

Report of Course Project (COP 5536)

NAME: Xiyao Ma UFID:14311282

I. Introduction

In my course project, I wrote a software in java to encode and decode txt file with 4-way min heap and Huffman tree. In Chapter 2, I will illustrate the structure of my project and explain the idea behind some critical function. In Chapter 3, I will analysis and compare the performance of binary heap, 4-way heap, and pairing heap, when building the Huffman tree with them.

II. Structure

In this chapter, I would introduce the structure of the whole project and basic function of every java file.

Arralist_node.java and pairing_heap.java create their own node structure for the three kind of data structures. Binary_heap.java, four-way-heap.java, pairing_heap.java has some function to process the three kind of data structures. Huffman.java provide some function to build a Huffman tree with the three kind of data structure. Test.java build Huffman tree with each kind of data structure for 10 times, and compare the time and decide which one data structure is most fast and appropriate for me to build a Huffman tree, encode, and decode. Then encoder.java can read an input txt file to build a Huffman tree with 4-way heap. It also creates a code table and write in code_table.txt file, and encode the input txt file and write the encoded code into binary encoded.bin file. In the last, decoder.java read code_table.txt to create a decode tree and decode encoded.bin with it, and write the encoded string to decoded.txt. The structure of the whole project is shown in Fig. 1.

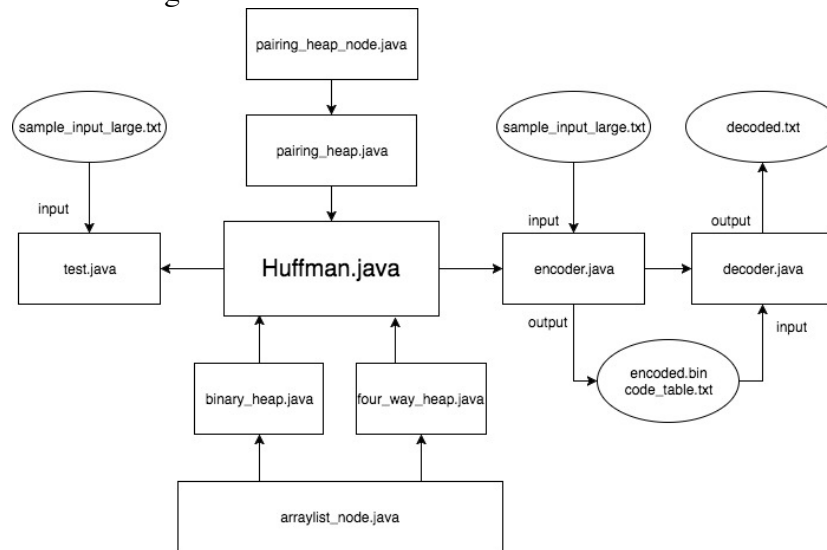


Fig. 1. Structure of project

Detailed illustration of classes and functions are as follows:

In arraylist_node.java, I create a class called arraylist_node, and it has key(int), value(String), liftchild(arraylist_node), rightchild(arraylist_node), and isleaf(boolean).

Binary_heap.java and four_way_heap.java create their own arraylist with the node structure created in arraylist_node.java. They have their own function such as heapsort, insert, removemin, bubbleup, bubbledown, etc, all of which will be used in Huffman.java.

Pairing_heap.java create another arraylist with the node structure in pairing_heap_node.java. It also has some functions that are similar to the functions in binary_heap.java and four_way_heap.java, like heapsort, insert. But it has a special function, called compare_and_meld, which compare the key of the root of two tree, and make the tree with smaller key root subtree of the other tree. In addition, it has another complex function, called remove_min. I write the pseudocode of it as follows:

```
Public void remove_nim(){
    if (only one child) firstnode = root.child;
    num = get_child_num(root);
    create a array and put all the children in it;
    // two pass merge
    for (int i = 0; i < num_child/2; i++) {
        nodeSet[i] = compare_and_meld(nodeSet[2 * i], nodeSet[2 * i + 1]);
    }
    for(int i = num_child/2 - 1; i > 0; i--) {
        nodeSet[i - 1] = compare_and_meld(nodeSet[i], nodeSet[i - 1]);
    }
    if (num_child % 2 == 1) {
        for (int i = 0; i < num_child/2; i++) {
            nodeSet[i] = compare_and_meld(nodeSet[2 * i], nodeSet[2 * i + 1]);
        }
        nodeSet[num_child / 2] = nodeSet[num_child - 1];
        for (int i = num_child / 2; i > 0; i--) {
            nodeSet[i - 1] = compare_and_meld(nodeSet[i], nodeSet[i - 1]);
        }
    }
}
```

Huffman.java has several functions that are used to build a Huffman tree. Functions, build_huffman_binaryheap, build_huffman_4wayheap, and build_huffman_pairingheap are three function to build a Huffman tree with each of the three kinds of data structure. The **procedure of building a Huffman tree** is remove the two nodes with smallest key(frequency), combine them and insert the new combined node in the heap. To avoid building a Huffman tree with too much height, I made some change to bubbledown function in binary_heap.java, four_way_heap.java and pairing_heap.java. Here I take the **bubbledown** function in four_way_heap.java as example, first thing is to pick the child with smaller key of the root, and if there are some children with same smallest key, I will pick the child if it is a leaf node, if not, picking any node is OK. In this case, I can build a Huffman tree with a not too much height. The pseudocode is as follows:

```
Public void bubbledown(){
```

```

        Default, first child is the smallest child;
        If
        (child2.key>smallerchild.key||(child2.key==smallerchild.key&&child2.isleaf)){
            Smallerchild =child2;
            .....
        }
        .....
    }
}

```

Test.java read the input txt file and build Huffman trees with each of the three kind of data structure for 10 times, and compare the time and decide which one data structure is most fast and appropriate for me to build a Huffman tree, encode, and decode.

Encoder.java has two functions, build_code_table and encode. The former one will create a code table with the Huffman tree built with 4-way heap, if and put the value and code in an array. This function is a recursive function, if the current node is a left child of parent, I add a “0” to the code, on the contrary, if it is a right child of parent, I add a “1” to the code. Here I was used to try to put them in a hash map, but when I run ran the code, there was a StackOverflow error. Although array will take a lot of space, but it is faster, and most importantly, it works. Then I use encode function to read the input file and encode it, first I read every line of the input file and find its according code in code table array, and append the code into a stringbuffer. Then, every time I pick 1 byte(8 bits) and convert it to binary and write it to encoded.bin. The code and pseucode are as follows:

```

public void build_code_table(arraylist_node x, String code) {
    if (x.isleaf == false) {
        build_code_table(x.leftchild, code + '0');
        build_code_table(x.rightchild, code + '1');
    } else {
        code_table[Integer.parseInt(x.value)] = code;
    }
}

public void encode(input_file){
    while(line = line of input file){
        stringbuffer.append(line);
    }
    if (code_table[line ]){
        value = code_table[line];
        output value's binary
    }
}

```

Decoder.java has several functions to read the binary bin file, build decode tree and decode the input txt fiel with it. Readcode function is used to read a binary file and convert it to a string. In build_decode_tree function, first, I read every line of code_table.txt and split every line

with “ ” and put the value into array[0] and put the code into array[1]. Then, as for every code sequence, I read every char in this sequence. If the char is a “0”, the node is a left child of the current node, while if the char is a “1”, it is a right child of the current node. Eventually, I build a decode tree. In decode function, I first use readcode function to read encoded.bin, and get the string containing all codes. Then I check every char of the string from left to right, if the char is a “0”, the node is a left child of the current node, while if the char is a “1”, it is a right child of the current node. Everytime we will check if the node were a leaf node, if it is, then take the key of the current node and put the key into a buffer, and updata the current node to rootnode. if not, continue to check. Evenrually, I write the content of stringbuffer to decoded.txt. All done. The pseudocode is as follows:

```

    Public void build_decode_tree(){
        Read every line of code_table.txt;
        Split it into array[0] and array[1] with “ ”;
        Put the value and code in code_table array;
        While(code_table[value]){
            For (char in code_table[value]){
                If (char is “0”) node = node.leafchild;
                Else node = node.rightchild;
                When the char is the last char of the code, make the node a
leaf node;
            }
        }
    }

    public decode(){
        string = readcode(“encoded.txt”);
        for(i = 0;I < stirng.length(); i++){
            if (char is “o”) node= node.leftchild;
            else node = node.rightchild;
            if the node is a leaf node, put the value into stringbuffer;
            update node to root_node;
            if not, continue;
        }
    }

```

To compile my code in terminal, I write a make file:

```

hashtagcounter: arraylist_node.java binary_heap.java four_way_heap.java
pairing_heap_node.java pairing_heap.java huffman.java test.java encoder.java decoder.java
    javac arraylist_node.java binary_heap.java four_way_heap.java pairing_heap_node.java
pairing_heap.java huffman.java test.java encoder.java decoder.java

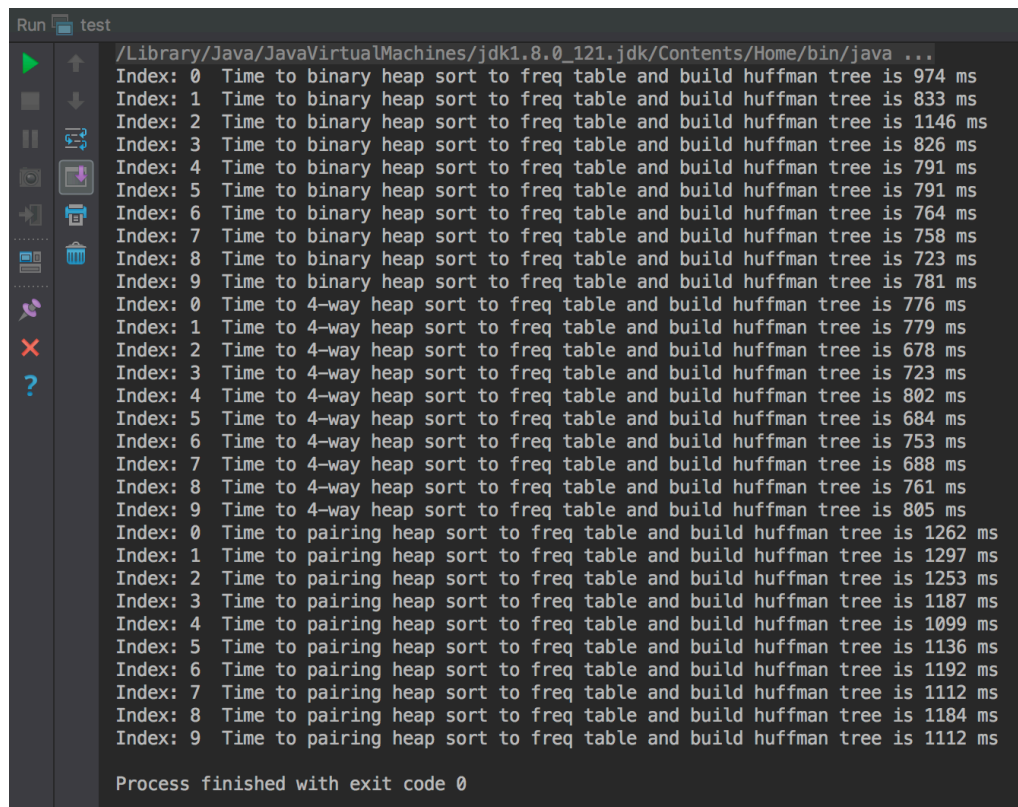
```

clean:

```
rm*.class
```

III. Analysis performance

When I use test.java to test the performance of each data structure when building Huffman tree with them. The time they cost is shown in Fig. 2.



```
Run test
/Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/Contents/Home/bin/java ...
Index: 0 Time to binary heap sort to freq table and build huffman tree is 974 ms
Index: 1 Time to binary heap sort to freq table and build huffman tree is 833 ms
Index: 2 Time to binary heap sort to freq table and build huffman tree is 1146 ms
Index: 3 Time to binary heap sort to freq table and build huffman tree is 826 ms
Index: 4 Time to binary heap sort to freq table and build huffman tree is 791 ms
Index: 5 Time to binary heap sort to freq table and build huffman tree is 791 ms
Index: 6 Time to binary heap sort to freq table and build huffman tree is 764 ms
Index: 7 Time to binary heap sort to freq table and build huffman tree is 758 ms
Index: 8 Time to binary heap sort to freq table and build huffman tree is 723 ms
Index: 9 Time to binary heap sort to freq table and build huffman tree is 781 ms
Index: 0 Time to 4-way heap sort to freq table and build huffman tree is 776 ms
Index: 1 Time to 4-way heap sort to freq table and build huffman tree is 779 ms
Index: 2 Time to 4-way heap sort to freq table and build huffman tree is 678 ms
Index: 3 Time to 4-way heap sort to freq table and build huffman tree is 723 ms
Index: 4 Time to 4-way heap sort to freq table and build huffman tree is 802 ms
Index: 5 Time to 4-way heap sort to freq table and build huffman tree is 684 ms
Index: 6 Time to 4-way heap sort to freq table and build huffman tree is 753 ms
Index: 7 Time to 4-way heap sort to freq table and build huffman tree is 688 ms
Index: 8 Time to 4-way heap sort to freq table and build huffman tree is 761 ms
Index: 9 Time to 4-way heap sort to freq table and build huffman tree is 805 ms
Index: 0 Time to pairing heap sort to freq table and build huffman tree is 1262 ms
Index: 1 Time to pairing heap sort to freq table and build huffman tree is 1297 ms
Index: 2 Time to pairing heap sort to freq table and build huffman tree is 1253 ms
Index: 3 Time to pairing heap sort to freq table and build huffman tree is 1187 ms
Index: 4 Time to pairing heap sort to freq table and build huffman tree is 1099 ms
Index: 5 Time to pairing heap sort to freq table and build huffman tree is 1136 ms
Index: 6 Time to pairing heap sort to freq table and build huffman tree is 1192 ms
Index: 7 Time to pairing heap sort to freq table and build huffman tree is 1112 ms
Index: 8 Time to pairing heap sort to freq table and build huffman tree is 1184 ms
Index: 9 Time to pairing heap sort to freq table and build huffman tree is 1112 ms
Process finished with exit code 0
```

Fig.2. cost time

Performance analysis: Based on the cost time shown above, 4-way heap is the best data structure among the three for me to build a Huffman tree. In my opinion, the reason that the 4-way heap is faster than binary node is 4-way node compare 4 nodes at one time rather than 2. Meanwhile, the reason that pairing heap takes so much time is that every time remove min element from pairing heap, it need two pass merge, which takes a lot of time.

Decoding algorithm analysis: In decoder.java, the first thing I do is to create a decode tree from code_table.txt. In decoding part, we check exactly every bit in encoded.bin file and find if its according value in the code table. If we define the size of the input file n, decoing complexity is $O(n)$.

Encoding and Decoding performance analysis: In terminal, access the code file, and make as shown in Fig. 3.

```
$ make
```

```
Ma_Xiyao — -bash — 80x24
Last login: Wed Apr  5 10:52:51 on ttys000
[Xiyaos-MacBook-Pro:~ xiyaoma$ cd /Users/xiyaoma/Dropbox/Courses/2017_Spring/ADS/]
project/Ma_Xiyao
[Xiyaos-MacBook-Pro:Ma_Xiyao xiyaoma$ ls ]
Report of Course Project.docx  huffman.java
Report of Course Project.pdf   makefile
arraylist_node.java           pairing_heap.java
binary_heap.java              pairing_heap_node.java
decoder.java                   test.java
encoder.java                   ~$port of Course Project.docx
four_way_heap.java
[Xiyaos-MacBook-Pro:Ma_Xiyao xiyaoma$ make ]
javac arraylist_node.java binary_heap.java four_way_heap.java pairing_heap_node.
java pairing_heap.java huffman.java test.java encoder.java decoder.java
[Xiyaos-MacBook-Pro:Ma_Xiyao xiyaoma$ ]
```

Fig. 3. make

```
$ java encoder sample_input_large.txt
[Xiyaos-MacBook-Pro:Ma_Xiyao xiyaoma$ java encoder sample_input_large.txt ]
Time to build huffman tree with 4-way heap is 618 ms
Time to build code table with 4-way heap is 379 ms
Encoding complete.
Time to encode with 4-way heap is 7217 ms
[Xiyaos-MacBook-Pro:Ma_Xiyao xiyaoma$ ]
```

Fig. 4 encoding

```
$ java decoder encoded.bin decoded.txt
[Xiyaos-MacBook-Pro:Ma_Xiyao xiyaoma$ java decoder encoded.bin code_table.txt ]
Time to decode with 4-way heap is 8659 ms
[Xiyaos-MacBook-Pro:Ma_Xiyao xiyaoma$ ]
```

Fig. 5. decoding

The encoded.bin file produced by encoder has the same size as the encoded.bin provided by Teacher's sample, as shown in Fig. 6.

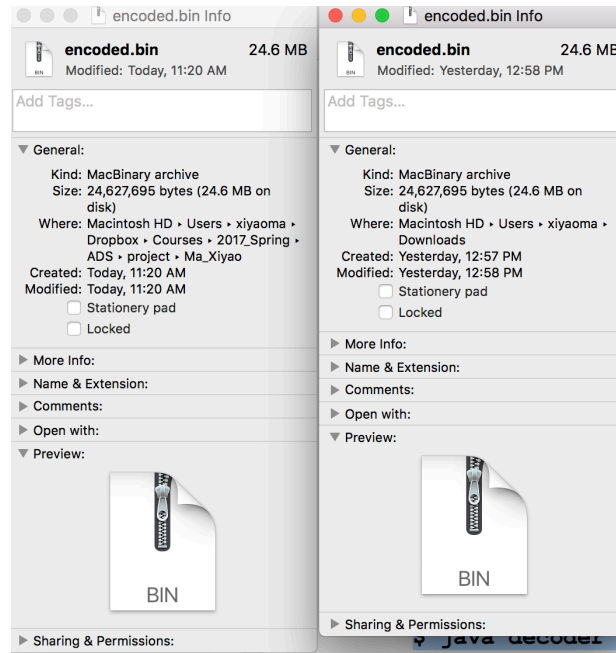


Fig. 6. The size of encoded.bin

Decoded.txt file has the same size as sample_input_large.txt, as shown in Fig. 7.

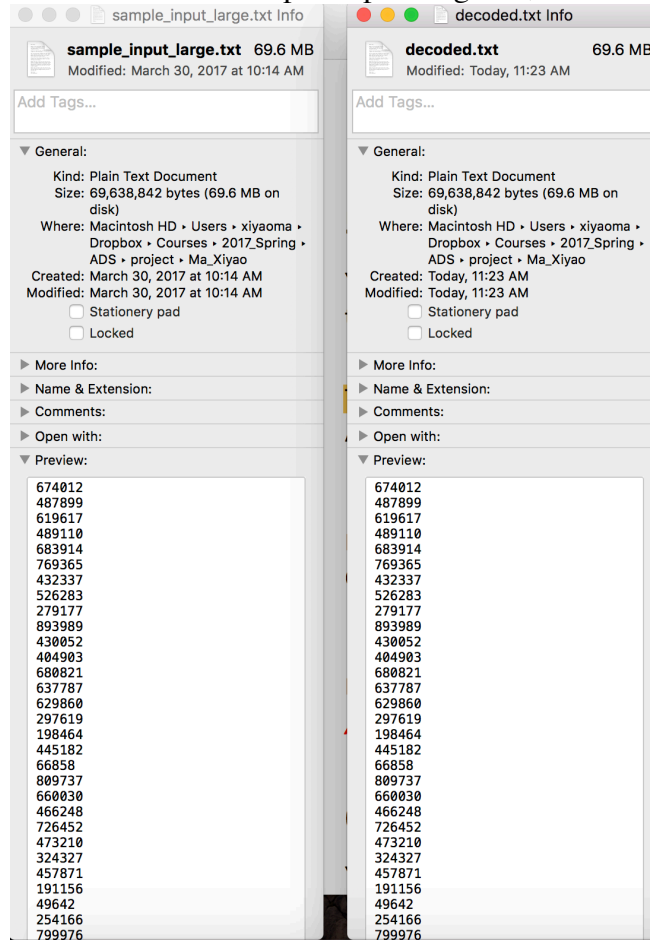


Fig. 8. The size of decoded.txt and sample_input_large.txt are same

As I said above, my code works correctly.