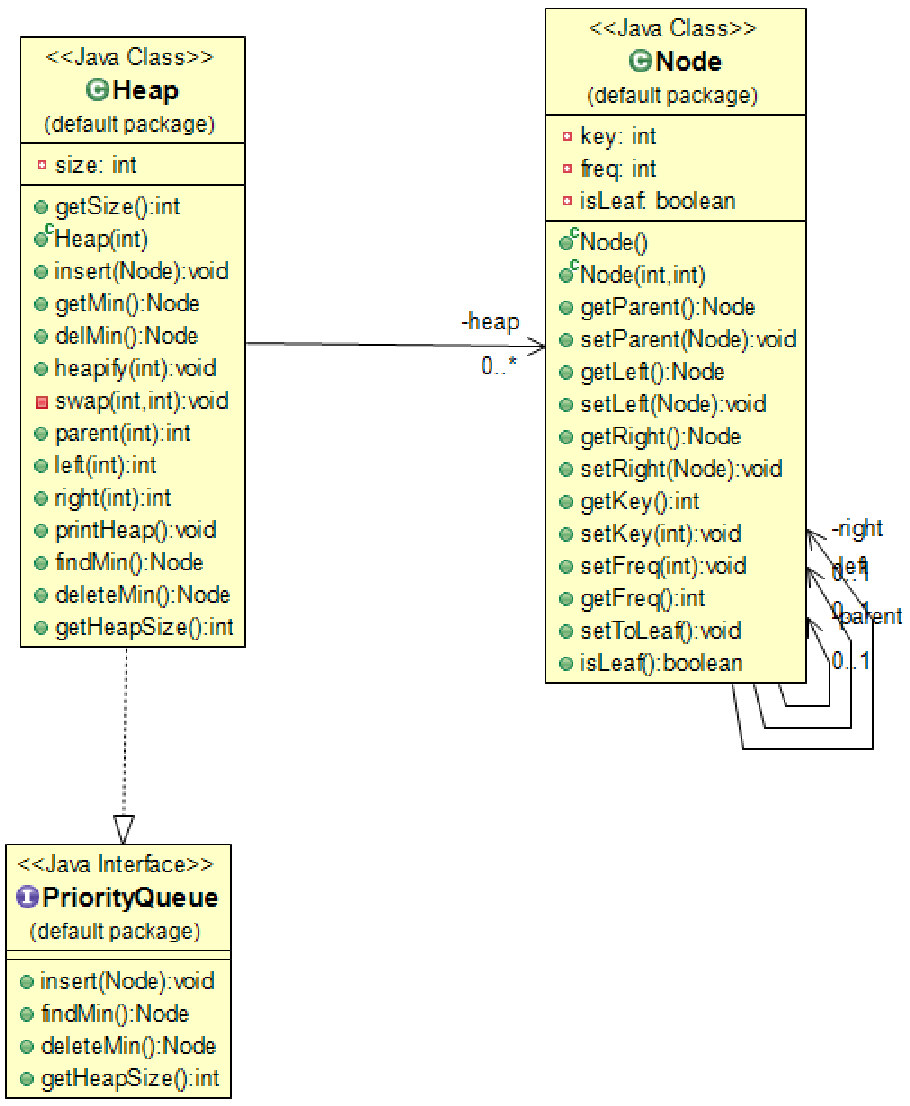


Figure 3: Class Diagram for the Binary Heap

- Put “last” leaf node at root
- Find smallest child of node
- Swap node with its smallest child if needed.
- Repeat steps 3 and 4 until no swaps needed.

- **public int getHeapSize()**:returns heap size

3.1.2 4-way cache optimized Heap

The 4-way heap essentially overcomes the limitations of binary heap which are:

- Each percolate step looks at only two new nodes

- Operations jump widely through the heap

In a 4-way cache optimized heap

- fit one set of children in a cache line
- fit one set of children on a memory page/disk block

Implements the PriorityQueue interface that has four methods primarily

- **public void insert(Node x)**: Put x at "next" leaf position.Percolate up by repeatedly exchanging node until no longer needed
- **public Node findMin()**:returns root of the heap
- **public Node deleteMin()**:
  - Remove root (that is always the min!)
  - Put "last" leaf node at root
  - Find smallest child of node
  - Swap node with its smallest child if needed.
  - Repeat steps 3 and 4 until no swaps needed.
- **public int getHeapSize()**:returns heap size

3.1.3 Pair Heap

Implements the PriorityQueue interface that has four methods primarily

- **public void insert(Node x)**:Create 1-element min tree with new item and meld with existing min pairing heap.
- **public Node findMin()**: returns root of the tree
- **public Node deleteMin()**:The only non-trivial fundamental operation is the deletion of the minimum element from the heap. The standard strategy first merges the subheaps in pairs (this is the step that gave this datastructure its name) from left to right and then merges the resulting list of heaps from right to left
- **public int getHeapSize()**: returns heap size

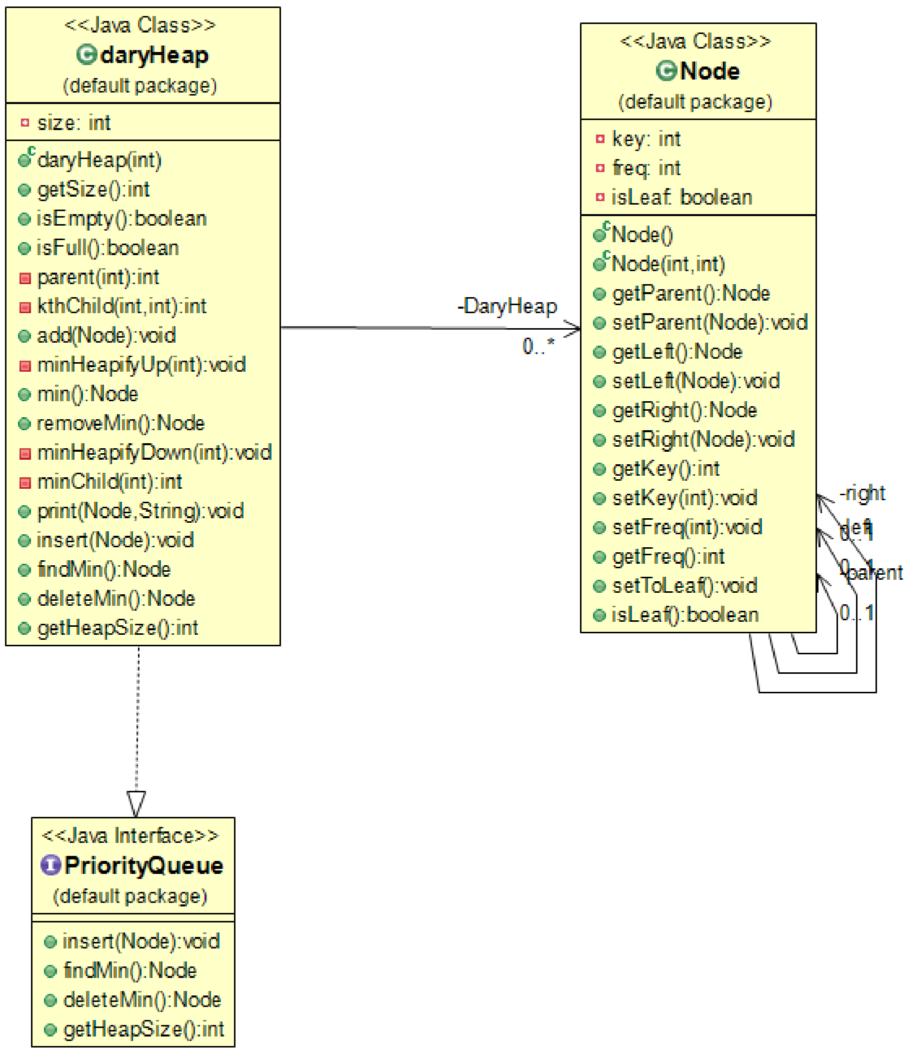


Figure 4: Class Diagram for the daryHeap

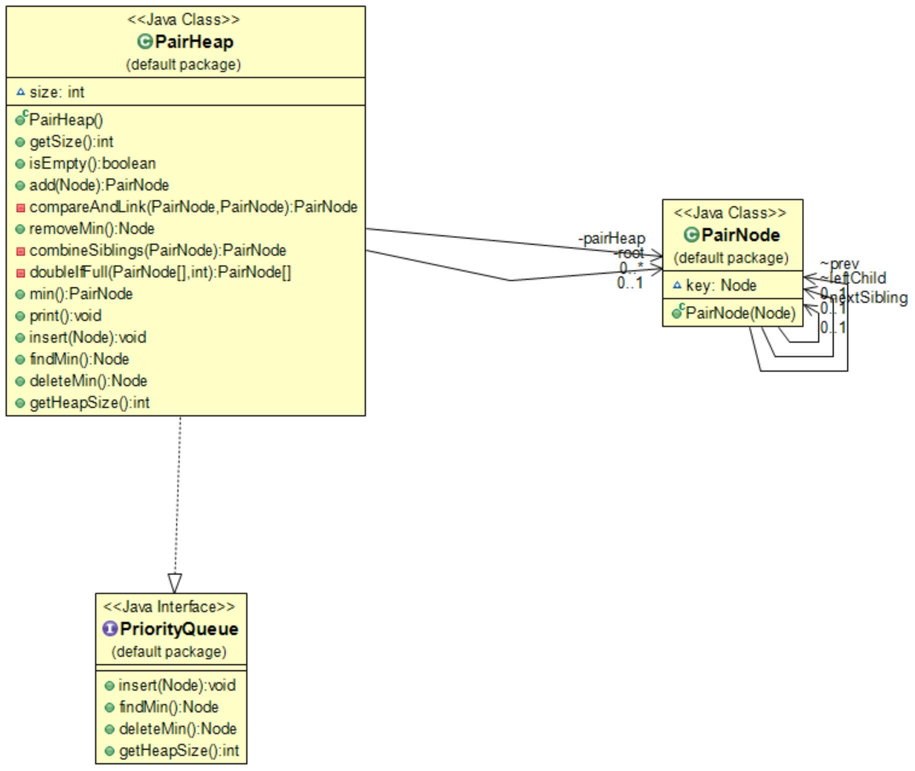
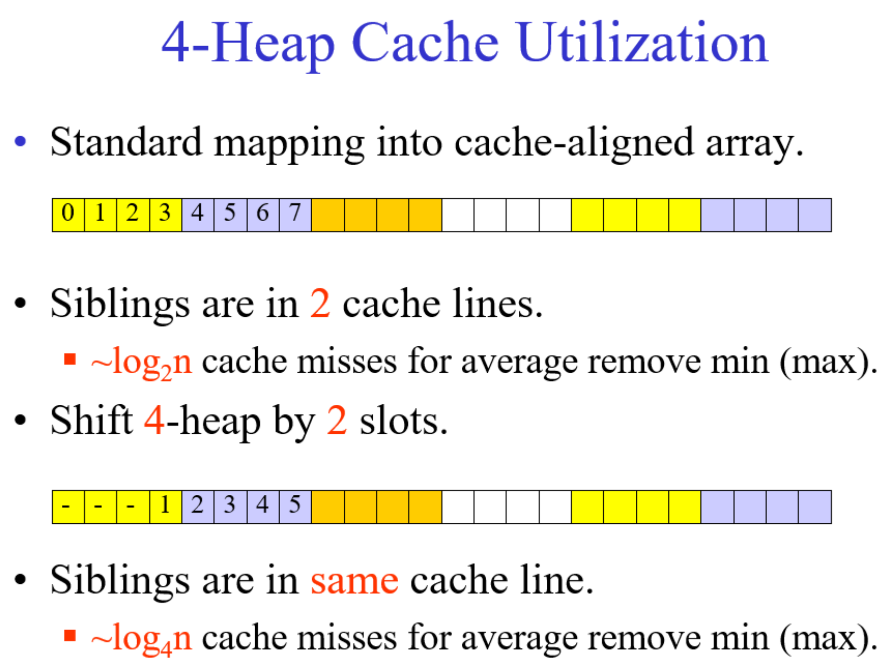


Figure 5: Class Diagram for the Pair Heap



4-Heap Cache Utilization

- Standard mapping into cache-aligned array.

- Siblings are in 2 cache lines.
  - ~$\log_2 n$ cache misses for average remove min (max).
- Shift 4-heap by 2 slots.

- Siblings are in same cache line.
  - ~$\log_4 n$ cache misses for average remove min (max).

Figure 6: Cache Aligned 4-way heap

3.2 4-way cache optimized heap:"better than the rest!"

The 4-way cache optimized heap has the following attributes that have contributed to making it relatively faster than other data structures:

- Worst-case insert moves up half as many levels as when d = 2.
- Average remains at about 1.6 levels.
- Remove-min operations now do 4 compares per level rather than 2 (determine smallest child and see if this child is smaller than element being relocated).
- Other operations like move small element up, loop iterations, associated with remove min are halved.
- Cache Optimization is achieved by shifting the first 3 elements so that when the children of the parent node lie in the same cache line and cache misses can be avoided as shown in the figure below.

4 Decoding algorithm and its complexity

4.1 Algorithm for the Decoder

The algorithm for the decoder is fairly simple and is elaborated in the steps below.

- Starting with the first bit in the stream, one then uses successive bits from the stream to determine whether to go left or right in the decoding tree.
- When we reach a leaf of the tree, we've decoded a character, so we place that character onto the (uncompressed) output stream.
- The next bit in the input stream is the first bit of the next character.

4.2 Complexity of the Decoding Algorithm

Considering 'n' number of inputs in the code table and 'k' to be the max length of the code.

- While constructing the Huffman Tree, the max depth of the tree is 'k' and since there are 'n' entries , the time complexity is O(n*k)
- While reading the encoded string we will in the worst case traverse the depth of the tree which is 'k' and traverse it 'n' times for 'n' inputs giving us a time complexity of O(n*k)

Time complexity will be **O(n*k)** where n is the number of inputs and k is the maximum number of digits in the code.