# Specification

This program illustrate the working of Huffman encoding algorithm which was devised by David A. Huffman of MIT in 1952 for compressing text data to make a file occupy a smaller number of bytes. Normally text data is stored in a standard format of 8 bits per character using ASCII encoding. It maps every character to a binary integer value from 0-255.

The idea of Huffman encoding is to abandon the 8-bits-percharacter requirement and use different-length binary encodings for different characters. The advantage of doing this is that if a character occurs frequently in the file, such as the common letter 'e', it could be given a shorter encoding, making the file smaller. The trade off is that some characters may need to use encodings that are longer than 8 bits, but this is reserved for characters that occur infrequently, so usually the extra cost is worth it.

- **Note :**
  This program is just to illustrate the Huffman encoding algorithm using MeldableHeap which is a priority queue implementation.

# Design

1. **File: HNode.h**

```
#pragma once
template<typename T>
struct HNode{
T x;
int weight;
HNode *left;
HNode *right;
HNode *parent;

HNode(T u = ' ', int w = 0){
        x = u;
        weight = w;
        left = right = parent = nullptr;
}
bool operator>(const HNode& n)
{return weight > n.weight; }
};
```

- Description :
  Structure HNode is to store the char and its frequency.
  While building the tree we will store this HNode
  inside Meldable Class Node
- Data Members :

(a) x :-
  To store the character

(b) weight :-
  To store the frequency of the character

(c) *left :-
  HNode pointer to store left child

(d) *right :-
  HNode pointer to store right child

(e) *parent :-
  HNode pointer to store parent of the HNode

- Constructor :
  The constructor of this struct will initialize
  x with the passed parameter and by default x
  will be a space. weight will be initialize to
  the passed integer.
  Left, right and parent will be set to a nullptr.
- Overloaded Operator > :
  greater than operator is overloaded so when creating the
  priority queue, two HNodes can be compared by their
  weight.

## 2. File: buildHuffmanTree.h

```
#include "MeldableHeap.hpp"
#include "HNode.h"
#include <map>
#include <string>
template<typename T>
class buildHuffmanTree
{
public:
MeldableHeap <HNode<T>> bh;
HNode<T>* r;
std::string s;

buildHuffmanTree(std::string str){
    s = str;
}
```

- Description :
  class buildHuffmanTree implements the Huffman encoding algorithm. It will create a Huffman tree with its tree() method which can be traversed to get the binary codes for each character.
- Data Members :
- Data Members :

(a) bh :-
bh is an object of class MeldableHeap
and type thats passed in when creating
bh is HNode.

(b) *r :-
pointer r of HNOde type is to store
the root of bh once its build using the
tree() method.

(c) s :-
s is the string which will be parsed to obtain
the characters which will futher be converted
to binary codes.

- Constructor :
  The constructor of this class will initialize the s data member with the passed string.
- tree() method :
- printCodes() method :
- isLeaf() methor :
- printArr() methor :

```
MeldableHeap <HNode<T>> tree() {
std::map<char, int> m;
for(auto e : s) m[e]++;
for(auto e : m) {
std::cout<<e.first <<" "<<e.second;
std::cout<<std::endl;
HNode<T>*n = new HNode<T>(e.first,e.second);
bh.add(*n);
}
while(bh.n > 1) {
        HNode<T>* n1 = new HNode<T>();
    *n1 = bh.remove();
    HNode<T>* n2 = new HNode<T>();
    *n2 = bh.remove();
    HNode<T>* h = new HNode<T>
    ('*', n1->weight+n2->weight);
    r = h;
    n1->parent = h;
    n2->parent = h;
    h->left = n1;
    h->right = n2;
    bh.add(*h);
}
return bh;
}
```

- tree() method :

(a) Step 1 :- In the tree method we first create a map m to map the character with its frequency using a for loop.

(b) Step 2 :-In the second for loop we display the character and its frequency and create a HNode pointer to store the char and frequency.

(c) Step 3 :-Then we add that to the tree which will implement the priority queue and keep the characters with lowest frequency on the top.

(d) Step 4 :- Then in a while loop we remove two nodes from the tree and store them in n1 and n2. Then we create another node with the accumlated weight of n1 and n2 and add it back to the tree. Also we set the left and right childs of h to n1 and n2 respectively.

(e) Step 5 :- Once we are done with the while loop we return the tree.

```
void printCodes
(HNode<T> * root, int arr[], int top){

if (root->left) {
    arr[top] = 0;
    printCodes(root->left, arr, top + 1);
}
if (root->right) {
    arr[top] = 1;
    printCodes(root->right, arr, top + 1);
}
if (isLeaf(root)) {
        std::cout << root->x.x << " :" ;
    printArr(arr, top);
}

}
```

- printCodes() method :
  It prints huffman codes from the root of Huffman Tree.
  It uses arr[] to store codes and top is the length of
  the code which increases with every recursive call to
  printCodes().
- In the first if statement we
  assign 0 to left edge and recur
- In the seconde if statement we
  assign 1 to right edge and recur
- In the third if statement if this it
  hits a leaf node, then
  it contains one of the input
  characters, print the character
  and its code from arr[]
- This function is to display the binary
  code for each character and it returns nothing

```
int isLeaf(HNode<T> * root){
   return !(root->left) && !(root->right);
}
```

- isLeaf() methor :
  It is a utility function to check if this node is leaf.
  It will return 0 if the passed node in not a leaf and
  1 if it is.

```
void printArr(int arr[], int n){
    int i;  std::string bits;
    for (i = 0; i < n; ++i){
        std::cout<< arr[i];
         bits += arr[i];
     }
    std::cout<<"\n";
}

};
```

- printArr() methor :
  It is a utility function to print an array of size n

## 3. File: MeldableHeap.h

```
#include <string>
#include "HNode.h"
#include <cstdlib>
```

struct Node is to store the date in x and has left, right and parent pointers.

```
template<typename T> struct Node{
                T x;
                Node *left;
                Node *right;
                Node *parent;
```

Constructor of node will intialize x with u which will be passed when creating a Node object.
The left, right and parent pointers are set null initially.

```
                Node(T u)
                {
                        x = u;
                        left = right = parent = nullptr;
                }
};

template<typename T> class MeldableHeap
{

public:
        Node<T> *r;
        int n;

        MeldableHeap()
        {
                r = nullptr;
```

```
            n = 0;
        }

        bool add(T x) {
            Node<T> *u = new Node<T>(x);
            r = merge(u, r);
            r->parent = nullptr;
            n++;
            return true;
        }

        Node<T>* merge(Node<T> *h1, Node<T> *h2) {
            if (h1 == nullptr) return h2;
            if (h2 == nullptr) return h1;
            if ((h1->x) > (h2->x)) std::swap(h1,h2);

            if (rand() % 2) {
            h1->left = merge(h1->left, h2);
            if (h1->left != nullptr) h1->left->parent = h1;
            } else {
            h1->right = merge(h1->right, h2);
            if (h1->right != nullptr) h1->right->parent = h1;
            }
            return h1;
        }

        T remove(){
            T x;
            x = r->x;
            r = merge(r->left,r->right);
            n--;
            return x;
        }



};
```

4. **File: lab.h**

```
#include <iostream>
#include <string>
#include "buildHuffmanTree.h"
```

```
int main (){
        std::string s1 = "ohlone college";
        buildHuffmanTree<HNode<char>> a(s1);
        a.tree();
        int arr[0];
        a.printCodes(a.r,arr,0);
        std::string s2 = "Huffman encoding is an algorithm devised by David A. Huffman";
        buildHuffmanTree<HNode<char>> b(s2);
        b.tree();
        int arr2[0];
        b.printCodes(b.r,arr2,0);
  return 0;
}
```

(a)

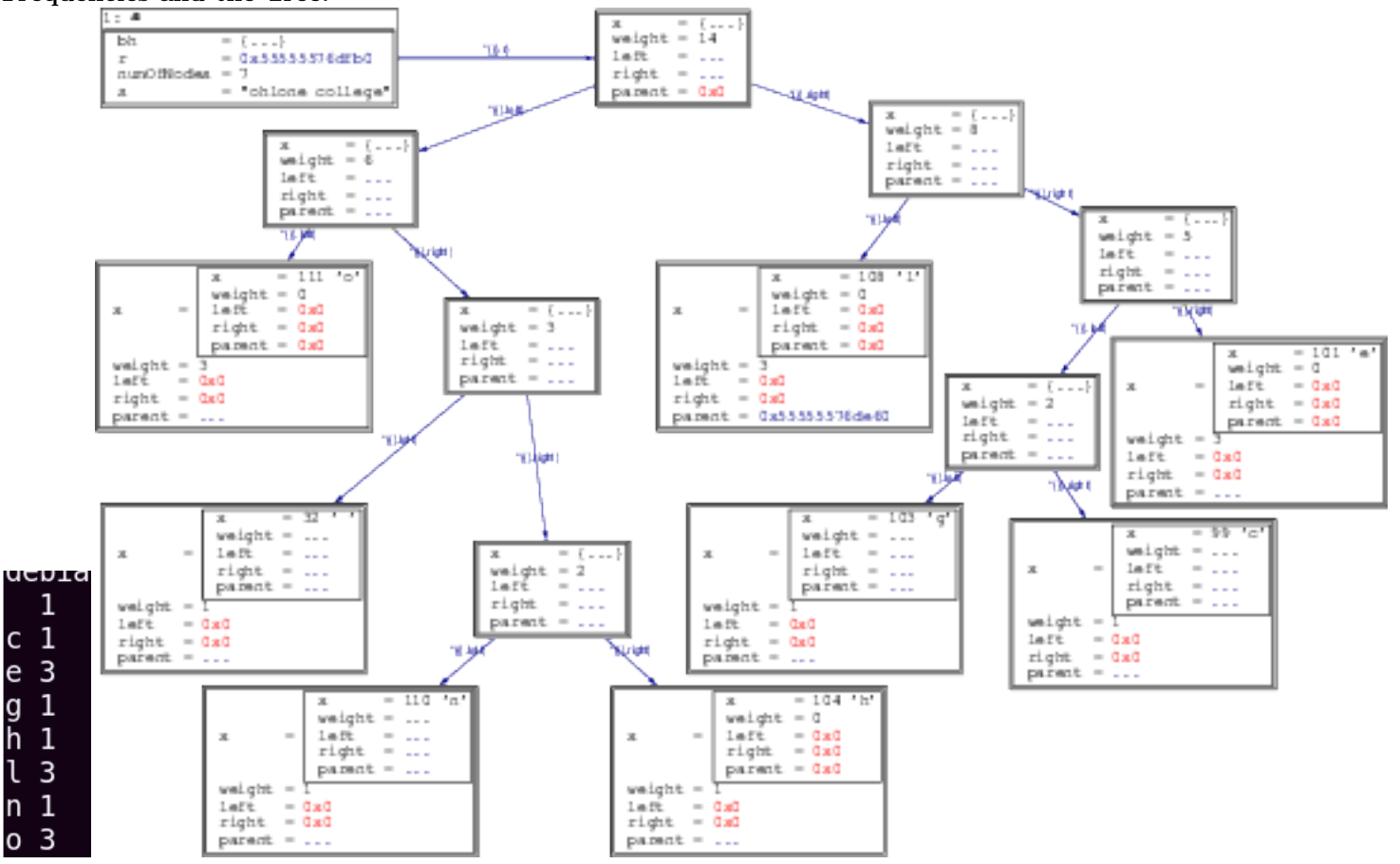
- In the main function we test the methods of our buildHuffmanTree class by creating a as its object and pass the string s to it.
- we call the tree method which should print out the character and their frequencies and it should also set r to the root of tree.
- Then we call the printCode() method to see if the tree build by the tree() method leads to the right binary code for each character.

# Test

## Testcase 1

For the string "ohlone college"

- **Frequencies and the Tree:**




**Frequencies match if counted manually**

- **Binary Code :**

```
o :00
  :010
n :0110
h :0111
l :10
g :1100
c :1101
e :111
```

## Testcase 2

For the string "Huffman encoding is an algorithm devised by David A. Huffman"

- **Frequencies :**

```
  9
. 1
A 1
D 1
H 2
a 5
b 1
c 1
d 4
e 3
f 4
g 2
h 1
i 5
l 1
m 3
n 5
o 2
r 1
s 2
t 1
u 2
v 2
y 1
```

**Frequencies match if counted manually**

- **Binary Code :**

```
n :000
i :001
e :0100
A :010100
. :010101
h :010110
c :010111
g :01100
b :011010
D :011011
f :0111
H :10000
r :100010
l :100011
d :1001
o :10100
u :10101
s :10110
v :10111
  :110
y :111000
t :111001
m :11101
a :1111
```