ASSIGNMENT 4 REPORT

Meryem Ülkü Kara

21727355

1- -i input_file.txt -encode –(ENCODING PART)

```
struct character_list
{
    char character;
    int character_frequency;
    struct character_list *next;
};
character_list *headList;
```

(Fig1.1)

I read the input_file.txt file given first and if the incoming character exists, I increased the frequency by 1 and switched to the other character. If the incoming character does not exist before, I created a new node and added it to the queue. If it is a line break I assumed it was \n and added it that way.This part is "LinkedList_Frequency"

Then I listed the node_tree I created according to their frequencies

```
do
{
    swapped = 0;
    lPtr = head;
    while(lPtr->next != rPrt)
    {
        if (lPtr->character_frequency > lPtr->next->character_frequency)
        {
            my_swap(lPtr, lPtr->next);
            swapped = 1;
        }
        lPtr = lPtr->next;
    }
    //as the largest element is at the end of the list, assign that to rPtr as there is no need to
    //check already sorted list
    rPrt = lPtr;

}while(swapped);
```

(Fig1.2)

this is my_swap method:

```
void my_swap (character_list *node_1,character_list *node_2)
{
    char temp = node_1->character;
    int temp1=node_1->character_frequency;
    node_1->character = node_2 -> character;
    node_1->character_frequency=node_2->character_frequency;
    node_2 -> character = temp;
    node_2->character_frequency=temp1;
}
```

(Fig1.3)

now i have a tiered character_list

Later I added a vector my character_list element.I used priority_queue

priority_queue is

std::priority_queue

Defined in header <queue>

```
template<
    class T,
    class Container = std::vector<T>,
    class Compare = std::less<typename Container::value_type>
> class priority_queue;
```

A priority queue is a container adaptor that provides constant time lookup of the largest (by default) element, at the expense of logarithmic insertion and extraction.

A user-provided Compare can be supplied to change the ordering, e.g. using `std::greater<T>` would cause the smallest element to appear as the `top()`.

Working with a `priority_queue` is similar to managing a heap in some random access container, with the benefit of not being able to accidentally invalidate the heap.

(Fig1.4)

```
// Create a vector for all elements
priority_queue<tree_node*, vector<tree_node*>, compare> create_tree;

void push_values_vector(char data, int freq)
{
        create_tree.push(new tree_node(data, freq));
}
```

(Fig1.5)

First create three tree_node this `*left, *right, *top`

```
while (create_tree.size() != 1) {


    //first find top and value added left leaf later delete top value
    left = create_tree.top();
    create_tree.pop();
    //second find top and value added right leaf later delete top value
    right = create_tree.top();
    create_tree.pop();
```

(Fig1.6)

Then I created top node for left and right.

```
      Top

   /      \

 Left    right
```

```
// Create a new internal node with old left leaf and right leaf.
//node data_name is R it is special value (I choose it is changable)
//frequency equals to sum of left_leaf and right_leaf frequency.
top = new tree_node('R', left->freq + right->freq);

//top left nodes is left
top->left = left;
//top right nodes is right
top->right = right;

//new top value added the create_tree
create_tree.push(top);
```

(Fig1.7)

After create create_tree added this a new linked list (tree_encode)

Finally I created a Tree.txt and printed my tree paying attention to the parent child.My tree have LeftSubtree0 (0) , RightSubtree1 (1) that is (1 and 0) Huffman encoding value. If the parent has a value of 0, it indicates that it is the left parent, in the same way if the parent has a value of 1 it indicates that it is the right parent.If the child's parent is 0, the child is left leaf, in the same way if the child's parent is 1 the child is right leaf.

```
Root
  |+Left Subtree0
  |   |+Parent0
  |   |   |Child -d-00
  |   |+Parent1
  |   |   |Child -b-01
  |+Right Subtree1
  |   |+Parent0
  |   |   |+Parent0
  |   |   |   |Child -e-100
  |   |   |+Parent1
  |   |   |   |Child -a-101
  |   |+Parent1
  |   |   |Child -c-11
```

(Fig1.8)

2 -i input_file.txt -decode –(DECODING PART)

First I read Tree.txt.

I used tokenize method to read tree.txt separated by "-" character

if the line doesn't have the hyphen character I skipped the line. I've checked the lines with hyphen characters.

I created a vector with values separated by hyphen character, then I created a new linked list with vectors

```
void tokenize(string &str, char delim, vector<string> &out)//split line space by space
{
    size_t start;
    size_t end = 0;
    int j=0;
    while ((start = str.find_first_not_of(delim, end)) != string::npos)
    {
        end = str.find(delim, start);
        out.push_back(str.substr(start, end - start));
        j++;
    }
}
```

(Fig2.1)

```
struct tree_decode{
    string data;
    string decode;
    tree_decode *next;
};
```

(Fig2.2)

Then I read input_file.txt. I put the information I read into a string, I checked the string with substr up to the longest decode and converted it to value.

i is line, j is char, k loop.

First I found longest encode from tree.txt. k must be smaller than longest encode value.If it is not decode is can't be done. l must be smaller than line total character if it is not we must skip after line. That is my decode skill format:

```
vector<string> characters;
while (getline(commands1,line1))
{characters.push_back(line1);}//end of while loop
int vector_size=characters.size();
bool a=false;
int i=0;//new line if it is not one line
int j=0;//skip if you are find
int k=1;
while(!a)
{
    int length_line=characters[i].size();
    string value = characters[i];
    int startIndex =j;//starting position 0
    int length = k;//length of the string
    string substring = value.substr(startIndex, length);
    a=decode_list_obj.search(substring);
```

(Fig2.3)

This is my search method

```
bool search( string search_element)
{
    tree_decode* current = head; // Initialize current
    while (current != NULL)
    {
        if (current->decode ==search_element)
        { string space=" ";
            if(current->data==space)
            {
                cout <<"\"" <<current->data <<"\"";
            }
            else{cout<<current->data; }

            return true;
        }
        current = current->next;
    }
    return false;
}
```

(Fig2.4)

İf value is space i print “ “.

İf find it skip this substring value and change new substring

```
if(a)//skip if you are find
{
    if(j<length_line)
    {
        j=j+k;k=1;
        if(j<length_line){a=false;}
        else
        {   if(i<vector_size){
                a=false;
                i++;
                j=0;
                k=1;}
        }
    }
    else{
        break;}
```

(Fig2.5)

3 -l – lists tree

In this command, I read the Tree.txt file in which I saved the structure of the tree

formed in the encoding section and printed its contents on the screen.
(like Fig1.8)

4 -s character –

First I read Tree.txt line by line. While reading, I checked the character of the data. When I found the character, I printed the decode value on the screen.

IF character(s character –) is uppercase , first i can change lowercase after search.

```
if(vector_size==3)//control tree child lines
{
    for(i=0;i<vector_size;i++)
    {
        if(i==1)
        {
            if(v[1]==character)
            {cout<<v[2];break;}
        }
    }
}
```

(Fig4.1)