COP 5536 Spring 2017

Advanced Data Structures

Project Report

Huffman Encoder and Decoder

(Programming Project)

Name: Akanksha Singh
UFID: 9551-8984
E - Mail: akanksha.singh@ufl.edu

Description of encoder.cpp:

Global Variable:
Name: my_vector **Datatype:** vector<string>

Class: Min_Heap_Node
Variables:

Name	Datatype	Description
data_field	string	Variable for input character
frequency	int	Stores character frequency
*leftChild	Min_Heap_Node pointer	pointer to left child
*rightChild	Min_Heap_Node pointer	pointer to right child

Functions:

Definition	Description	Arguments	Return Type/Value
Min_Heap_Node(string data_field, int frequency)	Class Constructor	data_field, frequency	None
int get_frequency()	Method to return frequency.	None	frequency
void set_children_Main(Min_Heap_Node *l, Min_Heap_Node* r)	Method to set pointer of right child and right child.	*l, *r (pointers to left and right child)	None
bool is_child(int child)	Method to check whether child of given node is present or not	child [receives numeric value].	True if child is present (i.e. child = 0/1), otherwise False.
Min_Heap_Node* get_child(int child)	Method to get child pointer of node.	child [receives numeric value].	Pointer to child (i.e. leftChild if child = 0; rightChild if child =1)

int is_Leaf()	Method to check if a node is a leaf node.	None	0 if it is a leaf node and 1 if not.
string get_data()	Method to get data field.	None	data_field

Class: Four_Way_Heap
Variables:

Name	**Datatype**	**Description**
CSize	int	Stores current size of heap
size	int	Stores size
**four_way_array	Four_Way_Heap Node pointer	For array implementation of heap

Functions:

Definition	Description	Arguments	Return Type/Value
Four_Way_Heap(int capacity)	Class Constructor	capacity	None
bool Check_Empty()	Method to check whether heap is empty or not.	None	True if empty, else False.
bool Check_Full()	Method to check whether heap is full or not.	None	True if full, else False.
void insert(Min_Heap_Node* temp)	Method to insert node in Heap.	temp has Node pointer to be inserted.	None

int parent(int i)	Method to return index of parent of i’th node.	index ‘i’	index of parent node.
int k_Child(int i, int k)	Method to return index of k’th child of node i.	index ‘i’, ‘k’.	index of k’th node.
Min_Heap_Node* findMin()	Method to find minimum element out of heap.	None	Pointer to Minimum Element of four_way _array
Min_Heap_Node* deleteMin()	Method to delete minimum element out of heap	None	Pointer to minimum element.
void buildHeap()	Method to build Heap	None	None
int smallestChild(int blank)	Method to get Smallest Child ofa node.	blank has index of the blank node created whose child is required.	index of smallest child
void Down_Heapify(int blank)	Method to Heapify Downwards	blank has index of blank node created in heap	None
void Up_Heapify(int blank)	Method to Heapify Upwards	blank has index of blank node created in heap	None

void set_array(int i, string data_field, int frequency)	Creates a new node and puts it in the array at i’th index	data_field and frequency for creating new node, and i is index of position.	None
void set_size(int size)	Sets the size of the heap for optimized heap (shift by 3)	size	None
string Print_Array(int arr[], int n)	Print the four way heap array as a string	arr has the array and n has the size of array	temp
void get_Huffman_Code(Min_Heap_Node* root, int arr[], int top)	Method to generate Huffman Code for each key	root – pointer to heap arr – array to store codes	None
void get_code_table()	Method to create Code Table	None	None
void get_encoded_data(vector<string> v)	Method to get data from input file and generate encoded.bin file	vector v	None
void Create_Huffman_Tree(map<string, int> FreqTable)	Method to build Huffman Tree	FreqTable – a hash map for storing frequency	None
map<string, int> read_file(char *filename)	Method to read input file and create a frequency table	filename	hash map that stores the frequency
int main(int argc, char *argv[])	Main Method	argc, argv	None

Description of decoder.cpp:

Class: Min_Heap_Node

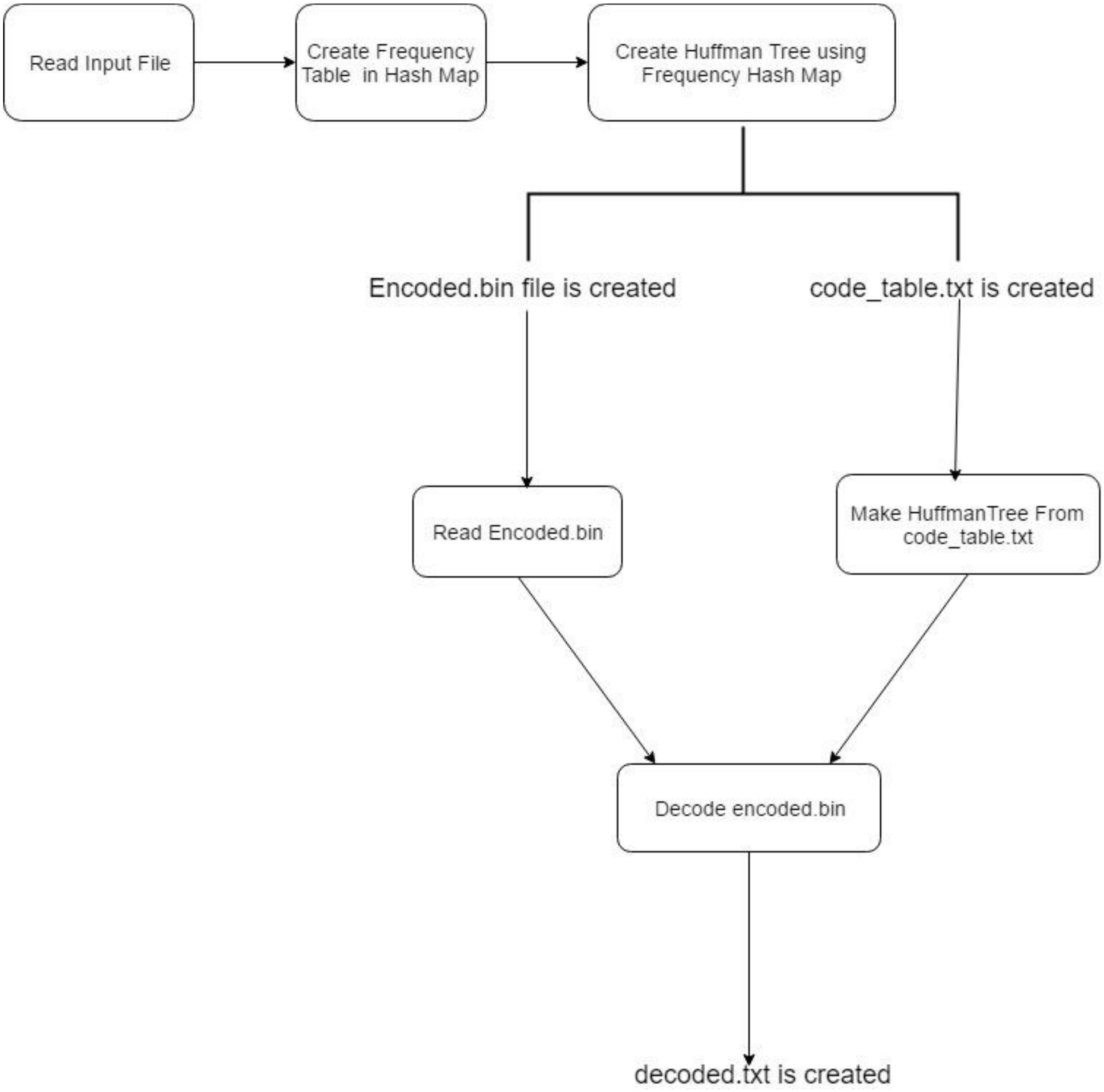
Variables:

Name	Datatype	Description
data_field	string	Variable for input character
*leftChild	Min_Heap_Node pointer	pointer to left child
*rightChild	Min_Heap_Node pointer	pointer to right child

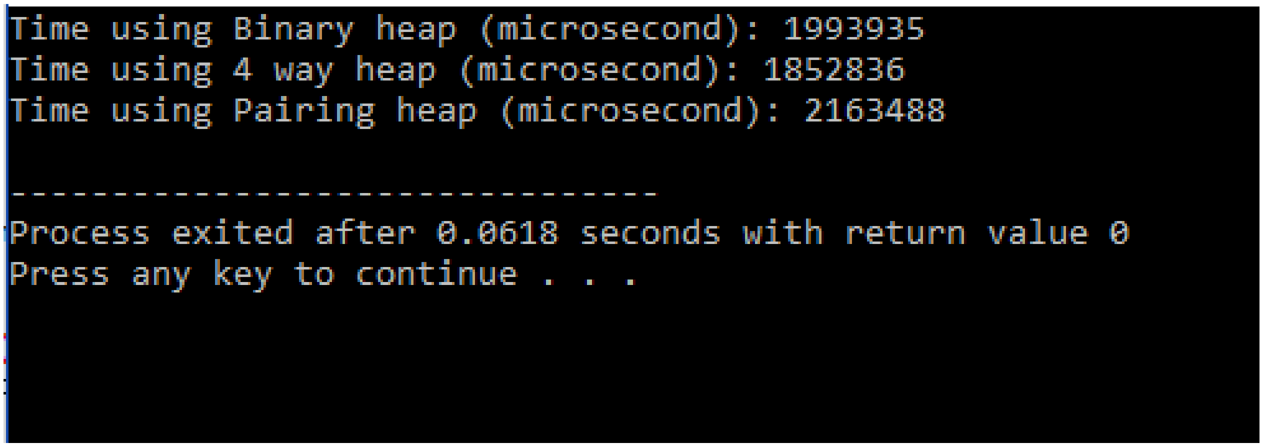
Functions:

Definition	Description	Arguments	Return Type/Value
Min_Heap_Node(string data_field)	Class Constructor	data_field	None
vector<int> read_file_Main(char* my_file)	Method to extract the binary bits from the file and store it in an integer vector	my_file – encoded file	vector
vector<int> Read_Encoded_File(char* my_file)	Method to read encoded file	my_file – encoded file	vector
Min_Heap_Node *Create_Huffman_Tree(char* my_file)	Method to create Huffman Tree from code table	my_file – encoded file	Min Heap Node
void Decode_Code_Table(Min_Heap_Node* root, vector<int> my_vector)	Method to decode data using Huffman tree.	root - of huffman tree vector – has encoded data in bits	None

Structure of Program


Read Input File
Create Frequency Table in Hash Map
Create Huffman Tree using Frequency Hash Map
Encoded.bin file is created
code_table.txt is created
Read Encoded.bin
Make HuffmanTree From code_table.txt
Decode encoded.bin
decoded.txt is created

Performance Analysis Results and Explanation



```
Time using Binary heap (microsecond): 1993935
Time using 4 way heap (microsecond): 1852836
Time using Pairing heap (microsecond): 2163488

--------------------------------
Process exited after 0.0618 seconds with return value 0
Press any key to continue . . .
```

From the analysis of the three data structures – pairing heap, binary heap and 4 way heap, we found that the cache optimised 4 way heap has the fastest running time amongst the three.
This is because in an optimised 4 way heap, as compare to a binary heap, the height is reduced by half, as it has 4 children (instead of 2), which generously reduces its time complexity. Also, for cache optimised 4 way has better memory cache than the other two, due to which it performs better in real time environment.

For this reason, we have chosen to 4 way cache optimised heap for our Huffman Encoder and Decoder.

Decoder - Complexity and Algorithm

Best Case – O(n logn)
We get a balanced tree when the frequency of all the items is same. As a result the traversal that one has to do to reach the leaf node is log(n). Number of nodes is n. Hence best case complexity is O(n logn).

Worst Case – O(n^2)
In worst case scenario, we get a tree with height equal to n-1. In this case we get a complexity of O(n^2).

Algorithm:

Step 1: Read code table from file and insert into hash map.
Step 2: Build Huffman Tree from code table map.

Step 3: Select two minimum from code table and create tree.
Step 4: Go left when we see a 0, right if we see a 1.
Step 5: If node is present in tree we traverse as it is. If not, we create a new node and go along it.
Step 6: When we reach the end of the ‘code bits’, we write its key in the data field.
Step 7: Repeat till the end of hash map.

```
Min_Heap_Node *Create_Huffman_Tree(char* my_file){
    map<string, string> Map;
    Min_Heap_Node* root = new Min_Heap_Node("$");
    Min_Heap_Node* head = root;
    Min_Heap_Node *temp = root;
    //feed code table to hashmap
    ifstream infile(my_file);
  if (infile.is_open()){
   string key, value;
   while (infile >> key >> value){
    Map[key] = value;
      }
    }
    else{
        cout<<"Could not open file\n";
    }

    //build huffman from code table
    for (map<string,string>::iterator x=Map.begin(); x!=Map.end(); ++x){
        root = temp;
    for (int i=0; i < x->second.size(); i++){

       if(x->second[i]!='0'){
        if(!root->right_child)
            root->right_child = new Min_Heap_Node("$");
        root = root->right_child;
       }
       else{
        if(!root->left_child)
            root->left_child = new Min_Heap_Node("$");
        root = root->left_child;
       }
    }
```

```
          root->data_field = x->first;
    }
    return head;
}

void Decode_Code_Table(Min_Heap_Node* root, vector<int> my_vector){
          Min_Heap_Node* head = root;
          ofstream outfile("decoded.txt");
    int i;

    if (outfile.is_open()){
              for(i = 0; i<my_vector.size(); i++){
                     if(my_vector[i])
                      root = root->right_child;
                  else
                      root = root->left_child;

                  if(!root->left_child && !root->right_child){
                      outfile << root->data_field << endl;
                      root = head;
                  }
              }
          }
          else{
                    cout<<"Could not open file\n";
          }
          outfile.close();
}
```