**Homework 6\_1 [heapsort.cpp]**

**Variable analysis**

**Heap**

**1)**

heap : heap represented in array

heap\_ size : size of heap

|  |  |
| --- | --- |
| **HeapType** | |
| **type** | **name** |
| Element [MAX\_ELEMENT] | heap |
| int | heap\_ size |

2) element : Elements used in the heap

key : the criteria for heap

|  |  |
| --- | --- |
| **element** | |
| **type** | **name** |
| int | key |

**Function analysis**

1) void build\_max\_heap(HeapType \*h)

: The purpose of this function is build max heap. We can build a max heap in a bottom-up manner by moving the element to meet the heap property. For the array of length N, all elements in N/2+1 …N already meet heap property.(no child node) So we don’t need to check N/2+1 ~ N th node meet heap property. Thus, walk backwards through the array from n/2 to 1 , moving the element on each node until it meets the heap property. The order of bottom-up processing guarantees that the children of node i are heaps when i is processed.

The basic principles are as belows.

while decreasing the index from heap\_size/2 to 1, follow these processes.

Compare the value of the child node with the parent node.

Variable `child` indicates index that has larger key between left and right .

if parent >= child , it satisfys heap property and break the loop

else ( parent < child), it doesn’t meet heap property. So parent =child and move down one level to find proper place. (we don’t need to swap parent and child.).

void build\_max\_heap(HeapType \*h)

{

int parent, child;

element temp ;

int i;

for (int i = h->heap\_size/2; i >= 1; i--) {

parent = i;

child = 2 \* i; //left

temp = h->heap[parent]; // find proper place where temp can be inserted.

// The process of comparing with the parent node as it traverses the tree

while (child <=h->heap\_size ) {

if ((child + 1 <= h->heap\_size) && (h->heap[child].key <= h->heap[child + 1].key)) // if right child exists and left < right

child++; //renew child as index of right node.

if (temp.key >= h->heap[child].key) // parent >= child -> meet heap property break;

//if do not meet heap property

h->heap[parent] = h->heap[child]; // parent = child

// Move down one level

parent = child;

child = parent \* 2;

}

h->heap[parent] = temp; // put temp in proper place

}

[result]

텍스트이(가) 표시된 사진

자동 생성된 설명Input\_size=10000



Input\_size=100000



Input\_size=1000000

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

Input\_size=10000000



:as the input size increases, the time required increases linearly. So the time complexity of buld\_max\_heap is O(N) .

**Homework 6\_2 [Huffman.cpp]**

**Huffman code**

- Algorithm used to efficiently compress data and belongs to the grid algorithm.  
  
- Depending on the frequency of data, there is a process of taking out two small nodes from the priority queue and combining them to create a tree.  
  
- There are two ways to express a fixed length code and a variable length code. And we use variable length code in this code.

**Variable analysis**

**Heap**

heap : heap represented in array

heap\_ size : size of heap

|  |  |
| --- | --- |
| **HeapType** | |
| **type** | **name** |
| element [MAX\_ELEMENT] | heap |
| int | heap\_ size |

**General**

1) Input\_huff : Input data for huffman code

data : Character array (a ~ f)

freq : Frequency array

size: Number of characters

|  |  |
| --- | --- |
| **Input\_huff** | |
| **type** | **name** |
| char \* | data |
| int\* | freq |
| Int | size |

2) TreeNode : Structure for huffman binary tree

data : Character array (a ~ f)

freq : Frequency

bits : Huffman codeword

bit\_size : Huffman codeword's size

l : Left child of huffman binary tree

r : Right child of huffman binary tree

|  |  |
| --- | --- |
| **TreeNode** | |
| **type** | **name** |
| char | data |
| int | key |
| int [MAX\_BIT] | bit |
| int | bit\_size |
| TreeNode\* | l |
| TreeNode\* | r |

3) Structure for bits stream

Stream : encoded data

Length : length of encoded data

|  |  |
| --- | --- |
| **bits\_stream** | |
| **type** | **name** |
| int \* | stream |
| int | length |

4) element : Elements used in the heap

ptree : the address of root

key : frequency of each character

|  |  |
| --- | --- |
| **element** | |
| **type** | **name** |
| TreeNode \* | ptree |
| int | key |

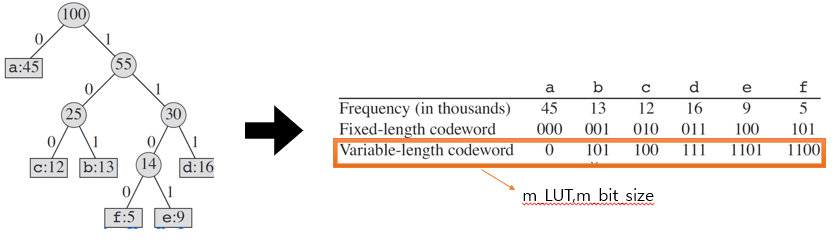
5) int \*\* m\_LUT , \*m\_bit\_size **:** save cord word for each character

6) m\_char\_size : the number of types of character used in data

**Function analysis**

1) void huffman\_traversal(TreeNode \*node)

: The purpose of this function is to make huffman cordword from given Huffman binary tree.



We have to travel all nodes in Huffman binary tree and find cordword for each character. For it, we can use variable ‘bits[]’ and bits\_size in each Tree Node. It saves the codeword for each node. When it moves from parent to child, the child has same codeword except for just last 1 bits. So child copy parent’s cordword and add just new bit depending on left or right child. If left, bit ‘0’ will added and vice versa. These can be implemented by following below rules.

1. If Left child exists

: inherit current node’s codeword to left child. And add bit ‘0’ to left child.

Then recursive call with parameter left.

1. If Right child exists

: inherit current node’s codeword to right child. And add bit ‘1’ to right child.

Then recursive call with parameter right.

1. If No child exists

: (=leaf node). The bits and bits\_size saved in current TreeNode should be moved into m\_LUT and m\_bit\_size. (because leafnode has character data). Depending on what character each node has, we have to save cordword at different index of m\_LUT and m\_bits\_size. (because m\_LUT and m\_bit\_size are ascending order) Thus we subtracts ‘a’ to find out which index the current cordword should be stored in.

The implementation of this explaination is as below.

// Generate the huffman codeword from the huffman binary tree

// input: root node

// output: m\_LUT, m\_bit\_size

void huffman\_traversal(TreeNode \*node)

{

if (node == NULL)

return;

//left -> 0 ,right ->1

if (node->l !=NULL) {

//inherit the same codeword from parent

for (int i = 0; i < node->bit\_size;i++) {

node->l->bits[i] = node->bits[i];

}

// add new bit ‘0’ to child node

node->l->bits[node->bit\_size] = 0;

node->l->bit\_size = node->bit\_size + 1;

huffman\_traversal(node->l); //recursive call

}

if (node->r!=NULL) {

//inherit the same codeword from parent

for (int i = 0; i < node->bit\_size; i++) {

node->r->bits[i] = node->bits[i];

}

// add new bit ‘1’ to child node

node->r->bits[node->bit\_size] = 1;

node->r->bit\_size = node->bit\_size + 1;

huffman\_traversal(node->r);

}

//if leaf node. set m\_LUT and m\_bit\_size.

if ((node->r == NULL) && (node->l == NULL)) {

int index = node->data - 'a'; //find index

//copy cord word from TreeNode to m\_LUT

for (int i = 0; i < node->bit\_size; i++) {

printf("%d", node->bits[i]);

m\_LUT[index][i] = node->bits[i];

}

//copy also size of cordword.

m\_bit\_size[index] = node->bit\_size;

}

}

**:** Finally, The cordword for each character will be saved into m\_LUT by this function.

2) void huffman\_encoding(char \*str, bits\_stream \*bits\_str)

// Return the total length of bits\_stream

void huffman\_encoding(char \*str, bits\_stream \*bits\_str)

{

int interval=0; // from where to save encoded data

for (int i = 0; i < strlen(str); i++) {

int index = str[i] - 'a'; // find m\_LUT’s index of current char

//copy codeword to stream per character

for (int j = 0; j < m\_bit\_size[index]; j++) {

bits\_str->stream[interval+j] = m\_LUT[index][j];

bits\_str->length++;

}

interval += m\_bit\_size[index]; //find next start index

}

printf("\n\* Huffman encoding\n");

printf("total length of bits stream: %d\n", interval);

printf("bits stream: ");

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

printf("%d", bits\_str->stream[i]);

printf("\n");

}

: In this function, we encode the given str into bit\_str using m\_LUT and m\_bit\_size. Check the character sequentially and save each character’s cordword to stream. The variable `interval` is used to know from which index of stream to save current character’s cordword sequentially. So in every loop ,`interval` is added as long as the length of the cordword stored just before. After all loops, ,`interval` indicates the length of encoded data. The variable `index` tells you at which index of m\_LUT the codeword corresponding to current character is stored.

3)int huffman\_decoding(bits\_stream \*bits\_str, TreeNode \*node, char \*decoded\_str)

: In this function, we decode given bits\_str using Huffman binary tree. In Huffman code, the prefix of one codeword cannot be another codeword and we can use this speciality in this function . It needs two process; the one is moving Treenode according to bit( 0 or 1) and the other is to check if node has char data.

Step 1. if current pointing bit is ‘0’ , move to left node. if current pointing bit is ‘1’, move to right node

Step 2. if node has char data, add it to decoded\_str.

int huffman\_decoding(bits\_stream \*bits\_str, TreeNode \*node, char \*decoded\_str)

{

int index\_char = 0; // index of decoded str

TreeNode\* cur =node;

for (int i = 0; i < bits\_str->length; i++) {

if (bits\_str->stream[i] == 0)

{

cur = cur->l; //move to left child

}

else {

cur = cur->r; // move to right child

}

if (cur->data != NULL) // if node has char data(= leaf node)

{

decoded\_str[index\_char++] = cur->data; // add data to decoded\_str

cur = node; // initialize to root for next decoding

}

}

printf("\n\* Huffman decoding\n");

printf("total number of decoded chars: %d\n", index\_char);

printf("decoded chars: ");

for (int i = 0; i < index\_char; i++)

printf("%c", decoded\_str[i]);

printf("\n");

return index\_char;

}

For example 0101100 with below Huffman binary tree can be decoded into a b c

|  |  |  |  |
| --- | --- | --- | --- |
| index | Bit\_stream | Step 1 | Step 2 |
| i=0 | 0 | Move to left child | Char ‘a’ is added to decoded\_str |
| i=1 | 1 | Move to right child | Node has no char data |
| i=2 | 0 | Move to left child | Node has no char data |
| i=3 | 1 | Move to right child | Char ‘b’ is added to decoded\_str |
| i=4 | 1 | Move to right child | Node has no char data |
| i=5 | 0 | Move to left child | Node has no char data |
| i=6 | 0 | Move to left child | Char ‘c’ is added to decoded\_str |

[result]

1)input : abacdebaf

**텍스트이(가) 표시된 사진

자동 생성된 설명**

2) input : ab

**텍스트이(가) 표시된 사진

자동 생성된 설명**