REPORT

Data Structures

Student ID : 1771008
Name : Minjeong Kim

Homework6_1

변수(variable) 분석

element	
Type	**name**
int	key

HeapType	
Type	**name**
element*	heap
int	heap_size

Heap은 배열로 이루어진다. 이때 heap 배열에 대한 정보를 담는 포인터의 역할로 HeapType 구조체를 선언하고, heap을 element의 배열로 저장한다.

함수(function) 분석

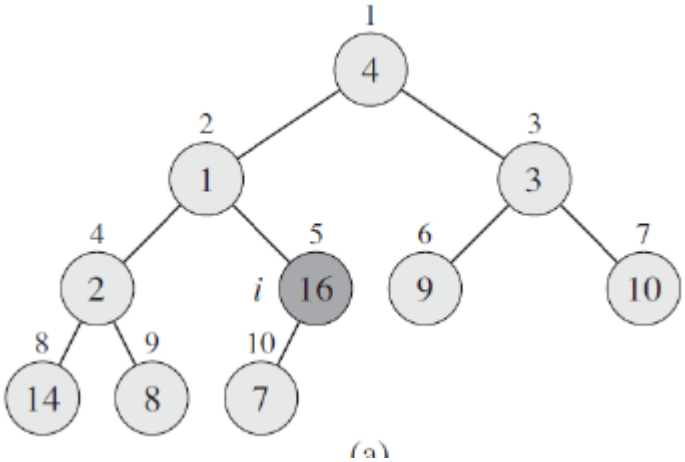
void insert_max_heap(HeapType *h, element item)
max_heap은 부모노드의 key가 자식노드의 key보다 크다는 특징을 갖고있다. 따라서
heap배열의 가장 마지막 index에 item을 삽입한후, 그 부모노드 (index/2)와의 크기 비교를한다.
이때 부모노드의 값이 더 작을 경우, max_heap이 아니게 되므로, 부모노드의 위치를 자식노드의 위치로 옮기고, 자신노드인 item은 부모노드의 위치로 이동한다.

element delete_max_heap(HeapType *h)
max_heap에서 최대값, 즉 최상위 노드를 빼고 그 값을 리턴한다. 최상위 노드가 삭제되기 때문에, 최상위 노드(index=1)보다 한단계 작은 노드가 최상위 노드로 갔을 때의 max_heap을 재 구현 해야 한다. 이떄, 부모에서 자식으로 처음부터 정렬하게되면 코드가 비효율적이다. 따라서 heap배열의 가장 마지막 index에 위치하는 (즉, 제일 작은 key를 갖는) node를 최상위 노드로 설정하고, 아래의 자식 노드(i*2 / i*2+1)와 비교해서 이동한다.(이때 위치를 swap 하는 자식 노드는 둘 중 max값을 갖는 것으로 한다.)

void cmp(HeapType *h ,int pind)
max_heap을 정렬할 때, delete_max_heap과 같은 원리가 적용된다. 두 자식 노드의 값을 비교하고, 두 자식 노드 중 큰 값의 자식노드와 부모 노드의 값을 비교한후, 자식 노드가 부모 노드보다 더 크다면 위치를 바꾼다. 이렇게 하기 위해, 자식노드 두 개의 key 값을 비교하는 함수를 분리하였다. 이때 parameter로 받는 pind는 비교해야 하는 부모 노드의 index값이다.

void change(HeapType *h, int pind)

build_max_heap을 사용할 때, 자식 노드에서 부모 노드로 올라가면서 비교한다.



build_max_heap의 기본 원리는 아래와 같다.

heap_size/2인 index부터 1까지 비교한다. 따라서 위 그림에서는 1,2,3,4,5의 index만 비교하면 된다.

비교하는 순서는 아래와 같다.

5 -> 4 -> 3-> 2 -> 1

5 : 5-10 비교 후 swap 선택

4 : 8-9 max값 찾기 위해 cmp 후 cmp 값과 4 비교 후 swap 선택

3 : 6-7 max값 찾기 위해 cmp 후 cmp 값과 3 비교 후 swap 선택

2 : 4-5 max값 찾기 위해 cmp 후 cmp 값과 2 비교 후 swap 선택 // 그리고 4, 5 단계 다시 반복

1 : 2-3 max값 찾기 위해 cmp 후 cmp 값과 1 비교 후 swap 선택 // 그리고 2, 3, 4, 5 단계 다시 반복

본인은 이것의 recursion의 형태를 갖고있다고 생각했다.

따라서 recursion을 구현할 change 함수를 선언하고 조건을 만족할 때 recursion이 일어난다

1 . 먼저 cmp를 이용해 자식노드 중 max값을 찾는다.

2 . 이때 자식 노드가 부모 노드(pind로 찾는다.)보다 클 경우 max_heap 만족을 위해 자리를 swap한다.

3 . recursion의 조건은 자리를 바꾼 자식 노드가 leaf 노드가 아닐 때이다. 따라서 자식노드의 index값인 max_i가 input_size/2 보다 작을 경우(leaf node가 아닌 경우) change 함수를 호출한다.

void build_max_heap(HeapType *h)

단순히 build_max_heap으로 recursion을 안 한 이유는, heap_size가 홀수인지 짝수인지에 따라 right child node의 유무가 결정되기 때문이다. 따라서 build_max_heap을 하기위한 5번의 반복중에 첫번째 반복일 때만, 홀수, 짝수 여부를 판단한 후 build_max_heap 안에서 자체적으로 swap이 이뤄지도록하고 나머지 2~5번의 반복에서는 change함수의 recursion을 이용한다.

void heap_sort(HeapType *h, element *a, int n)

main에서 생성한 random한 키 값을 가진 정렬되지 않은 heap을 정렬하기 위해 build_max_heap을 호출하고, 이후 output array에는 작은 값에서 큰 값으로 저장하기 위해 for문에서 heap의 가장 마지막 index에서 0까지 I 값을 설정하고, delete_max_heap을 이용해서 a의 요소로 넣는다.

bool check_sort_results(element *output, int n)

heap_sort를 한 array가 작은 값부터 큰 값으로 올바르게 정렬되었는지 확인하기 위해, 비교한다. 이후 올바른지 여부를 return한다. (true = 1 / false = 0)

main()

1. HeapType 선언하기, heap 배열 공간 할당하기, output 배열 공간 할당하기
   (이때 heap은 index가 1부터 시작한다는 점을 주의한다.)
2. random을 이용해서 heap 배열의 key값 무작위로 설정하기
3. heap_sort()함수 사용 -> sub process : build_max_heap + change + cmp
4. sorted data 출력하기 : output배열의 key값 출력
5. check_sort_results로 heap_sort()가 올바른지 확인하기.

전체 코드(full code)

```
1   // heapsort.cpp : Defines the entry point for the console application.
2   //
3
4   #include "stdlib.h"
5   #include "stdio.h"
6   #include "string.h"
7   #include "time.h"
8
9    #define MAX_ELEMENT 2000
10  typedef struct element {
11      int key;
12  } element;
13
14  typedef struct {
15      element *heap;
16      int heap_size;
17  } HeapType;
18
19   // Integer random number generation function between 0 and n-1
20  int random(int n)
21  {
22      return rand() % n;
23  }
24
25   // Initialization
26  void init(HeapType *h) {
27      h->heap_size = 0;
28  }
29
30   // Insert the item at heap h, (# of elements: heap_size)
31  void insert_max_heap(HeapType *h, element item)
32  {
33      int i;
34      i = ++(h->heap_size);
35
36      // The process of comparing with the parent node as it traverses the tree
37      while ((i != 1) && (item.key > h->heap[i / 2].key)) {
38          h->heap[i] = h->heap[i / 2];
39          i /= 2;
```

```
40      }
41      h->heap[i] = item; // Insert new node
42  }
43
44   // Delete the root at heap h, (# of elements: heap_size)
45  element delete_max_heap(HeapType *h)
46  {
47      int parent, child;
48      element item, temp;
49
50      item = h->heap[1];
51      temp = h->heap[(h->heap_size)--];
52      parent = 1;
53      child = 2;
54      while (child <= h->heap_size) {
55          // Find a smaller child node
56          if ((child < h->heap_size) &&
57              (h->heap[child].key) < h->heap[child + 1].key)
58              child++;
59          if (temp.key >= h->heap[child].key) break;
60          // Move down one level
61          h->heap[parent] = h->heap[child];
62          parent = child;
63          child *= 2;
64      }
65      h->heap[parent] = temp;
66      return item;
67  }
68
69   int input_size = 10;    //10, 100, 1000
70
71  int cmp(HeapType *h, int pind) {
72      if (h->heap[pind * 2].key > h->heap[pind * 2 + 1].key)
73          return pind * 2;
74      else
75          return pind * 2 + 1;
76  }
77
78  void change(HeapType *h, int pind) {
79      int max_i = cmp(h, pind);
80      if (h->heap[pind].key < h->heap[max_i].key) {
81          int tmp;
82          tmp = h->heap[pind].key;
83          h->heap[pind].key = h->heap[max_i].key;
84          h->heap[max_i].key = tmp;
85
86          if (max_i <= input_size/2) {
87              change(h, max_i);
88          }
89      }
90  }
91
```

```
92  void build_max_heap(HeapType *h)
93  {
94      int max_i;
95      for (int i = h->heap_size/2; i > 0; i--) {
96          if (i==h->heap_size / 2 && h->heap_size % 2 == 0) {
97              max_i = h->heap_size;
98              if (h->heap[i].key < h->heap[i].key) {
99                  int tmp;
100                 tmp = h->heap[i].key;
101                 h->heap[i].key = h->heap[max_i].key;
102                 h->heap[max_i].key = tmp;
103             }
104         }
105         else {
106             change(h, i);
107         }
108     }
109 }
110
111 //input: heap 'h'
112 //output: sorted element array 'a'
113 void heap_sort(HeapType *h, element *a, int n)
114 {
115     int i;
116
117     build_max_heap(h);
118
119     for (i = (n - 1); i >= 0; i--) {
120         a[i] = delete_max_heap(h); //output
121     }
122 }
123
124 bool check_sort_results(element *output, int n)
125 {
126     bool index = 1;
127     for (int i = 0; i<n - 1; i++)
128         if (output[i].key > output[i + 1].key)
129         {
130             index = 0;
131             break;
132         }
133     return index;
134 }
135
136 void main()
137 {
138     time_t t1;
139     //Intializes random number generator
140     srand((unsigned)time(&t1));
141
142
143     int data_maxval = 10000;
144
145     HeapType *h1 = (HeapType *)malloc(sizeof(HeapType));
```

```
146     // 'heap' is allocated according to 'input_size'.  heap starts with 1, so 'input_
147     h1->heap = (element *)malloc(sizeof(element)*(input_size + 1));
148
149     // output: sorted result
150     element *output = (element *)malloc(sizeof(element)*input_size);
151
152     // Generate an input data randomly
153     for (int i = 0; i < input_size; i++)
154         h1->heap[i + 1].key = random(data_maxval);  // note) heap starts with 1.
155
156     //int arr[] ={4, 1, 3, 2, 16, 9, 10, 14, 8, 7};
157     //for (int i = 0; i < input_size; i++) {
158     //  h1->heap[i + 1].key = arr[i];
159     //}
160
161     h1->heap_size = input_size;
162
163     if (input_size < 20) {
164         printf("Input data\n");
165         for (int i = 0; i < input_size; i++)    printf("%d\n", h1->heap[i + 1].key);
166         printf("\n");
167     }
168
169     // Perform the heap sort
170     heap_sort(h1, output, input_size);
171
172     if (input_size < 20) {
173         printf("Sorted data\n");
174         for (int i = 0; i < input_size; i++)    printf("%d\n", output[i].key);
175         printf("\n");
176     }
177
178     // Your code should pass the following function (returning 1)
179     if (check_sort_results(output, input_size))
180         printf("Sorting result is correct.\n");
181     else
182         printf("Sorting result is wrong.\n");
183 }
```

Homework6_2

변수(variable) 분석

input_huff	
Type	**name**
char * [array]	data
int * [array]	freq
int	size

huffman 코드의 data와 freq 값을 저장해두는 구조체
이 구조체 값을 tree와 heap의 요소값에 넣는 역할을 한다.

TreeNode	
Type	**name**
char	data
int	key
int[MAX_BIT]	bits
int	bit_size
TreeNode *	l
TreeNode *	r

huffman 코드의 freq에 따른 data값에 대한 tree를 생성할 때 사용한다
처음에, freq 값만 있던 단말노드들을 min값부터 차례로 merge해서
tree로 엮는 작업을 할 예정이다.
다음과 같은 tree를 만드는 이유는 encoding과 decoding 때문이다.

bits_stream	
Type	**name**
int *	stream
int *	length

tree로 문자와 하나씩 대응시킨 bit값을 이용해서
input으로 주어진 문자열을 bit로 바꾼 결과를 저장할 때 사용한다.

element	
Type	**name**
TreeNode *	ptree
int	key

HeapType	
Type	**name**
element[MAX_ELT]	heap
int	heap_size

heap을 사용하는 이유는, tree로 freq값들의 두개의 min값을 찾을 때, search에 걸리는 시간을 줄이기 위해서이다.
즉, heap을 사용하면 단순한 for문으로 찾는 것 보다 시간 복잡도가 더 적게 나오기 때문이다.

int **m_LUT
int *m_bit_size
int m_char_size = 6;
문자와 bit를 하나씩 대응하기 위해서 사용되는 전역 변수이다.
m_LUT : 각각의 row가 각각의 문자와 대응한다.
m_bit_size: m_LUT의 n번째 row에 있는 문자열의 길이는 m_bit_size의 n번째 인덱스와 대응한다.
m_char_size : bit와 대응시키는 문자의 개수 (여기서는 a,b,c,d,e,f 총 6개)

함수(function) 분석

void insert_min_heap(HeapType *h, element item)
element delete_min_heap(HeapType *h)
위 homework 6_1에서 했기 때문에 생략

TreeNode *make_tree(TreeNode *left, TreeNode *right)
parameter의 link값 과 대응되는 새로운 TreeNode 할당 후 return

element huffman_tree(input_huff *huff)
Huffman code에 대응되는 tree를 만들기 위한 함수이다.

1. Huffman tree 초기 설정
Huffman code에 해당하는 문자의 빈도수에 따라 tree를 만들기 위해, 문자(tree->data)값을 가진 leaf node를 생성하기 위한 작업이다. 따라서 Huffman code의 크기만큼 반복문을 실행한다.

먼저 leaf node를 만드는 것이기 때문에 child node가 null인 node를 생성한다.
이후, 해당 node의 값들을 하나씩 초기화한다. data에 Huffman code의 문자를 넣고, key에 Huffman code의 frequency를 넣는다. 그리고, bits와 bit_size는 모두 0으로 초기화 한다.

node의 key값이 제일 작은 두 값을 비교해서 merge하는 식으로 tree가 생성되기 때문에, min값을 빠르게 찾기 위해 heap을 이용할 예정이다. 따라서, element를 선언하고, element의 요소의 ptree에는 tree에 넣기 위해 만든 node를 저장하고, key값으로 huffman code의 frequency를 지정한다. 이후, heap에서는 이 element의 key값을 이용해서, min값을 찾을 것이다. min값으로 heap이 정렬되어야 하기 때문에 insert_min_heap()을 호출한다.

2. key값이 작은 두개의 값을 비교한 후 merge
delete_min_heap을 이용해서, key값 (= huffman의 frequency)이 작은 element를 변수로 지정한다.
이후, 이 두 elt의 key값이 합쳐진 노드를 생성하기 위해 make_tree를 이용해서 node를 만들고, node와 heap의 key값을 두 elt의 합으로 설정한다. 이때, 이 node는 data를 가진 노드가 아니라 Huffman_tree를 형성하기 위한 중간 node이므로 data는 null값을 갖는다.
결과적으로 두개의 elt 값을 child node로 갖는 parent node가 생성되고, heap에는 delete 2번 insert 1번으로 인해, heap_size가 하나 줄어든다.

3. 남은 heap element는 root node
leaf node의 개수만큼 반복을 한 후에는, heap의 함수 호출 delete 2번 insert 1번으로 인해, 6개의 node의 key값이 합쳐진 root node만이 남겨지게 된다. 이 root node를 return하여, 생성한 huffmantree에 접근할 수 있게 한다.

void Huffman_traversal(TreeNode *node)

huffman tree를 이용해서, data를 가진 각 leaf node의 bit를 알 수 있다.
이를 위해서 left의 node의 추가될 bit값은 0, right node의 추가될 bit값은 1을 이용하였다. 이때 recursion을 사용해서 모든 node에 올바르게 접근할 수 있게 하였다.

1. 도달한 node가 leaf node일 때,
   TreeNode의 배열에 해당 node를 저장한다.
2. 도달한 node가 leaf node가 아닐 때
   2-1. 도달한 node의 left값이 있을 때
   left node의 bit값을 설정한다. 먼저 현재 node의 bit값을 복사하고, left이므로 새 bit값으로 0을 추가한다.
   이때 left node의 bit_size도 현재 node의 bit_size에 1을 더해서 갱신해준다.
   그리고 recursion을 사용해서 node->l에 접근한다.
   2-2. 도달한 node의 right값이 있을 때
   right node의 bit값을 설정한다. 먼저 현재 node의 bit값을 복사하고, right이므로 새 bit값으로 1을 추가한다.
   이때 right node의 bit_size도 현재 node의 bit_size에 1을 더해서 갱신해준다.
   그리고 recursion을 사용해서 node->r에 접근한다.

결론적으로 TreeNode의 배열인 arr[]에 data값에 해당하는 bits와 bit_size를 가진 leaf node들이 저장되어있다.

void sort_leaf()

Huffman_traversal() 함수를 이용해서 arr[]에 bits와 bit_size를 가진 leaf node들이 저장되었지만, 이들의 순서는 a, b, c, d, e, f 처럼 순서대로가 아니다. 따라서 이것을 순서대로 정렬한 결과를 m_LUT 배열과 m_bit_size에 최종적으로 반영하기 위해서 sort_leaf()함수를 추가하였다.
먼저, data를 순서대로 정렬하기 위해서 a,b,c,d,e,f 각각의 character가 대응되는 int값을 갖는다는 점을 이용하였다. 따라서, data를 key값으로 삼아 insert_min_heap함수를 이용해서, data값이 작은 것부터 정렬되는 즉, a부터 f까지 정렬되는 heap을 생성한다.
이후 delete_min_heap함수를 사용해서 작은 data값을 가진 node부터 접근해서, 각 node의 bit_size와 bit를 각각 m_bit_size와 m_LUT에 저장한다.

void print_codeword()

sort_leaf()를 이용해서 a부터 f까지 순차적으로 정렬된 m_LUT와 m_bit_size를 이용해서, 각 문자 데이터의 비트값을 출력한다.

void huffman_encoding(char *str, bits_stream *bits_str)

입력으로 주어진 string을 bit로 encoding하기위한 함수이다.
string의 각 char에 접근하고, 각 char에 맞는 bits를 bits_str->stream에 저장하고, 해당 bit_size를 bits_str->length에 저장한다. 이때, 접근한 각 char에 맞는 bit값을 대응시키기위해 m_LUT과 m_bit_size를 사용해야 하는데, 이때 char이 int로도 바뀔 수 있다는 점을 이용해서 m_LUT과 m_bit_size의 index를 찾았다.
a = 97, b = 98, c = 99, d = 100, e = 101, f = 102이고, m_LUT와 m_bit_size에는 각각 0, 1, 2, 3, 4, 5 index에 저장되어 있기 때문에 , idx = str[i] – 97 을이용해서 char data와 bit의 대응을 찾는다.
이렇게 찾은 idx값을 이용해서, stream과 length를 저장한다.

void Huffman_decoding(bits_stream *bits_str, TreeNode *node, char *decoded_str)

주어진 bit code를 char로 decoding하기 위한 함수이다.

먼저, decoding 방법은 이전에 생성한 Huffman_tree를 이용한다. 하나씩 읽는 bit값에 따라 Huffman_tree의 root node에서 차례로 bit에 맞는 child node로 이동한다 (0일경우 left, 1일경우 right) 이렇게 이동한 node가 leafnode일 경우에는, 그 leafnode가 가진 char data값을 output string에 저장하고, Huffman_tree의 root에서 새로운 search를 시작한다.

output string의 길이와, 새로 추가할 char의 위치를 지정하기위해서 count를 이용한다.

main()

1. input_huff 타입의 변수 할당 : Huffman_code의 data와 freq, size를 모두 저장한 변수
2. m_LUT, m_bit_size배열에 공간을 할당한다.
3. huff에 따라 tree를 생성하기 위해 huffman_tree()함수를 사용하고, 이 Huffman_tree의 root값을 저장한다.
4. huffman_traversal() 함수를 이용해서, leaf node가 각 huffman_code에 맞는 bit값을 가진 tree로 수정한다.
   그리고 leaf_node만 모아둔 sorting이 안된 배열 arr을 생성한다.
5. sort_leaf()를 이용해서, arr배열의 node들을 data에 맞게 a부터 f까지 정렬하고,
   정렬된 결과를 m_LUT, m_bit_size에 넣는다.
6. print_codeword()를 이용해서, huffman_code의 data의 bit값을 출력한다.
7. input string 설정 및 출력
8. bits_stream 타입의 변수를 생성하고, 공간을 할당한다.
9. input string의 character값을 huffman_code의 bit값으로 encoding한 결과를 bit_str에 저장하기위해
   huffman_encoding()함수를 이용한다.
10. encoding한 bit_str을 다시 decoding하기 위해 huffman_decoding 함수를 이용한다.
    이 함수는, bit string에 맞는 character string을 return하기 위해 huffman_tree를 사용한다.

전체 코드(full code)

```c

#include "stdlib.h"
#include "stdio.h"
#include "string.h"

 #define MAX_ELEMENT 1000
 #define MAX_BIT     10
 #define MAX_CHAR    20

 // Input data for huffman code
typedef struct input_huff {
    char *data;     // Character array (a ~ f)
    int *freq;      // Frequency array
    int size;       // Number of characters
} input_huff;

 // Structure for huffman binary tree
typedef struct TreeNode {
    char data;          // Character (a ~ f)
    int key;            // Frequency
    int bits[MAX_BIT];  // Huffman codeword
    int bit_size;       // Huffman codeword's size
    struct TreeNode *l; // Left child of huffman binary tree
    struct TreeNode *r; // Right child of huffman binary tree
} TreeNode;
```

```c
// Structure for bits stream
typedef struct bits_stream {
    int *stream;
    int length;
} bits_stream;

// Elements used in the heap
typedef struct element {
    TreeNode *ptree;
    int key;    // frequency of each character
} element;

// Heap
typedef struct HeapType {
    element heap[MAX_ELEMENT];
    int heap_size;
} HeapType;

int **m_LUT, *m_bit_size;
int m_char_size = 6;

// Initialization
void init(HeapType *h)
{
    h->heap_size = 0;
}
//
int is_empty(HeapType *h)
{
    if (h->heap_size == 0)
        return true;
    else
        return false;
}

void insert_min_heap(HeapType *h, element item)
{
    int i;
    i = ++(h->heap_size);

    // compare it with the parent node in an order from the leaf to the root
    while ((i != 1) && (item.key < h->heap[i / 2].key)) {
        h->heap[i] = h->heap[i / 2];
        i /= 2;
    }
    h->heap[i] = item; // Insert new node
}

element delete_min_heap(HeapType *h)
{
    int parent, child;
    element item, temp;
    item = h->heap[1];
    temp = h->heap[(h->heap_size)--];
    parent = 1;
    child = 2;
```

```c
    while (child <= h->heap_size) {
        if ((child < h->heap_size) && (h->heap[child].key) > h->heap[child + 1].key)
            child++;
        if (temp.key <= h->heap[child].key) break;
        h->heap[parent] = h->heap[child];
        parent = child;
        child *= 2;
    }
    h->heap[parent] = temp;
    return item;
}

// Node generation in binary tree
TreeNode *make_tree(TreeNode *left, TreeNode *right)
{
    TreeNode *node = (TreeNode *)malloc(sizeof(TreeNode));
    if (node == NULL) {
        fprintf(stderr, "Memory allocation error\n");
        exit(1);
    }
    node->l = left;
    node->r = right;
    return node;
}

// Binary tree removal
void destroy_tree(TreeNode *root)
{
    if (root == NULL) return;
    destroy_tree(root->l);
    destroy_tree(root->r);
    free(root);
}

// Huffman code generation
element huffman_tree(input_huff *huff)
{
    int i;
    TreeNode *node, *x;
    HeapType heap;
    element e, e1, e2;
    init(&heap);

    int n = huff->size;

    for (i = 0; i<n; i++) {
        node = make_tree(NULL, NULL); //branch가 null인 tree node 생성
        e.ptree = node;
        node->data = huff->data[i];
        e.key = node->key = huff->freq[i]; //key: frequency
        memset(node->bits, 0, sizeof(int)*MAX_BIT);
        //메모리 시작 주소, 메모리 채우고 싶은 값, 채우고자하는 바이트 수
        node->bit_size = 0;
        //treenode의 값을 모두 0으로 초기화하고,
        //이 treenode를 elt와 연결지어, 이 elt를 heqp에 삽입

        insert_min_heap(&heap, e); //key값 즉 frequency에 따른 min_heap이 생성된다.
        //단말노드를 생성했다고 생각하자!
    }
```

```
142
143     for (i = 1; i<n; i++) {
144         // Delete two nodes with minimum values
145         e1 = delete_min_heap(&heap);
146         e2 = delete_min_heap(&heap); //key값 비교하면 e1<e2
147
148         // Merge two nodes
149         x = make_tree(e1.ptree, e2.ptree); //x는 tree node
150         e.ptree = x;
151         x->data = NULL;
152         e.key = x->key = e1.key + e2.key;
153         memset(x->bits, 0, sizeof(int)*MAX_BIT);
154         x->bit_size = 0;
155         //두개의 node를 합친 중간노드 생성해서 heap에 넣기
156
157         insert_min_heap(&heap, e);
158     }
159
160     e = delete_min_heap(&heap); // Final Huffman binary tree
161     //insert1번 delete2번을 하니까 heap에 들어있는 node의 수가 6에서 1로 줄고,
162     //그 1개는 maxkey를 갖는 노드가 된다.
163
164     return e;
165     //  destroy_tree(e.ptree);
166 }
167
168
169 // Generate the huffman codeword from the huffman binary tree
170 // Hint: use the recursion for tree traversal
171 // input: root node
172 // output: m_LUT, m_bit_size
173 int count =0;
174 TreeNode *arr[MAX_CHAR];
175
176 void huffman_traversal(TreeNode *node)
177 {
178     //m_LUT에 숫자를 부여한 것들을 저장한다.
179     //m_bit_size 각 부여한 byte들의 길이를 잰다.
180
181     while (node != NULL) {
182         //abcdef 순서로 m_LUT m_bit_size
183         if (node->l == NULL & node->r == NULL) {
184             arr[count++] = node;
185             break;
186         }
187         else {
188             if (node->l != NULL) {
189                 for (int i = 0; i < node->bit_size; i++) {
190                     node->l->bits[i] = node->bits[i];
191                 }
192                 node->l->bits[node->bit_size] = 0;
193                 node->l->bit_size = node->bit_size + 1;
194                 huffman_traversal(node->l);
195             }
196
```

```
197             if (node->r != NULL) {
198                 for (int i = 0; i < node->bit_size; i++) {
199                     node->r->bits[i] = node->bits[i];
200                 }
201                 node->r->bits[node->bit_size] = 1;
202                 node->r->bit_size = node->bit_size + 1;
203                 huffman_traversal(node->r);
204             }
205             break;
206         }
207     }
208 }
209
210 void sort_leaf() {
211     HeapType h;
212     init(&h);
213
214     for (int i = 0; i < m_char_size; i++) {
215         element e;
216         e.key = arr[i]->data;
217         e.ptree = arr[i];
218         insert_min_heap(&h, e);
219     }
220
221     for (int i = 0; i < m_char_size; i++) {
222         element e;
223         e = delete_min_heap(&h);
224         m_bit_size[i] = e.ptree->bit_size;
225         for (int j = 0; j < m_bit_size[i]; j++) {
226             m_LUT[i][j] = e.ptree->bits[j];
227         }
228     }
229 }
230
231 int **mem_2D_int(int row, int col)
232 {
233     int **m2 = (int **)malloc(sizeof(int *)*row);
234     for (int i = 0; i<row; i++)
235         m2[i] = (int *)malloc(sizeof(int)*col);
236     return m2;
237 }
238
239 void print_codeword()
240 {
241     printf("* Huffman codeword\n");
242     for (int i = 0; i < m_char_size; i++)
243     {
244         switch (i) {
245         case 0:
246             printf("%c: ", 'a');
247             break;
248         case 1:
249             printf("%c: ", 'b');
250             break;
251         case 2:
252             printf("%c: ", 'c');
253             break;
254         case 3:
255             printf("%c: ", 'd');
256             break;
```

```
257             case 4:
258                 printf("%c: ", 'e');
259                 break;
260             case 5:
261                 printf("%c: ", 'f');
262                 break;
263             }
264
265             for (int j = 0; j < m_bit_size[i]; j++)
266                 printf("%d", m_LUT[i][j]);
267
268             printf("\n");
269         }
270
271     }
272
273
274
275     // Input: 'str'
276     // Output: 'bits_stream' (consisting of 0 or 1)
277     // 'bits_stream' is generated using 'm_LUT' generated by the huffman binary tree
278     // Return the total length of bits_stream
279     void huffman_encoding(char *str, bits_stream *bits_str)
280     {
281         for (int i = 0; i < strlen(str); i++) {
282             int idx = str[i] - 97; //a값의 int형은 97이고, m_LUT에는 index 0에 들어있기 때문에
283             for (int j = 0; j < m_bit_size[idx]; j++) {
284                 bits_str->stream[bits_str->length + j] = m_LUT[idx][j];
285             }
286             bits_str->length += m_bit_size[idx];
287         }
288
289         printf("\n* Huffman encoding\n");
290         printf("total length of bits stream: %d\n", bits_str->length);
291         printf("bits stream: ");
292         for (int i = 0; i < bits_str->length; i++)
293             printf("%d", bits_str->stream[i]);
294         printf("\n");
295     }
296
297     // input: 'bits_stream' and 'total_length'
298     // output: 'decoded_str'
299     int huffman_decoding(bits_stream *bits_str, TreeNode *node, char *decoded_str)
300     {
301         int count = 0;
302         TreeNode *cur = node;
303         for (int i = 0; i < bits_str->length; i++) {
304             if (bits_str->stream[i] == 0)
305                 cur = cur->l;
306             else
307                 cur = cur->r;
308             if (cur->l == NULL and cur->r == NULL) {
309                 decoded_str[count++] = cur->data;
310                 cur = node;
311             }
312         }
313
```

```
314     printf("\n* Huffman decoding\n");
315     printf("total number of decoded chars: %d\n", count);
316     printf("decoded chars: ");
317     for (int i = 0; i < count; i++)
318         printf("%c", decoded_str[i]);
319     printf("\n");
320
321     return (count);
322 }
323
324 void main()
325 {
326     char data[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
327     int freq[] = { 45, 13, 12, 16, 9, 5 };
328
329     input_huff *huff1 = (input_huff *)malloc(sizeof(input_huff));
330     huff1->data = data; //data array
331     huff1->freq = freq; //freq array
332     huff1->size = m_char_size; //6
333
334     // m_LUT: each row corresponds to the codeword for each character
335     // m_bit_size: 1D array of codeword size for each character
336     // For instance, a = 0, b = 101, ...
337     // 1st row of 'm_LUT': 0 0 ... 0
338     // 2nd row of 'm_LUT': 1 0 1 ...0
339     // m_bit_size = {1, 3, ...}
340     m_LUT = mem_2D_int(m_char_size, MAX_BIT); //2차원 배열 설정 6X10 //2차원포인터
341     m_bit_size = (int *)malloc(sizeof(int)*m_char_size); //1차원 포인터 //크기가 6인 배열 bit
342
343     // Generate the huffman binary tree on heap
344     // 'element_root': element containing the root node
345     element element_root = huffman_tree(huff1);
346
347     // Generate the huffman codeword from the huffman binary tree
348     huffman_traversal(element_root.ptree);
349
350     sort_leaf();
351
352     //printf out the huffman codeword
353     print_codeword();
354
355     //example of input data
356     char str[MAX_CHAR] = { "abacdebaf" };
357     //char str[MAX_CHAR] = { "ab" };
358     char decoded_str[MAX_CHAR];
359
360     printf("\n* input chars: ");
361     for (int i = 0; i < strlen(str); i++)
362         printf("%c", str[i]);
363     printf("\n");
364
365     //start encoding
366     bits_stream *bits_str1 = (bits_stream *)malloc(sizeof(bits_stream));
367     bits_str1->stream = (int *)malloc(sizeof(int)*MAX_BIT * MAX_CHAR);
368     memset(bits_str1->stream, -1, sizeof(int)*MAX_BIT * MAX_CHAR);
369     bits_str1->length = 0;
370
371     huffman_encoding(str, bits_str1);
372
373     int decoded_char_length = huffman_decoding(bits_str1, element_root.ptree, decoded_str);
374
375 }
```