COP 5536 Fall 2020 Programming Project Report

Zhengcheng Song 9455-5960

1. Data structure

Before dig into the algorithms. Implementation of three types pf priority queues and Huffman tree is shown below.

1) Binary heap

node structure:

```
struct node
{
    struct node* left, * right;
    int data;
    int freq;
    treenode* root; //used for built Huffman tree by queue
};
```

Class Member Functions:

```
    vector <node> heap;
```
Use vector to store binary heap.

```
     int left(int parent);
```
Get the left child of a node.

```
    int right(int parent);
```
Get the right child of a node.

```
    int parent(int child);
```
Get the parent of a node.

```
    int Size();
```
Get the size of the heap.

```
    void siftUp (int index);
```
Move up a node to restore heap condition after insertion.

```
    void siftDown(int index);
```

Move down a node down to restore heap condition after deletion.

```
    void insert(node&newNode,treenode*);
```
Insert a new node into the heap.

```
    void deleteMin();
```
Delete the minimum node of the heap.

```
    node extractMin();
```
Get minimum node of the heap.

2) 4-way cache heap

node structure:

```
struct node
{
    struct node* left, * right;
    int data;
    int freq;
    treenode* root; //used for built Huffman tree by queue
};
```

Class Member Functions

```
    vector <node> heap;
```
Use vector to store 4-way heap.

```
    int fisrtChild(int parent);
    int secondChild(int parent);
    int thirdChild(int parent);
    int fourthChild(int parent);
```
Get the four children of the node.

```
    int findMin(int fisrtChild, int secondChild, int thirdChild, int
   fourthChild);
```
Find the minimum child of a node.

```
    int parent(int child);
    int Size();
    void siftUp(int index);
    void siftDown(int index);
    void insert(node& newNode, treenode* temp);
    void deleteMin();
    node extractMin();
```

These functions have the same functionality as Binary heap's functions.

3) Pairing heap

node structure

```
class PairingNode
{

public:

    int data, freq;

    PairingNode* leftChild;

    PairingNode* nextSibling;

    PairingNode* prevSibling;

    treenode* root; //used for built Huffman tree by queue
};
```

Class Member Functions

```
    PairingNode* root;
```
Record the root of pairing heap.

```
    void meld(PairingNode*& first, PairingNode* second);
```
Meld the newly created heap and the existing heap after insertion.

```
    PairingNode* merge(PairingNode* firstSibling);
```
Repeat meld subtrees until one tree remain after deleting the minimum node(root).

```
    int size
```
Record the size of pairing heap.

```
    bool isEmpty();
```
Test whether the tree is empty for convenience.
```
    treenode* extractMin();
```
Get minimum node of the heap as the type of Huffman tree's node.

```
    void insert(node&newNode, treenode* root);
```
Insert a new node.

```
    void deleteMin();
```
Delete the minimum node of pairing heap.

4) Huffman tree

node structure

```
struct treenode {
    int data;
    int freq;
    treenode* left, * right;
};
```

(Some part of the code is modified based on https://github.com/rushi13/Huffman-Coding)

2. Building Huffman tree

I. Build a frequency table using *freq_table_builder.cpp* simply count how many times each number appears. Frequency table is saved into an unordered_map.

II. Call *build_tree_using_pairing_heap(freqTable,codeTable)* (or other types of priority queues) to do the following job.

  a) Build a priority heap according to frequency table. Although the implementation of priority queue is different, the exposed functionalities are the same.

b) Extract the minimum and the second minimum node as the left child and right child of a new tree node. The frequency field of the new node is the sum of its children’s frequency. Delete those two nodes and insert the new node into priority queue. The new node is the root of a subtree.

c) Repeat step b) until there is only one node in priority queue. The remaining node is the root of Huffman tree.

3. Encoder

Build a Huffman code table using *code_table_builder.cpp* by traversal from the root. Record a 0 each time goes to left child and record a 1 each time goes to right child. Save the recoding sequence of 0s and 1s and the data into an unorder_map once reaching a leaf node.

For every data in input file, write corresponding Huffman code found by code table into *encoded.bin.*

4. Decoder

I. Build code table according to code_table.txt.

II. Create a new tree with only one node.

III. Get a data and a sequence of code from code table:

   a) Set root of the tree as current node.

   b) Read a bit from code. If the byte equals to 0, go to left child of current node; else go to right child of current node. If there is no node among the path, create a new tree node.

   c) Repeat a) until every bit of code is used. Set the data field of last node of the path as data in code table.

IV. Repeat III. until entire code table is done. The Huffman tree is built

V. Start from the root of Huffman tree. Read a bit from *encoded.bin*:

   a) If the bit is 0, goes to left child; else goes to right child.

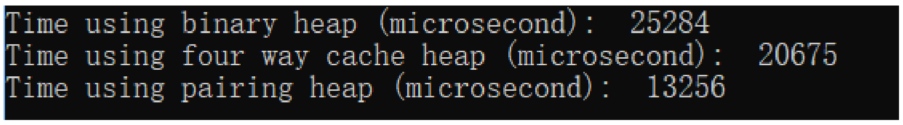
   b) If reaches leaf node, output the data saved in leaf node into *decoded.txt*, repeat entire step V; else repeat a)

VI. While all bit has been read from the binary file. The job is done.

Hence, time complexity equals to $O(n \log n)$, where n is the total number of elements and $\log n$ is height of corresponding Huffman tree.

5. Performance measurement

Only the time of building a Huffman tree through priority queue is recorded. Each method has been run for 10 times for more accurate result. The time is the mean time of 10 runs.



```
Time using binary heap (microsecond):  25284
Time using four way cache heap (microsecond):  20675
Time using pairing heap (microsecond):  13256
```

The comparing tree turns to be the fastest. The main operation in building a Huffman tree is extract min, insert, and delete min. The table show the complexity.

	Insert (and meld)	Extract min	Delete min
Pairing Heap	O(1) and O(1)	O(1)	O($log\ n$)(amortized)
Binary Heap	O($\log_2 n$)	O(1)	O($2\log_2 n$)
4-way Heap	O($\log_4 n$)	O(1)	O($4\log_4 n$)

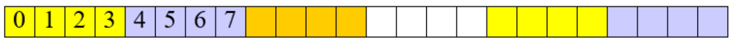
Pairing Heap is much faster due to faster insertion.

Every time we update Huffman tree, one insert and 2 delete min is done.

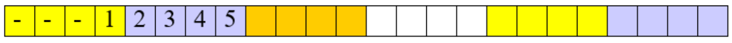
$$\log_4 n + 8\log_4 n = \log_2 n^{4.5}$$
$$\log_2 n + 4\log_2 n = \log_2 n^5$$

This is one reason why 4-way heap is faster than binary heap. Another reason is better use of cache. This slide explained 4-way heap uses cache in a more efficient way.

- Standard mapping into cache-aligned array.



- Siblings are in 2 cache lines.
  - ~$\log_2 n$ cache misses for average remove min (max).
- Shift 4-heap by 2 slots.



- Siblings are in same cache line.
  - ~$\log_4 n$ cache misses for average remove min (max).