Q-1: Implement a program to build a Huffman tree from a given set of characters along with their frequencies.

Sample test case:

|  |
| --- |
| Input: arr = { 'a', 'b', 'c', 'd', 'e', 'f' };  freq = { 5, 9, 12, 13, 16, 45 }  Output:  Huffman Codes:  f: 0  c: 100  d: 101  a: 1100  b: 1101  e: 111 |

Solution:

#include <cstdlib>

#include <iostream>

using namespace std;

// This constant can be avoided by explicitly calculating height of Huffman Tree

#define MAX\_TREE\_HT 100

// A Huffman tree node

struct MinHeapNode {

// One of the input characters

char data;

// Frequency of the character

unsigned freq;

// Left and right child of this node

struct MinHeapNode \*left, \*right;

};

// A Min Heap: Collection of

// min-heap (or Huffman tree) nodes

struct MinHeap {

// Current size of min heap

unsigned size;

// capacity of min heap

unsigned capacity;

// Array of minheap node pointers

struct MinHeapNode\*\* array;

};

// A utility function allocate a new

// min heap node with given character

// and frequency of the character

struct MinHeapNode\* newNode(char data, unsigned freq)

{

struct MinHeapNode\* temp = (struct MinHeapNode\*)malloc(

sizeof(struct MinHeapNode));

temp->left = temp->right = NULL;

temp->data = data;

temp->freq = freq;

return temp;

}

// A utility function to create

// a min heap of given capacity

struct MinHeap\* createMinHeap(unsigned capacity)

{

struct MinHeap\* minHeap

= (struct MinHeap\*)malloc(sizeof(struct MinHeap));

// current size is 0

minHeap->size = 0;

minHeap->capacity = capacity;

minHeap->array = (struct MinHeapNode\*\*)malloc(

minHeap->capacity \* sizeof(struct MinHeapNode\*));

return minHeap;

}

// A utility function to

// swap two min heap nodes

void swapMinHeapNode(struct MinHeapNode\*\* a,

struct MinHeapNode\*\* b)

{

struct MinHeapNode\* t = \*a;

\*a = \*b;

\*b = t;

}

// The standard minHeapify function.

void minHeapify(struct MinHeap\* minHeap, int idx)

{

int smallest = idx;

int left = 2 \* idx + 1;

int right = 2 \* idx + 2;

if (left < minHeap->size

&& minHeap->array[left]->freq

< minHeap->array[smallest]->freq)

smallest = left;

if (right < minHeap->size

&& minHeap->array[right]->freq

< minHeap->array[smallest]->freq)

smallest = right;

if (smallest != idx) {

swapMinHeapNode(&minHeap->array[smallest],

&minHeap->array[idx]);

minHeapify(minHeap, smallest);

}

}

// A utility function to check

// if size of heap is 1 or not

int isSizeOne(struct MinHeap\* minHeap)

{

return (minHeap->size == 1);

}

// A standard function to extract

// minimum value node from heap

struct MinHeapNode\* extractMin(struct MinHeap\* minHeap)

{

struct MinHeapNode\* temp = minHeap->array[0];

minHeap->array[0] = minHeap->array[minHeap->size - 1];

--minHeap->size;

minHeapify(minHeap, 0);

return temp;

}

// A utility function to insert

// a new node to Min Heap

void insertMinHeap(struct MinHeap\* minHeap,

struct MinHeapNode\* minHeapNode)

{

++minHeap->size;

int i = minHeap->size - 1;

while (i

&& minHeapNode->freq

< minHeap->array[(i - 1) / 2]->freq) {

minHeap->array[i] = minHeap->array[(i - 1) / 2];

i = (i - 1) / 2;

}

minHeap->array[i] = minHeapNode;

}

// A standard function to build min heap

void buildMinHeap(struct MinHeap\* minHeap)

{

int n = minHeap->size - 1;

int i;

for (i = (n - 1) / 2; i >= 0; --i)

minHeapify(minHeap, i);

}

// A utility function to print an array of size n

void printArr(int arr[], int n)

{

int i;

for (i = 0; i < n; ++i)

cout << arr[i];

cout << "\n";

}

// Utility function to check if this node is leaf

int isLeaf(struct MinHeapNode\* root)

{

return !(root->left) && !(root->right);

}

// Creates a min heap of capacity

// equal to size and inserts all character of

// data[] in min heap. Initially size of

// min heap is equal to capacity

struct MinHeap\* createAndBuildMinHeap(char data[],

int freq[], int size)

{

struct MinHeap\* minHeap = createMinHeap(size);

for (int i = 0; i < size; ++i)

minHeap->array[i] = newNode(data[i], freq[i]);

minHeap->size = size;

buildMinHeap(minHeap);

return minHeap;

}

// The main function that builds Huffman tree

struct MinHeapNode\* buildHuffmanTree(char data[],

int freq[], int size)

{

struct MinHeapNode \*left, \*right, \*top;

// Step 1: Create a min heap of capacity

// equal to size. Initially, there are

// modes equal to size.

struct MinHeap\* minHeap

= createAndBuildMinHeap(data, freq, size);

// Iterate while size of heap doesn't become 1

while (!isSizeOne(minHeap)) {

// Step 2: Extract the two minimum

// freq items from min heap

left = extractMin(minHeap);

right = extractMin(minHeap);

/\* Step 3: Create a new internal

node with frequency equal to the

sum of the two nodes frequencies.

Make the two extracted node as

left and right children of this new node.

Add this node to the min heap

'$' is a special value for internal nodes, not used \*/

top = newNode('$', left->freq + right->freq);

top->left = left;

top->right = right;

insertMinHeap(minHeap, top);

}

// Step 4: The remaining node is the

// root node and the tree is complete.

return extractMin(minHeap);

}

// Prints huffman codes from the root of Huffman Tree.

// It uses arr[] to store codes

void printCodes(struct MinHeapNode\* root, int arr[],

int top)

{

// Assign 0 to left edge and recur

if (root->left) {

arr[top] = 0;

printCodes(root->left, arr, top + 1);

}

// Assign 1 to right edge and recur

if (root->right) {

arr[top] = 1;

printCodes(root->right, arr, top + 1);

}

// If this is a leaf node, then it contains one of the input characters, print the character and its code from arr[]

if (isLeaf(root)) {

cout << root->data << ": ";

printArr(arr, top);

}

}

// The main function that builds a Huffman Tree

void HuffmanCodes(char data[], int freq[], int size)

{

// Construct Huffman Tree

struct MinHeapNode\* root

= buildHuffmanTree(data, freq, size);

// Print Huffman codes using the Huffman tree built above

int arr[MAX\_TREE\_HT], top = 0;

printCodes(root, arr, top);

}

int main()

{

char arr[] = { 'a', 'b', 'c', 'd', 'e', 'f' };

int freq[] = { 5, 9, 12, 13, 16, 45 };

int size = sizeof(arr) / sizeof(arr[0]);

HuffmanCodes(arr, freq, size);

return 0;

}

Q-2: Implement a program to encode and decode a given text using Huffman coding.

Sample test case:

|  |
| --- |
| Input: text= linkedin  Output:  Character With there Frequencies:  d 010  e 011  i 00  k 111  l 110  n 10  Encoded Huffman data: 11000101110110100010  Decoded Huffman Data: linkedin |

Solution:

#include <bits/stdc++.h>

#define MAX\_TREE\_HT 256

using namespace std;

// to map each character its huffman value

map<char, string> codes;

// To store the frequency of character of the input data

map<char, int> freq;

// A Huffman tree node

struct MinHeapNode {

char data; // One of the input characters

int freq; // Frequency of the character

MinHeapNode \*left, \*right; // Left and right child

MinHeapNode(char data, int freq)

{

left = right = NULL;

this->data = data;

this->freq = freq;

}

};

// utility function for the priority queue

struct compare {

bool operator()(MinHeapNode\* l, MinHeapNode\* r)

{

return (l->freq > r->freq);

}

};

// utility function to print characters along with

// there huffman value

void printCodes(struct MinHeapNode\* root, string str)

{

if (!root)

return;

if (root->data != '$')

cout << root->data << ": " << str << "\n";

printCodes(root->left, str + "0");

printCodes(root->right, str + "1");

}

// utility function to store characters along with

// there huffman value in a hash table, here we

// have C++ STL map

void storeCodes(struct MinHeapNode\* root, string str)

{

if (root == NULL)

return;

if (root->data != '$')

codes[root->data] = str;

storeCodes(root->left, str + "0");

storeCodes(root->right, str + "1");

}

// STL priority queue to store heap tree, with respect

// to their heap root node value

priority\_queue<MinHeapNode\*, vector<MinHeapNode\*>, compare>

minHeap;

// function to build the Huffman tree and store it

// in minHeap

void HuffmanCodes(int size)

{

struct MinHeapNode \*left, \*right, \*top;

for (map<char, int>::iterator v = freq.begin();

v != freq.end(); v++)

minHeap.push(new MinHeapNode(v->first, v->second));

while (minHeap.size() != 1) {

left = minHeap.top();

minHeap.pop();

right = minHeap.top();

minHeap.pop();

top = new MinHeapNode('$',

left->freq + right->freq);

top->left = left;

top->right = right;

minHeap.push(top);

}

storeCodes(minHeap.top(), "");

}

// utility function to store map each character with its

// frequency in input string

void calcFreq(string str, int n)

{

for (int i = 0; i < str.size(); i++)

freq[str[i]]++;

}

// function iterates through the encoded string s

// if s[i]=='1' then move to node->right

// if s[i]=='0' then move to node->left

// if leaf node append the node->data to our output string

string decode\_file(struct MinHeapNode\* root, string s)

{

string ans = "";

struct MinHeapNode\* curr = root;

for (int i = 0; i < s.size(); i++) {

if (s[i] == '0')

curr = curr->left;

else

curr = curr->right;

// reached leaf node

if (curr->left == NULL and curr->right == NULL) {

ans += curr->data;

curr = root;

}

}

// cout<<ans<<endl;

return ans + '\0';

}

int main()

{

string str = "linkedin";

string encodedString, decodedString;

calcFreq(str, str.length());

HuffmanCodes(str.length());

cout << "Character With there Frequencies:\n";

for (auto v = codes.begin(); v != codes.end(); v++)

cout << v->first << ' ' << v->second << endl;

for (auto i : str)

encodedString += codes[i];

cout << "\nEncoded Huffman data:\n"

<< encodedString << endl;

decodedString= decode\_file(minHeap.top(), encodedString);

cout << "\nDecoded Huffman Data:\n"

<< decodedString << endl;

return 0;

}

Q-3: Write a C++ Program for Huffman Coding using Priority Queue

Sample test case:

|  |
| --- |
| Input: arr = { 'a', 'b', 'c', 'd', 'e', 'f' };  freq = { 5, 9, 45, 13, 16, 12 }  Output:  c 0  f 100  d 101  a 1100  b 1101  e 111 |

Solution:

#include <iostream>

#include <queue>

using namespace std;

// Maximum Height of Huffman Tree.

#define MAX\_SIZE 100

class HuffmanTreeNode {

public:

// Stores character

char data;

// Stores frequency of the character

int freq;

// Left child of the current node

HuffmanTreeNode\* left;

// Right child of the current node

HuffmanTreeNode\* right;

// Initializing the current node

HuffmanTreeNode(char character,

int frequency)

{

data = character;

freq = frequency;

left = right = NULL;

}

};

// Custom comparator class

class Compare {

public:

bool operator()(HuffmanTreeNode\* a,

HuffmanTreeNode\* b)

{

// Defining priority on

// the basis of frequency

return a->freq > b->freq;

}

};

// Function to generate Huffman Encoding Tree

HuffmanTreeNode\* generateTree(priority\_queue<HuffmanTreeNode\*,

vector<HuffmanTreeNode\*>,

Compare> pq)

{

// We keep on looping till only one node remains in the Priority Queue

while (pq.size() != 1) {

// Node which has least

// frequency

HuffmanTreeNode\* left = pq.top();

// Remove node from Priority Queue

pq.pop();

// Node which has least frequency

HuffmanTreeNode\* right = pq.top();

pq.pop();

// A new node is formed

// with frequency left->freq

// + right->freq

// We take data as '$' because we are only concerned with the frequency

HuffmanTreeNode\* node = new HuffmanTreeNode('$',

left->freq + right->freq);

node->left = left;

node->right = right;

pq.push(node);

}

return pq.top();

}

// Function to print the

// huffman code for each

// character.

// It uses arr to store the codes

void printCodes(HuffmanTreeNode\* root,

int arr[], int top)

{

// Assign 0 to the left node

// and recur

if (root->left) {

arr[top] = 0;

printCodes(root->left,

arr, top + 1);

}

// Assign 1 to the right

// node and recur

if (root->right) {

arr[top] = 1;

printCodes(root->right, arr, top + 1);

}

// If this is a leaf node,

// then we print root->data

// We also print the code

// for this character from arr

if (!root->left && !root->right) {

cout << root->data << " ";

for (int i = 0; i < top; i++) {

cout << arr[i];

}

cout << endl;

}

}

void HuffmanCodes(char data[],

int freq[], int size)

{

// Declaring priority queue

// using custom comparator

priority\_queue<HuffmanTreeNode\*,

vector<HuffmanTreeNode\*>,

Compare>

pq;

// Populating the priority

// queue

for (int i = 0; i < size; i++) {

HuffmanTreeNode\* newNode

= new HuffmanTreeNode(data[i], freq[i]);

pq.push(newNode);

}

// Generate Huffman Encoding Tree and get the root node

HuffmanTreeNode\* root = generateTree(pq);

// Print Huffman Codes

int arr[MAX\_SIZE], top = 0;

printCodes(root, arr, top);

}

int main()

{

char data[] = { 'a', 'b', 'c', 'd', 'e', 'f' };

int freq[] = { 5, 9, 45, 13, 16, 12 };

int size = sizeof(data) / sizeof(data[0]);

HuffmanCodes(data, freq, size);

return 0;

}