Pu Fang
61871845

Tips: The program is based on java using eclipse.

Functions

Heap.java

```
public abstract class Heap {
    protected int count;
    protected int size;
    Heap(int size){
    public int getSize(){
    public abstract node peek();
    public abstract node removeMin();
    public abstract void add(node newnode);
    public abstract void clear();
}
```

This is an abstract class extended by three heaps with the aim of reusing the code of encoding.

Variables:

count: Keep track of the current size of the Heap.

size: Total size of the heap.

Functions:

Heap(int size): Initialize count and set size of heap according to the size of fre_table.

getSize(): return count.

peek(): get the first element.

removeMin(): remove the min element

add(): add a new element to heap.

clear(): clear all data

node.java

```
public class node {
    int freq;
    int text;
    node left;
    node right;
    node(int freq, int text){
        this.freq = freq;
        this.text = text;
        left = null;
        right = null;
    }
}
```

Basic structure to store information and also be used to build Huffman tree.

BinaryHeap.java

```
public class BinaryHeap extends Heap {

    public node[] nodes;
    BinaryHeap(int size){
    public node peek(){
    public void add(node newnode){
    public node removeMin(){
    private void shiftdown(int k) {
    private void shiftup(int k) {
    private void swap(int a, int b){
    public void clear() {
}
```

This is a realization of Binary Heap using array.

Variables:

nodes: heap space.

Functions:

shiftdown(int k): After removing min element and make the last element at the top of binary heap, this function adjusts this element.

shiftup(int k): after adding a new element, this function adjust the element.

swap(int a, int b): swap two elements in the heap.

Other functions are realized according to Heap.

PairingHeap.java

```
class heapNode{
    node nd;
    heapNode left;
    heapNode leftchild;
    heapNode rightsibling;
    heapNode(node nd){
}
public class PairingHeap extends Heap {
    public heapNode root;

    PairingHeap(int size){
    public node peek(){
    public void add(node newnode){
    public node removeMin(){
    public heapNode meld(heapNode silbing){  //two pass
    public heapNode combine2(heapNode a, heapNode b){
    public void clear() {
}
```

This is a realization of Pairing Heap using doubly linked list.

Variables:

root: min element.

Functions:

meld(heapNode silbing): combine all nodes after removing min, using two pass scheme.

combine2(heapNode a, heapNode b): combine two nodes.

Other functions are realized according to Heap.

FourwayHeap.java

```
public class FourwayHeap extends Heap{

    public node[] nodes;
    FourwayHeap(int size){
        super(size);
        count = 3;
        nodes = new node[size+3];
    }
    public int getSize(){
    public node peek(){
    public void add(node newnode){
    public node removeMin(){
    private void shiftdown(int k) {
    private void shiftup(int k) {
    private void swap(int a, int b){
    public void clear() {
}
```

This is a realization of Pairing Heap using doubly linked list.

Variables:

nodes: heap space.

Functions:

same functions with Binary heap. But I shift the whole array 3 indices to left in order to get cache optimization. The first node will be at 3 and the children of any nodes will be cache aligned by four.

encoder.java

```
public class encoder {
    public static Map<Integer, Integer> fre_table;
    public static Map<String, String> code_table;
    private static Heap heap;
    private static int heapkind;
    public static String filename;
    encoder(int heapkind){
    public void readData() throws IOException{
    public void setHeap(int heapkind){
    public node buildTree(){
    public void encode(node root) throws IOException{
    public static void dfs(node root, String huff){
    public double eval() throws IOException{
    public static void main(String[] args) throws IOException{
}
```

Variables:

fre_table: record the kinds of numbers and count their frequency.

code_table: code_table according to fre_table.

heap: the heap that used by encoder.

heapkind: determine which kind of heap being used.

filename: input file.

Functions:

readData(): build fre_table.

setHeap(int heapKind): determine the kind of heap.

buildTree(): build Huffman tree.

encode(): the process of encoding.

dfs(node root, String huff): this program builds code_table using dfs.

eval(): for evaluating the performance of three heaps.

decoder.java

```java
public class decoder {
    private node root;
    decoder(){
    public void buildLeaf(String str){
    public boolean isLeaf(node root){
    public void buildTree(String filename) throws IOException{
    public void decode(String filename) throws IOException{
    public static void main(String[] args) throws IOException{
}
```

Variables:

root: root node for Huffman decoding tree.

Functions:

buildLeaf(String str): building one path from root to node according to input code.

isLeaf(node root): judge whether a node is leaf.

buildTree(String filename): build tree with inputfile

decode(String filename): process of decoding.

evaluation.java

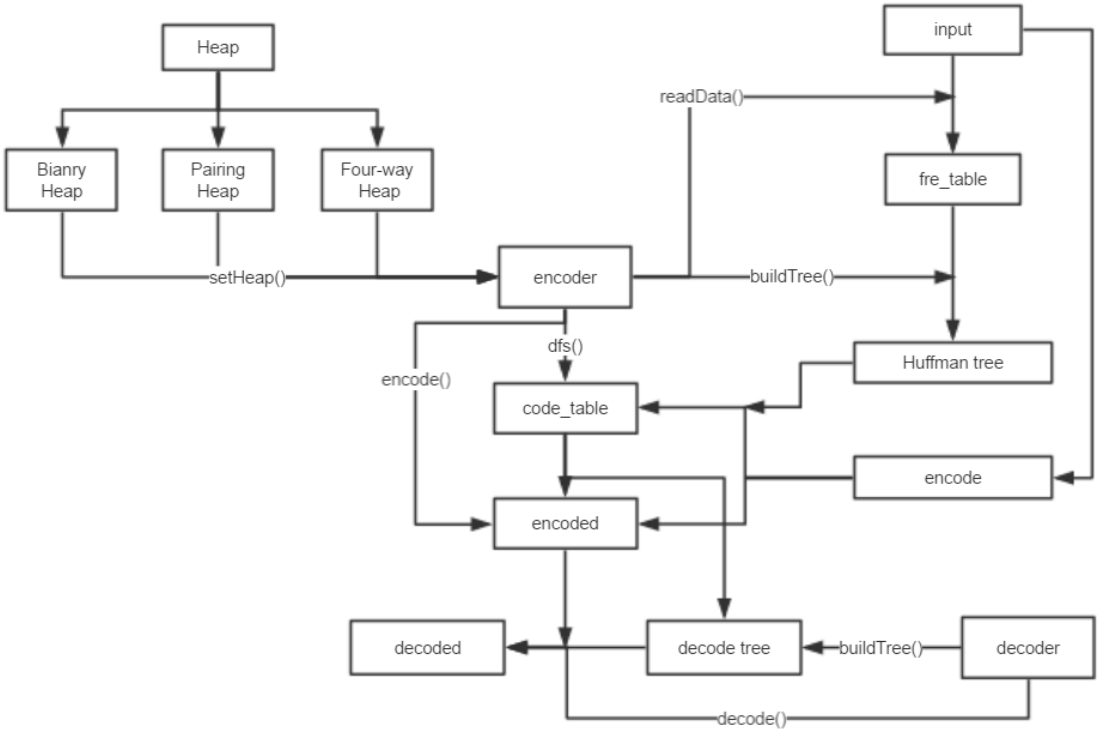
```java
public class evaluation {

    public static void main(String[] args) throws IOException {

}
```

This is for performance analysis.

Structure



Performance Analysis

Binary Heap: 465.6 ms
Pairing Heap: 518.8 ms
Four-way Heap: 425.8 ms

We use cache optimization for 4-way heap, which reduces the number of cache miss. So 4-way heap has the best performance. And it also has fewer level.
Pairing heap uses pointer and it means that the space of heap will be very discrete. So generally, there're more cache miss than array-based heap. And the actual cost of removeMin is O(n), without optimization, insertion sequence and data scale of siblings may leave great influence on time. Therefore, pairing heap comes to the end.

Decoding Algorithm

1. Initialize a root, if we go left, append a "0" to the value of root and give it to the left node, if we go right, append a "1" to the value of root and give it to the right node.
2. For each entry in code_table, we read every bit from the value, and if there's no node when reading, we build a new node and give value according to (1). Apparently, the last bit will become a leaf in the tree. In other word, we build a path from root to leaf for every entry.
3. Read every bit from the encoded data, if it's 0, we move left, otherwise, we move right. After moving, if we encounter a leaf, we decode the previous bits, start a new path from root and continue reading. If not, we continue reading until we find a leaf or reach the end of file.

The time cost mainly from two parts: building tree and read data.

1) The cost for creating a new node is O(1), when we get an entry, the path of this entry in the tree has the same length with the length of its binary code. Therefore, the number of nodes we visit is same with the length of its binary code. So if we have m entries and each entry has ni (i=1, 2, 3…, m) length binary code. The time complexity will be $\Theta(\sum_{i=1}^{m} n_i)$.
2) The cost for decoding, when we read one bit from the input, we visit one node and then the next bit. So the total time cost will be the length of input data. Assume that the length is L.

So the total cost will be $\Theta(\sum_{i=1}^{m} n_i + L)$. And if we have very long encoded data but smaller code table. The time cost will be $\Theta(L)$, and if we have very short code with large code table, the time cost will be $\Theta(\sum_{i=1}^{m} n_i)$. The height of huffman tree will be in (logn, n-1), and in most

cases, the average height of leaves will be near logn. So we can also treat it approximately as $\Theta(n \log n + L)$.