HUFFMAN CODING USING PRIORITY QUEUE

ADVANCED DATA STRUCTURES

SPRING 2017

By: Yagna Namburi

UFID:1190-5171

Huffman Coding

Huffman tree is a data structure which is used for lossless data compression. It is data structure which has the data in the leaf nodes and the tree is constructed with taking two data nodes with lower frequency, combining them and this is combined with another node with the next lower frequency. After constructing the entire tree the data with higher frequency is closer to the root and the one with least frequency is farthest from the root. The reason to build it this way is to have minimum weighted external path length. The Huffman tree algorithm I have implemented first takes the input data from sample_input.txt and for each key in the text document it determines how many times it occurs which is the frequency of the key. This key and it corresponding frequency values are stored in a frequency table. I needed a data structure to store these keys along with their frequencies before constructing a Huffman tree and I was given the choice to use binary heap, 4-way caching heap and pairing heap as a priority queue. After analyzing that 4-way caching heap is faster than the other two I used it for construction of my Huffman tree in the encoder and it generated two files encoded.bin and codetable.txt which were later used for decoding by building decoder tree and then using codetable for generation of decoded.txt. All the functions and classes I have generated are mentioned below.

Function prototypes and Structure of the program:

First I wanted to analyze which priority queue gives the optimum performance and for that I created all the three priority queues. First I wanted to create a binary heap and for that I wanted to create a node class

Node.java: This is a class I created which is used to create nodes for binary heap and 4-way caching heap. These are the members of the Node.java class

- **Node(int freq,int i):** This is a constructor of Node class which takes frequency and the key as arguments.
- **Node getchild(int i):** This function takes index as argument and returns a node which id the child of that given indexed node.

- **Void addchild(Node node):** This function takes a node as argument and adds it as a child to another node.
- **Int getkey():** This function returns a key of a particular node. Here the key is an integer value which will be returned.
- **Void setkey(int key):** This function is used to set the value of a key of a node. It takes integer value as an argument.
- **Void setFreq(int freq):** This function is used to set frequency of a node to a value which is given as an argument to the function.
- **Int getFreq():** This function returns an integer value which is the frequency of a particular node.
- **Void setToLeaf():** This function is used to set a node as a leaf .
- **Boolean isLeaf():** This function is used to tell whether a node is a leaf or not. If the node is a leaf it returns true else returns false.
- **Int compareTo(Node tmp):** This function compares two nodes by taking into account their frequencies. It takes a node as an argument with which it compares the current node to.
- **Boolean isChildEmpty():** This functions returns true if a node's children are empty.
- **String toString():** This function is used to convert a set of characters into a string and returns that string.

After creating a node I created binary heap and four way heap using the same node except for the fact that the value of d is 2 for binary and 4 for 4-way cached which indicated the number of children for each node in the respective heaps.

BinaryHeap.java: I created a class called BinaryHeap which takes the nodes and creates a binary heap and uses a vector to store the nodes in the heap. It has the following members.

- **BinaryHeap(int capacity):** This is a constructor of BinaryHeap class which takes an integer as parameter and initializes the capacity of the vector with a value capacity+1.
- **Boolean IsEmpty():** It returns whether the heap is empty or not.
- **Void clear():** It clears the entire heap structure.

- **Int parent(int i):** It returns the index of the parent node of the node with index i.
- **Int kthchild(int I,int k):** It returns the kth child of the node with index i.
- **Void Insert(Node x):** This method is used to insert a node into the heap.
- **Node delete(int ind):** It is used to delete a node at a particular index. This takes integer as argument and return a node.
- **Void heapifyUp(int childInd):**
- **Void heapifyDown(int ind):**
- **Int minChild(int ind):** This method returns the index of the minimum child of a particular node of index I based on the frequency.
- **Int getsize():** This gives an integer value which is the size of the heap which is the number of nodes present in the heap.
- **Int getChild(int nodeindex,int childnum):** This method returns the index of any particular child based on the child number.
- **Node getStartNode():** This gives the first node in the heap.
- **Int getfrequency(int i):** it is used to get the frequency of a particular node of index i.
- **Node getNode(int i):** It is used to get the node at a particular r index i.
- **Void setNode(int i,int freq):** This method is used to set the frequency of a given node of index i.

FourwayCache.java: I created a Fourway heap structure using the same node as that of the binary heap. The only change is the d value which is 4 for heap because of the no. of children which is four for fourway heap.

PairHeap.java: I created a PairHeap class to generate pairing heap and every node in the heap is of the type Pairnode. These are the members of the class.

- **Pairnode insert(Node x):** This method is used to insert a node into the heap. This takes a node as input and returns a pairnode.
- **Pairnode companlin(Pairnode p1,Pairnode p2):** This methos takes two pairnodes as input and gives a pairnode as output.
- **Pairnode combinesib(Pairnode p1):** This method is used to combine sibblings. This method takes pairnode as input and gives pairnode as output.
- **void encode(Node node,String code,final<vector> prefix):** This method takes a node,string and vector as input and encodes the pairing heap.

- **Node delmin():** This method deletes and returns the node with minimum frequency.

Pairnode.java: This is a class which is the building block of pairing heap. This class is used to generate Pairnodes each of which contains a Node and three Pairnodes.

After creating all the three priority queues and testing them I found that fourway heap is faster than the other two. So I took fourway heap to go ahead with the project. Using fourway heap I created a Huffman tree and then I wanted to encode the Huffman tree for which I created an Encoder class.

encoder.java: This is a class which is used for encoding the Huffman tree. These are the members of the class.

- **Void genct(Node n,String c):** This function is used to generate the code table. It takes node and string as inputs.
- **void encode(Map<integer,string> codeTable):** This function internally calls buildencodebitmap() to generate encoded.bin file
- **HashTable<String,Byte> buildingencodebitmap():** This function appends "1" or "0" depending on the branch. This returns a hashtable whose key is of typr is string and value is of type byte.

decoder.java: This class takes encoded.bin and code_table.txt as input and then decodes the coded input to decoded output which will be same as the input file. It has the following members:

- **void decode(String[] args):** This method takes array of strings as input and does not return anything.
- **Decodebm(Map<byte,string> decodedbm):** This function takes a map as input and does not return anything.
- **String bintostr(Decodenode r, String decstr):** This takes a node and string as input and returns a string.
- This class uses a method in another class Parse.java called parsectfile().
- **Decnode parsectfile():** This method is used to parse through the codetable file and it returns Decnode.

Decnode.java: This is a class which creates nodes with element and two children.

Structure:

1. First the encoder.java file is executed and input file is given as input for the class
2. The frequency table is created from the values in the input file.
3. Then creation of priority queues takes place and it first creates fourway caching heap and the binary heap using nodes from Node.java class. Then it creates pairingheap using nodes in Pairnode.java.
4. After three data structures are created Huffman trees are created for the data structures for the three data structures.
5. Then the data structure with lowest execution time i.e. fourway caching is taken to encode the data and gives encoded.bin and code_table.txt.
6. Then decoder.java class is executed. In this class encoded.bin and code_table.txt are taken as input as decoding is done and decoded.txt is the output.

Performance analysis:

```
thunderx:58% java encoder sample_input_large.txt
total time for fourwayheap:29341
total time for binaryheap :34832
total time for pairheap :31456
4way Heap
thunderx:59% java decoder encoded.bin code_table.txt
time for decoding: 80475
thunderx:60% diff decoded.txt sample_input_large.txt
thunderx:61% java encoder sample_input_large.txt
total time for fourwayheap:25042
total time for binaryheap :27426
total time for pairheap :30468
4way Heap
thunderx:62% java encoder sample_input_large.txt
total time for fourwayheap:30236
total time for binaryheap :31152
total time for pairheap :38364
4way Heap
```

From the above analysis it is clear that the time taken for 4 way cache heap is lesser than binary and pairing heap. It is faster than binary heap because the height of 4 way cache is less than the binary heap as each and very node has four children in 4 way cache heap where as there are only two children for each node in binary heap. This is faster than pairing heap because pairing heap might be taking lot of time for searching the nodes.

The Huffman tree is a good way to send data by coding as there is no loss of data which can be seen from the same size of input and output files which is 68000KB.

Algorithm used for decoding and its complexity:

The algorithm for decoding I used is:

- I first created a code table which has the key and its code.
- Then I created a dummy hashmap which has codes of the codetable as keys and keys of the codetable as values.
- The decode tree is created by going through each and every (key,value) pair in the dummy hashmap and if a node is present at the (length of the key-1) height then no node is inserted and if there is no node is present then a node is inserted with the corresponding value of the key.
- During the selection of position of the node at a particular height if the bit in the key is “0”then the node is inserted at the left child of the node and if the bit in the key is “1” then it is inserted in the right of the node.
- When a key is completely parsed go back to the root node and do the same for the next keys until no (key,value) pairs are left unparsed.

The complexity of this decoder algorithm is n*k where n is the no. of values in the input file including repetitions and k is the no. of bits in the code of that value.

Conclusion:

After my analysis I found that huffman tree is the best way to send data in the encoded format as there is no loss of data and from the pairing heaps I was given 4 way cached heap gives the optimum performance.