COP 5536 Spring 2017 Programming Project Report

Rushikesh Gawali

UFID:8330-4331

Huffman Tree Node Structure:

```
Struct treenode:
        int freq;
        int element;
        treenode *left,*right;
```

Binary Heap Node Structure:

```
        int element;
        int freq;
        treenode *root;
```

4way Heap Node Structure:

```
        int element;
        int freq;
        treenode *root;
```

Pairing Heap Node Structure:

```
         treenode *root;

         int element,freq;

         PairingNode *leftChild;

         PairingNode *nextSibling;

         PairingNode *prev;
```

```
Main()
```

Read contents of input file into a **frequency** array which records the frequency(number of occurrences) for each number in the file.

```
pairing_start_time

for(int xyz = 0;xyz  < 10;xyz++)
        build_tree_using_pairing_heap(frequency array);

pairing_finish_time


binary_start_time

for(int xyz = 0;xyz  < 10;xyz++)
        build_tree_using_binary_heap(frequency array);

binary_finish_time

fourway_start_time

for(int xyz = 0;xyz  < 10;xyz++)
        build_tree_using_4way_heap(frequency array);

fourway_finish_time

if(((binary_finish_time - binary_start_time) < (pairing_finish_time -
pairing_start_time)) && ((binary_finish_time -
        binary_start_time) < (fourway_finish_time - fourway_start_time)))
        {
                Encode using Binary Heap
        }
        else if(((pairing_finish_time - pairing_start_time) <
(binary_finish_time - binary_start_time) ) && ((pairing_finish_time -
pairing_start_time) < (fourway_finish_time - fourway_start_time)))
        {
                Encode using Pairing Heap
        }
        else if(((fourway_finish_time - fourway_start_time) <
(binary_finish_time - binary_start_time) ) && ((fourway_finish_time -
fourway_start_time) < (pairing_finish_time - pairing_start_time)))
        {
                Encode using 4way Heap
        }
```

`build_tree_using_pairing_heap(`**frequency array**`);`

1) for every element whose frequency > 0 in the frequency array, insert a new Huffman tree node in the Huffman tree as well as a Binary heap node in the binary heap which also includes the Huffman tree node just created.
2) Delete 2 nodes from the binary heap with the lowest frequency
3) Add a new Huffman tree node that has the sum of frequencies of the two deleted nodes in the Huffman tree as well as a Binary heap node that has the sum of frequencies of the two deleted nodes in the binary heap which also includes the Huffman tree node just created.
4) The Huffman tree nodes deleted in step 2 become the left and right children of the new Huffman tree node inserted in step 3 above
5) Repeat steps 2,3 & 4 till size of Binary Heap becomes 1.

```
void BinaryHeap::Insert(int element,int freq, treenode *temp)
```
Inserts new node. Steps 1 and 3 above.

```
node BinaryHeap::ExtractMin()
```
Returns node with minimum frequency. Called before Delete Min. (Implicit) Step 2 above

```
void BinaryHeap::DeleteMin()
```
Deletes node with minimum frequency Step 2 above.

```
int left(int parent);
```
Called by heapifydown() to get left child of node.

```
int right(int parent);
```
Called by heapifydown() to get right child of node.

```
int parent(int child);
```
Called by heapifyup() to get parent of node.

```
void heapifyup(heap.size() -1);
```
Called to Honor Binary heap property after every insert.

```
void heapifydown(0);
```
Called to Honor Binary heap property after every DeleteMin.

`build_tree_using_4way_heap(`**frequency array**`);`

1) for every element whose frequency > 0 in the frequency array, insert a new Huffman tree node in the Huffman tree as well as a 4way heap node in the 4way heap which also includes the Huffman tree node just created.
2) Delete 2 nodes from the 4way heap with the lowest frequency
3) Add a new Huffman tree node that has the sum of frequencies of the two deleted nodes in the Huffman tree as well as a 4way heap node that has the sum of frequencies of the two deleted nodes in the 4way heap which also includes the Huffman tree node just created.
4) The Huffman tree nodes deleted in step 2 become the left and right children of the new Huffman tree node inserted in step 3 above
5) Repeat steps 2,3 & 4 till size of 4way Heap becomes 1.

`void fourwayHeap::Insert(int element,int freq, treenode *temp)`
Inserts new node. Steps 1 and 3 above.

`node fourwayHeap::ExtractMin()`
Returns node with minimum frequency. Called before Delete Min. (Implicit) Step 2 above

`void BinaryHeap::DeleteMin()`
Deletes node with minimum frequency Step 2 above.

`int one(int parent);`
Called by heapifydown() to get first child of node.

`int two(int parent);`
Called by heapifydown() to get second child of node.

`int three(int parent);`
Called by heapifydown() to get third child of node.

`int four(int parent);`
Called by heapifydown() to get fourth child of node.

`int findmin(int one,int two,int three,int four);`
Called by heapifydown(). returns child number with lowest frequency.

`int parent(int child);`
Called by heapifyup() to get parent of node.

`void heapifyup(heap.size() -1);`
Called to Honor 4way heap property after every insert.

`void heapifydown(0);`
Called to Honor 4way heap property after every DeleteMin.

`build_tree_using_pairing_heap(`**frequency array**`);`

1) for every element whose frequency > 0 in the frequency array, insert a new Huffman tree node in the Huffman tree as well as a Pairing heap node in the Pairing heap which also includes the Huffman tree node just created.
2) Delete 2 nodes from the Pairing heap with the lowest frequency
3) Add a new Huffman tree node that has the sum of frequencies of the two deleted nodes in the Huffman tree as well as a Pairing heap node that has the sum of frequencies of the two deleted nodes in the Pairing heap which also includes the Huffman tree node just created.
4) The Huffman tree nodes deleted in step 2 become the left and right children of the new Huffman tree node inserted in step 3 above
5) Repeat steps 2,3 & 4 till size of Pairing Heap becomes 1.

```
void PairingHeap::Insert(int freq,int x,treenode *temp)
```
Inserts new node. Steps 1 and 3 above.

```
treenode *PairingHeap::findMin()
```
Returns node with minimum frequency. Called before Delete Min. (Implicit) Step 2 above

```
void PairingHeap::deleteMin()
```
Deletes node with minimum frequency Step 2 above.

```
void PairingHeap::Link(PairingNode * &first, PairingNode *second)
```
Called to Honor Pairing heap property after every insert.
first is root node of Pairing Heap.
second is new node just inserted.

```
PairingNode *PairingHeap::Combine(PairingNode *firstSibling)
```
Called to Honor Pairing heap property after every DeleteMin.
firstSibling is left child of node just deleted(old root).

Encoder:

The algorithm for encoder for the Huffman trees generated by each of the above 3 Heaps is the same.

1) Start at the root of the corresponding Huffman tree.
2) If left &/or right child exists go to left &/or right recursively, till you fall out of the tree.
3) Append 0 to Huffman code if left child. 1 if right child.
4) Terminate when you fall off the tree.
5) Store Huffman code for that particular element in Map.

```
void printcodes(treenode *root, string str)
{


    if (root == NULL)
        return;

    if (root->element >= 0)
        {
              huffman[root->element] = str;
       }
    printcodes(root->left, str + "0");
    printcodes(root->right, str + "1");

}
```

1) Open input file.
2) For each element in input file, write corresponding Huffman code in **encoded.bin**
3) Repeat step 2 till end of file.
4) Write Map to **code_table.txt.**

Decoder:

1. Read **code_table.txt** into a map.
2. Create new treenode root.
3. For every row in map,
   3.1 Set root to root of the Huffman tree.
   3.2 For every bit of the Huffman code, for that particular row,
      3.2.1 If current bit is 0, create left child of root node. Make the left child root. If current bit is one, create right child of root node. Make right child as root.
4. Open **encoded.bin**
5. Set root to root of the Huffman tree.
6. For 0 make the left child of the root as the root.
7. For 1 make the right child of the root as the root.
8. Repeat steps 6 and 7 till you fall off the tree.
9. Write the element to the file **decoded.txt** when you fall off the tree.
10. Repeat steps 5 to 9 till end of file **encoded.bin**

Complexity of Decoding algorithm: O(**nlogn)** where n is number of elements and logn is the height of the Huffman tree.

Performance analysis results and explanation:

build_tree_using_pairing_heap takes approximately 7 seconds for large file(70 mb).

build_tree_using_binary_heap takes approximately 8 seconds for large file(70 mb).

build_tree_using_4way_heap takes approximately 9 seconds for large file(70 mb).

```
rushikesh@rushikesh-VirtualBox:~/Desktop$ ./encoder sample_input_large.txt
Time using pairing heap (microsecond):  6730236
Time using binary heap (microsecond):  7546863
Time using fourway heap (microsecond):  8993330
```

Pairing Heap is fastest since I have implemented it using Linked List, whereas Binary Heap and 4way Heap have been implemented using vector.

During Insert and deletemin using Pairing Heap only pointers of the nodes need to be updated, whereas in Binary Heap and 4way Heap since it has been implemented using vectors, the entire node needs to be copied/swapped. This take longer than just updating the pointers.

4way heap is further slower than binary heap since for every insert/delete there are 4 comparisons, 3 with fellow sibilings and 1 with parent. For binary its just 2 comparisons per insert/delete: 1 with sibling and 1 with parent.