Advance Data Structures
COP 5536

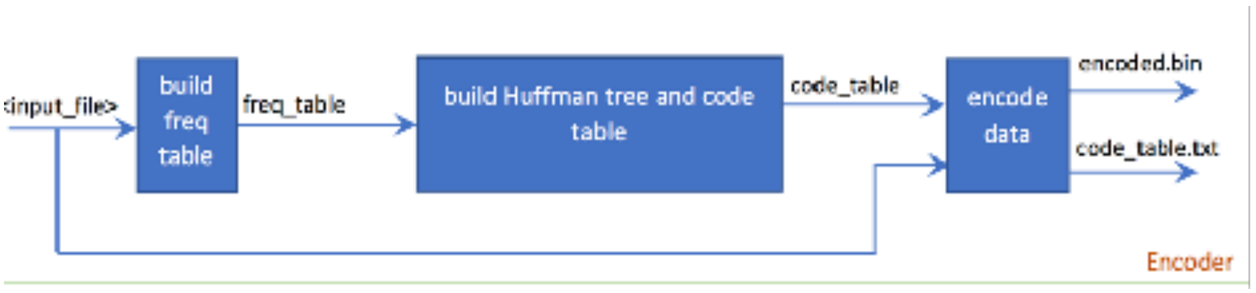
PROGRAMMING PROJECT REPORT
SPRING 2017

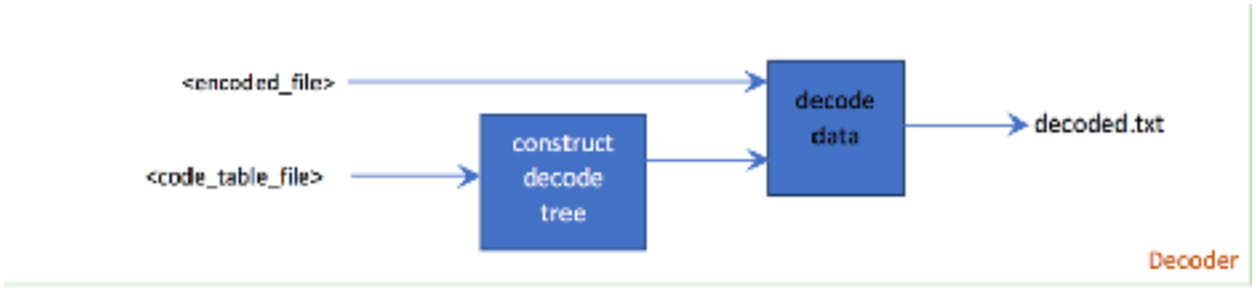
SUBMITTED BY

ANMOL KHANNA
UFID: 6514-0549
anmolkhanna93@ufl.edu

Introduction

In the given project we have to design a Data Compression System by implementing Huffman Encoding which can be used by MyTube (Acquired by toggle) to transfer huge amount of data to the toggle servers. As a programmer, the main task was to design an encoder and decoder for the compression mechanism. The encoder takes in an input file and gives out two files namely encoded.bin and code_table.txt. These two files are then used by our decoder to generate the original message sent to the encoder.





The main data structures implemented for the project were :

1. Pairing Heap :   A min(or max) pairing heap is a min or max tree in which the operations are performed in a special manner. This is a multiway tree structure, which is considered robust for implementing algorithms like Minimum Spanning tree algorithms (Eg. Prim’s).

Following is the actual time complexity of the operations performed by the pairing heap.

Operation	Complexity
Insert	O(1)
Remove Min (or Max)	O(n)
Meld	O(1)
Remove	O(n)
Decrease or Increase Key	O(1)

2. Cache Optimized four way heap : These are special types of d-ary heaps in which we have a speed up of about 1.5 to 1.8 when sorting about 1 million elements using heap sort. These tend to perform usually or better than other d-heaps. The performance increase with this data structure is that to maintains a cache for the operations which makes the operations to be executed at faster pace than our normal d-ary heaps.

3. Binary Heaps : This is a special data structure of d-ary heaps with d having value of two. A binary heap satisfies the property of a complete binary tree and it satisfies the heap ordering property. We can have two types of binary heaps namely the min heap and the max heap. In case of a min heap, the value of the root is the minimum value among all the values in the heap and the value at each node is greater than or equal to its parent node value. While in case of a max heap, the value at the root is the maximum of all the values present in the heap and the value at each node is less than or equal to the value of its parent node. Following is the time complexity of the operations performed by a min heap.

OPERATION	**COMPLEXITY**
Insert	O(log n)
deleteMin	O(log n)
Remove	O(log n)
findMin	O(1)

PROJECT ENVIRONMENT

Hardware Requirements :

COMPONENT	VALUE
Hard Disk	4 GB Minimum
Memory	512 MB
Operating System	Mac OS Sierra
CPU	x86
COMPILER	javac 1.8.0_121

Compilation Instructions :

The project has been implemented in Java Language. The code has been tested on Java environment of thunder.cise.ufl.edu and it produces the results as is expected i.e. the size of the encoded.bin matches the one for testing purposes and the output message is same as the input message.

To connect to the thunder server :

```
username@thunder.cise.ufl.edu
```

As per the project requirements, I have used a Make file to create the executable files which can be run using the following commands :

```
$ java encoder  <input_file_name>
```

```
$ java decoder <encoded_file_name> <code_table_file_name>
```

PROGRAM STRUCTURE

Packages imported:-

- import java.util.*;
- import java.util.List;
- import java.util.LinkedList;
- import java.util.Hashtable;
- import java.io.FileReader;
- import java.io.BufferedReader;
- import java.io.FileNotFoundException;
- import java.io.IOException;
- import java.io.FileWriter;
- import java.io.BufferedWriter;

The program uses the following classes :

1. binaryNode.java
2. binaryHeap.java
3. encoder.java
4. decoder.java
5. fileHandling.java

6. fourWayHeap.java

7.huffmanCodeGeneration.java

8. pairingHeap.java

9. pairingHeapNode.java

binaryNode.java

● This class contains the structure of a node and the getter/setter function for its properties.

● Properties of a node are following:-

  ○ data:- Data present in the node.

  ○ frequency :- the frequency of the occurrence of the data item.

  ○ Vector of nodes : it contains the elements for the heaps.

● Methods supported in this class:-

○ public void setData():-

■ Set the data of the node object.

○public int getData():-

■ returns the data for the current node object.

○ public void isEmpty():-

■ Used to check if the vector of items is empty or not.

○ public void setFrequency():-

■ Used to set the frequency parameter of the object

○ public int compareTo():-

■ used to perform the comparisons of the frequency values when the node is created.

binaryHeap.java

The following methods are implemented in this class.

○ public int getParent():-

■ returns the parent for the object.

○public int getChildren():-

■ returns the children for the current parent object.

○ public void downHeapify():-

■ Used to re adjust the heap after the node is deleted.

○ public void upHeapify():-

■ Used to re adjust the heap after the node is inserted.

○ public void deleteMin():-

■ Used to delete the minimum element in the heap, the minimum element is always the root.

○ public void isEmpty():-

■ to check if the heap is empty or not.

○ public void encodeTree():-

■ Used to traverse the heap structure and to assign the code values while heap traversal.

encoder.java

This is a class that contains a main method and will be run to perform the encoding operation. This class takes in as a input filename from the command line and gives two files encoded.bin and code_table.txt

decoder.java

This class takes in as input the output from the encoder and gives as output decoded.txt which matches are input message. Following functions have been implemented in this class.

- public Node decode():-
  - this takes in as input from the code_table.txt and creates a decode tree for our message which is then used out decoder along with the encoded.bin to regenerate the original input message.
- public String convertBinaryToString():-
  - this function reads from the encoded.bin and the decoder table to generate the input message line by line which can be used to write to the decoded.txt file.
- public void createDecodingBitMap():-
  - Used by the decode method above to assign code values while creation and traversing of the decoder tree.

fileHandling.java

This class file contains all the functions for file read and write operations which are required for the purpose of our project. Some of the functions I have implemented for this class are as follows :

○ public void scanFile():-

■ This function reads in the input file and starts the encoding process. i.e. it makes the appropriate function calls to generate the Huffman tree and start the process of creation of code table and the encoded message.

○ public void encode():-

■ This function starts the encoding of the input message and writes the encoded message to the encoded.bin file.

○ public void EncodingMap():-

■ This function is used for creation of our code_table.txt.This works by first reading the input file and then creating a HashMap of the input values along with their frequency and then used to write the code table file.

○ public void decodeTree(Map<Integer,String> code_table):-

■ This function is used for creation of our decoder tree from our code table.

○ public void encodeFile(Map<Integer,String> code_table):-

■ I have used this function to generate the encoding.bin file for our input message.

fourWayHeap.java

This class is used for implementing the four way cache optimized heap. The following functions have been implemented in this class to obtain the functionality of the four way cache optimized heap.

○ public int getParent():-

■ returns the parent for the object.

○public int getChildren():-

■ returns the children for the current parent object.

○ public void downHeapify():-

■ Used to re adjust the heap after the node is deleted.

○ public void upHeapify():-

■ Used to re adjust the heap after the node is inserted.

○ public void deleteMin():-

■ Used to delete the minimum element in the heap, the minimum element is always the root.

○ public void isEmpty():-

■ to check if the heap is empty or not.

○ public void encodeTree():-

■ Used to traverse the heap structure and to assign the code values while traversing the heap structure.

○ public int peek():-

■ implemented to return the root of our four way cache optimized heap.

huffmanCodeGeneration.java

This is the main class that is called by our encoder class to perform the functionality. This class check the performance of all three heaps to find out which one is the fastest for creation of our Huffman tree. After determining which data structure is the fastest, it performs the encoding operation using the corresponding data structure by making appropriate function calls.

pairingHeapNode.java

This class contains the implementation details for our pairing heap. The following properties have been set in this class,

- This class contains the structure of a node and the getter/setter function for its properties.

- Properties of a node are following:-
    - data:- Data present in the node.
    - leftchild :- the left child for our object.
    - nextSibling : it contains the information about the siblings of the node.
    - previous :- pointer to the previous node.

Apart from these attributes, I have implemented their getters and setters.

pairingHeap.java

This class is used for implementing the pairing heap. The following functions have been implemented in this class to obtain the functionality of the pairing heap.

○ public int getParent():-

■ returns the parent for the object.

○public int getChildren():-

■ returns the children for the current parent object.

○ public void meld():

■ Used to re adjust the heap after the node is deleted.

○ public void deleteMin():-

■ Used to delete the minimum element in the heap, the minimum element is always the root.

○ public void isEmpty():-

■ to check if the heap is empty or not.

○ public void encodeTree():-

■ Used to traverse the heap structure and to assign the code values while traversing the heap structure.

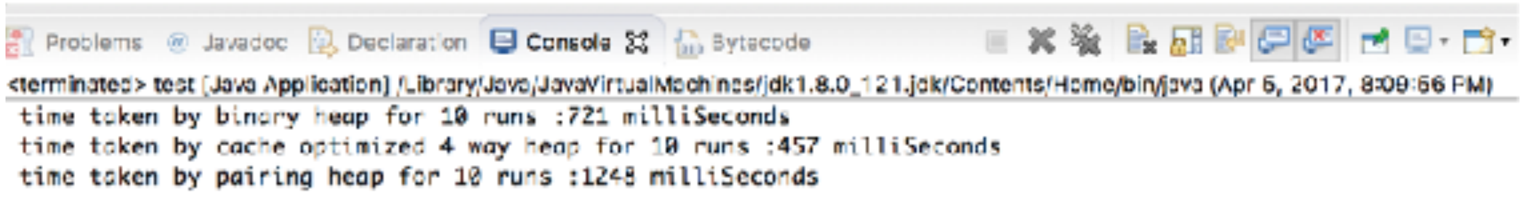
○ public int peek():-

■ implemented to return the root of our four way cache optimized heap.

ANALYSIS

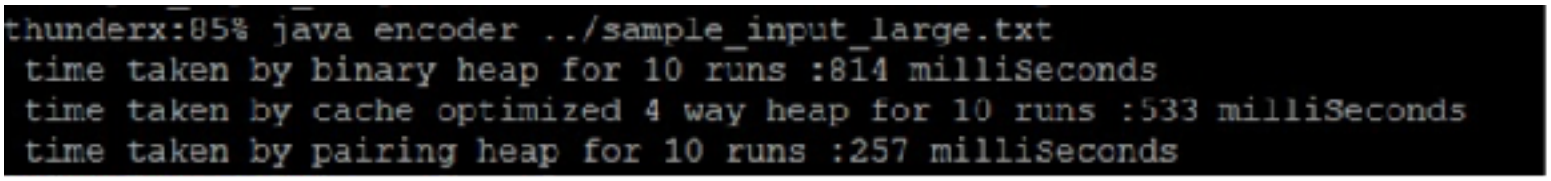
After running all the three heaps for 10 runs on the sample_input_large.txt file, I can conclude that four way cache optimized heap is the fastest on the system on which I did the algorithm, but the performance of pairing heap was better when I ran my code on the CISE Servers. So I have included the screenshots for both the cases :

Times taken on my Machine for the sample input given to us.



```
<terminated> test [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/Contents/Home/bin/java (Apr 6, 2017, 8:09:56 PM)
 time taken by binary heap for 10 runs :721 milliSeconds
 time taken by cache optimized 4 way heap for 10 runs :457 milliSeconds
 time taken by pairing heap for 10 runs :1248 milliSeconds
```

Time taken on the thunder server



```
thunderx:85% java encoder ../sample_input_large.txt
 time taken by binary heap for 10 runs :814 milliSeconds
 time taken by cache optimized 4 way heap for 10 runs :533 milliSeconds
 time taken by pairing heap for 10 runs :257 milliSeconds
```

Also I will like to say that I ran my code on 100 million input data at least 10 times and found that there is a close competition between 4 way cache optimized heap and pairing heap in terms of the time difference. According to me, this makes sense also as if we implement the deletion in pairing heap using the two pass scheme which is observed to be better than the multi pass scheme as the professor discussed in the class.

DECODING ALGORITHM

I have used the following algorithm for our decoding function.

1. First we read the code_table.txt.
2. The Huffman codes have a unique property that no Huffman code will ever be a prefix of the other code, therefore the relevant data we require will be in our leaf nodes of the corresponding Huffman tree. Now if we start to unroll the bit code for each integer in the Huffman tree, no code will match or overwrite the previous code we have. Hence we can create a decode tree from the codes we get from the code_table.txt file.
3. Then I have read the information contained in the encoded.bin to get a byte array to store this information from the encoded.bin.
4. Since from the third step I get a byte array, so i will have to mask the bits to get all the 8 bits and then I used these bits to traverse the decode tree. While traversing the tree, whenever i reach a leaf node, write the data to the output file called the output.txt and traverse the tree again until the end of file is reached.

Complexity of the Algorithm:

The complexity of the decoding algorithm will be O(nb) where b is the number of bits that are used to encode the number and n is the number of elements we have in our code table. The factor of nb is because we would be traversing the decoded tree n times once for each element of our code table and the max length of the tree that we would traverse will be b which we can say as the max length of the encoded message in our code table.