Huffman Encoder and Decoder

Harsh Shah
UFID-1847 4916
harshyshah@ufl.edu

COP 5536 Spring 2017

Harsh Shah – 18474916

COP 5536 Spring 2017

Harsh Shah – 18474916

File Structure:

The project consists of the following java files:

- HeapNode.java
- Heap.java
- BinaryHeap.java
- FourWayHeap.java
- PairingHeap.java
- HuffmanTree.java
- encoder.java
- decoder.java

Introduction

Huffman Tree is a technique for lossless compression of data, to be sent across a network. The sender encodes the data using this tree and the receiver decodes the encoded file using a code table. The idea is to assign shorter length bits to high frequency data. Thus, the tree generation algorithm is a greedy algorithm.

COP 5536 Spring 2017

Harsh Shah – 18474916

Priority Queues

As the Huffman tree generation algorithm requires the greedy approach of extracting minimum frequency elements, we implement this using a priority queue, which is implemented using a Min heap. We will implement three types of Min Heaps – Binary Heap, 4-way Heap and Pairing Heap, and choose the one with the fastest runtime.

Binary Heap

Binary Heap is an almost complete binary tree as it makes sure that levels are filled except possibly the last level. As the name suggests, every node has maximum of 2 children. The operations include – insert(), removeMin(), Heap_DecreaseKey () and MIN_HEAPIFY(). All of the above operations take O(log n) time. The operation build_MinHeap() takes O(n) time.
Following is the function prototype:

```
public class BinaryHeap implements Heap {

An array of HeapNodes
HeapNode[] hn;
int size;

A function to return the current size of the heap
public int getSize() {}

A function to bubble down the violation at index i
public void MIN_HEAPIFY (int i) {}

A function to build the heap using the frequency table
public void build_MinHeap (HashMap<Integer,Integer> freqTable) {}

A function to decrease the key of an element at index i, and then bubble up the violation, if any
public void Heap_DecreaseKey (int i,int key){}

A function to insert a record and bubble up the violation using DecreaseKey operation
public void insert (int data, int freq, HeapNode left, HeapNode right){}

A function to remove the minimum element from the heap, by replacing it with the last element
and then calling MIN_HEAPIFY to eliminate the violations.
public HeapNode removeMin () {}
}
```

4-way Cache Optimized Heap

This is a special case of d-ary heap. It works exactly like Binary Heap, except that every node has at most 4 children. The cache optimization works in a way such that all the siblings are brought into the cache together which reduces the number of cache misses to (log n/log 4) for a removeMin () operation.

The functions in 4-way Heap are similar to Binary Heap with minor modifications like the parent is not child/2.

COP 5536 Spring 2017

Harsh Shah – 18474916

Following is the signature of 4-way Heap.

```
public class FourWayHeap implements Heap {
HeapNode[] hn;

The displacement is used for cache optimization. In my case, it is set to 3.
int size, displacement;

public FourWayHeap(int capacity) {}

A function to find the parent of element i
public int parent(int i) {}

A function to find the ith child of element j
public int child (int i, int j) {}

The remaining functions are exactly like Binary Heap
}
```

Pairing Heap

A pairing heap node consists of a child pointer and leftSibling and rightSibling pointer in addition to storing data and frequency. Practically, pairing heap is faster than Fibonacci heap and is easier to implement. It has an actual complexity of O(n) for both removeMin () and DecreaseKey () but and amortized complexity of O(log n). The most important operation is meld which has an actual complexity of O(1). Following is the signature of Pairing Heap,

COP 5536 Spring 2017

Harsh Shah – 18474916

public class PairingHeap **implements** Heap {
HeapNode root;

public PairingHeap () {}

A function to meld two heaps. It compares the roots of the heaps and makes the root with higher frequency, the left child of other.
private HeapNode meld (HeapNode root1, HeapNode root2) {}

A function to insert a node in the heap, which in turn calls meld. The function is overloaded to provide more flexibility.
public void insert (HeapNode node) {}
public void insert (**int** data, **int** freq, HeapNode left, HeapNode right) {}

A function to remove the minimum element and melding the children using the Multi-Pass scheme, to generate a heap with a new root.
public HeapNode removeMin () {}

A function to build the heap using n inserts where n is the number of distinct keys.
public void build_MinHeap (HashMap<Integer,Integer> freqTable) {}

A function to get the size of the heap either 2 or 0.
public int getSize () {}

Performance analysis

To analyze the performance of each heap implementation, each heap was run 10 times to generate the code table using the frequency table and the following results were obtained.

Heap	Small input (nanoseconds)	Large input (seconds)
Binary Heap	157055.9	1.417
4-way Heap	101333.2	1.341
Pairing Heap	334549.1	4.7

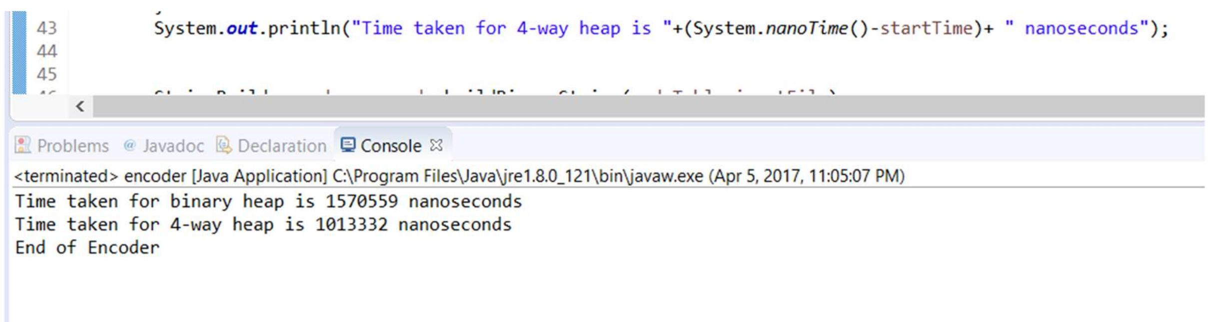
COP 5536 Spring 2017

Harsh Shah – 18474916

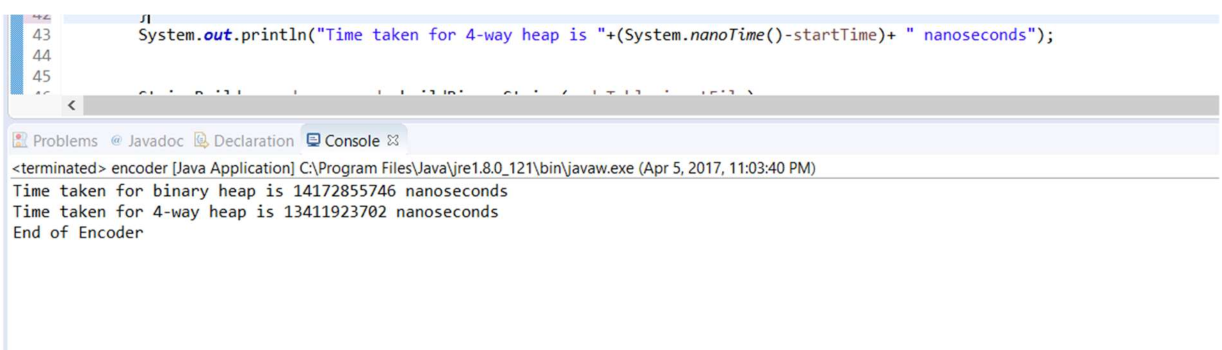
As seen from the experimental results, 4-way heap turned out to be the fastest in all scenarios i.e. small or large input. The reason being cache optimization. Since, the cache size is 64 bytes and each HeapNode is 8 bytes, we displaced the heap by 3 positions so that all the siblings are brought into the cache together always. This reduces the number of cache misses as mentioned above.

Pairing Heap turned out to be the slowest amongst all three implementations for all types of inputs which was a surprise because theoretically it should be faster. On generating Huffman tree using the heap implementations, the following results were found:

Pairing Heap turned out to be slow because of the removeMin () operations which have an actual complexity of O(n).



Small input (10 times)



Large Input (10 times)

COP 5536 Spring 2017

Harsh Shah – 18474916

Huffman Tree

As seen above, since 4-way cache optimized heap turned out to be the fastest, we use it to build the Huffman Tree.

Algorithm:

We use a Greedy algorithm to build the Huffman Tree as mentioned above.

- RemoveMin and store it in a temporary HeapNode temp1
- getMin and store it in another HeapNode temp2 (Note that we are just getting the minimum element and not extracting it, thus it is O(1) operation)
- Create a new node and set its frequency to temp1.frequency + temp2.frequency, and set its left and right pointers to temp1 and temp2.
- Make this new node the root of the heap, and call MIN_HEAPIFY to remove any violation caused.

The time complexity is 2*log n, as removeMin and MIN_HEAPIFY functions are called.

Building Code Table

Once the Huffman tree is built, we use the tree to generate the HashMap Code Table using the function buildCodeTable (). We keep appending 0's when we traverse left and append 1's when we traverse right until we reach the leaf node. Once the leaf node is reached, we copy the data of the node and the generated string to the code table. The signature of the function is shown below:

```
public static void buildCodeTable(HeapNode hn, HashMap<Integer,String> hm,   String s){

If the HeapNode is null, do nothing
Else check if leaf node is reached. If it is reached, put hn.data and String s into the code Table
	Else recursively call buildCodeTable on both hn.left and hn.right by appending 0 and 1 respectively.

}
```

COP 5536 Spring 2017

Harsh Shah – 18474916

Building encoded.bin

We create a binary string by iterating over the input using code table to encode the input data. Once the string is created, we need to write it to encoded.bin. We make use of BitSet for this purpose. The function buildEncodedFile () is used to create the file. It works as shown:

```
public void buildEncodedFile (String code, String filename) {

Create a HashSet set all the values by iterating over the String code.
Convert the HashSet to a byte Array using BitSet.toByteArray()
Write the byteArray to encoded.bin using FileOutputStream.write(byteArray)
Close the FileOutputStream

}
```

Decoding

The decoder takes two files as input – encoded.bin and code_table.txt and gives decoded.txt as output.

Algorithm

First, we create the Decode Tree using Code Table using the following function.

```
public HeapNode buildDecodeTree(HashMap<Integer,String> codeTable, HeapNode hn)
{
For each key in codeTable.keyset, do
String code = codeTable.get (key)
    -   For each character in String code, do
            1. If the node hn is null, create a node and set data to -1
            2. If the character is 0, create a node to left of hn and set hn to hn.left
            3. If the character is 1, create a node to right of hn and set hn to hn.right
    -   Set the data of the leaf node to the key and set hn to root again.
}
```

Now we, build the decoded file using the encoded.bin file and the decodeTree. The encoded.bin file read using BitSet again and converted into a String of 0’s and 1’s.

Time Complexity

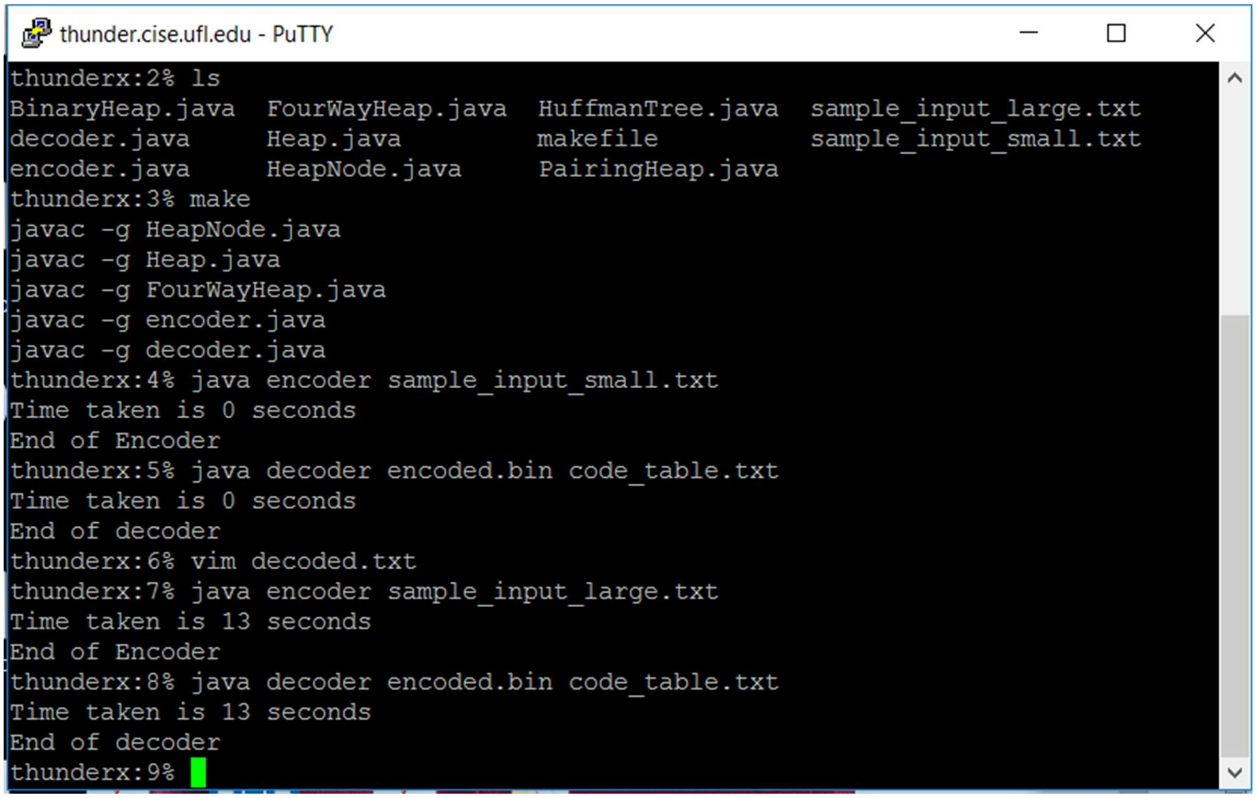
If the number of different keys are N and the maximum code length is L, the time complexity is O(NL). However, not all codes are of maximum length, so the actual complexity is sum of all code lengths in the HashMap.

Building Decoded.txt

We make use of BufferedWriter to write the StringBuilder created using encoded.bin and the decodeTree. The total time taken by the encoder and decoder both for large input is 13 seconds as shown below.

COP 5536 Spring 2017

Harsh Shah – 18474916



References

1. Lecture Slides
2. Introduction to Algorithms is a book by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.