

Data structures

Assignment 06

Technical Report

2176211 Jiwon Yang

I. Problem 1

1. Code Explanation

```
2. // heapsort.cpp : Defines the entry point for the console application.
3. //
4.
5. //#include "stdafx.h"
6. #include "stdlib.h"
7. #include "stdio.h"
8. #include "string.h"
9. #include "time.h"
10.
11. #define MAX_ELEMENT 2000
12. typedef struct element{
13.     int key;
14. } element;
15.
16. typedef struct {
17.     element *heap; //element 형 배열인 heap.
18.     int heap_size;
19. } HeapType;
```

- element: A structure representing an element with an integer key.
- HeapType: A structure representing a heap with an array of elements (heap) and the current heap size (heap_size).

```
20. // Integer random number generation function between 0 and n-1
21. int random(int n)
22. {
23.     return rand() % n;
24. }
25.
26. // Initialization
27. void init(HeapType *h) {
28.     h->heap_size = 0;
29. }
30.
31. // Insert the item at heap h, (# of elements: heap_size)
32. void insert_max_heap(HeapType *h, element item)
33. {
34.     int i;
35.     i = ++(h->heap_size);
36.
37.     // The process of comparing with the parent node as it traverses the tree
38.     while ((i != 1) && (item.key > h->heap[i / 2].key)) {
39.         h->heap[i] = h->heap[i / 2];
40.         i /= 2;
```

```

41.     }
42.     h->heap[i] = item; // Insert new node
43. }

```

- insert_max_heap: Inserts an element into the max heap and maintains the heap property.

```

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. }

```

- delete_max_heap: Deletes the root element (maximum) from the max heap and maintains the heap property.

```

68. int find_bigger_child(int index, HeapType *h){
69.     if (index*2 <= h->heap_size){
70.         int child_index = index*2;
71.         if (h->heap[child_index].key < h->heap[child_index+1].key)
72.             child_index++;
73.         return child_index;
74.     }
75.     return -1;
76. }
77.
78. void swap(element *node1, element *node2){
79.     int temp = node1->key;
80.     node1->key = node2->key;
81.     node2->key = temp;
82. }
83.

```

```

84. void recur_func(int index, HeapType *h) {
85.     int bigger_child = find_bigger_child(index, h);
86.
87.     if (bigger_child == -1 || h->heap[index].key >= h-
        >heap[bigger_child].key)
88.         return;
89.     swap(&h->heap[index], &h->heap[bigger_child]);
90.     recur_func(bigger_child, h);
91.
92. }
93. void build_max_heap(HeapType *h)
94. {
95.     int half = h->heap_size/2 + 1;
96.     for (int i = half; i >= 1; i--){
97.         recur_func(i, h);
98.     }
99. }
100.
101. //input: heap 'h'
102. //output: sorted element array 'a'
103. void heap_sort(HeapType *h, element *a, int n)
104. {
105.     int i;
106.
107.     build_max_heap(h); //이걸로 max heap 먼저 만들고,
108.     for (i = (n - 1); i >= 0; i--) {
109.         a[i] = delete_max_heap(h); //하나씩 꺼내서 뒤에서부터 저장하면
            크기순 정렬됨
110.     }
111. }
112.
113. bool check_sort_results(element *output, int n) //element 형 배열
    output. (그냥 heap 으로 받지..?)
114. {
115.     bool index = 1;
116.     for(int i=0; i<n-1; i++)
117.         if (output[i].key > output[i + 1].key)
118.         {
119.             index = 0; //한 번이라도 대소관계
                어긋나면 0 반환
120.             break;
121.         }
122.     return index; //전부 대소관계 맞으면 1 반환
123. }

```

- find_bigger_child: Finds the index of the bigger child of a node in the max heap.
- recur_func: Recursively fixes the max heap property starting from a given index.
- build_max_heap: Builds a max heap from the given array of elements.

```

124.     int main()
125.     {
126.         clock_t start, end;
127.         double result;
128.
129.         int input_size = 10;    //10, 100, 1000
130.         int data_maxval = 1000;
131.
132.         HeapType *h1 = (HeapType *)malloc(sizeof(HeapType));
133.         // 'heap' is allocated according to 'input_size'. heap starts
        with 1, so 'input_size+1' is used.
134.         h1->heap = (element *)malloc(sizeof(element)*(input_size + 1));
135.
136.         // output: sorted result
137.         element *output = (element *)malloc(sizeof(element)*input_size);
138.
139.
140.
141.         // Generate an input data randomly
142.         for (int i = 0; i < input_size; i++)
143.             h1->heap[i+1].key = random(data_maxval);    // note) heap
        starts with 1.
144.         h1->heap_size = input_size;
145.
146.         if (input_size < 20){
147.             printf("[Input data]\n");
148.             for (int i = 0; i < input_size; i++)
149.                 printf("%d\n", h1->heap[i + 1].key);
150.             printf("\n");
151.         }
152.
153.         start = clock();
154.
155.         // Perform the heap sort
156.         heap_sort(h1, output, input_size);
157.
158.         end = clock();
159.
160.         if (input_size < 10000) {
161.             printf("\n\n[Sorted data]\n");
162.             for (int i = 0; i < input_size; i++)
163.                 printf("%d\n", output[i].key);
164.             printf("\n");
165.         }
166.
167.         // Your code should pass the following function (returning 1)
168.         if(check_sort_results(output, input_size))
169.             printf("Sorting result is correct.\n");

```

```

170.         else
171.             printf("Sorting result is wrong.\n");
172.
173.             result = ((double)(end - start)) / CLOCKS_PER_SEC; // Convert to
seconds
174.             printf("time passed: %f seconds\n\n", result);
175.
176.             return 0;
177.     }

```

2. Experimental Result

(1) input_size = 10; int data_maxval = 100000;

◆ Result : (passed time : 0.00000sec)

(2) Input_size = 100; int data_maxval = 100000;

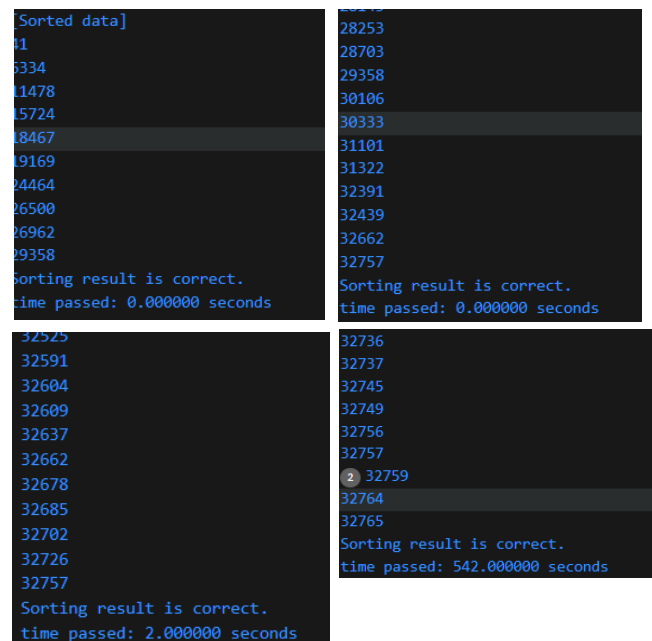
◆ Result: (passed time : 0.00000sec)

(3) Input_size = 1000; int data_maxval = 100000;

◆ Result : (passed time : 2.00000sec)

(4) Input_size = 10000; int data_maxval = 100000;

◆ Result : (passed time : 542.00000sec)



The four terminal screenshots show the following results:

- Top-left:** Sorted data: 41, 5334, 11478, 15724, 18467, 19169, 24464, 26500, 26962, 29358. Time passed: 0.000000 seconds.
- Top-right:** Sorted data: 28253, 28703, 29358, 30106, 30333, 31101, 31322, 32391, 32439, 32662, 32757. Time passed: 0.000000 seconds.
- Bottom-left:** Sorted data: 32525, 32591, 32604, 32609, 32637, 32662, 32678, 32685, 32702, 32726, 32757. Time passed: 2.000000 seconds.
- Bottom-right:** Sorted data: 32736, 32737, 32745, 32749, 32756, 32757, 32759, 32764, 32765. Time passed: 542.000000 seconds.

3. Code Analysis

When using insert_max_heap to insert elements one by one, each insertion likely takes $O(n \log n)$ time. This is because, in the worst case scenario, when inserting one element at the very end of the heap, we might need to move the newly inserted node up the heap for as many levels as the height of the heap, which is $\log n$.

On the other hand, using build_max_heap to construct the max heap has a time complexity of $O(n)$. This is because, when constructing a heap in a random order, from the second half of the heap to the last node, the max heap property is already satisfied. In other words, traversing backward from the second half to the first node and adjusting nodes to satisfy the max heap property is sufficient.

In summary, while `insert_max_heap` may take $O(n \log n)$ time for each insertion, `build_max_heap` ensures a more efficient construction of the max heap in $O(n)$ time. This is due to the fact that it optimally takes advantage of the random initial order of the heap, reducing the overall time complexity.

II. Problem 2

1. Code Explanation

```
2. // #include "stdafx.h"
3. #include "stdlib.h"
4. #include "stdio.h"
5. #include "string.h"
6.
7. #define MAX_ELEMENT 1000
8. #define MAX_BIT 10
9. #define MAX_CHAR 20
10.
11. // Input data for huffman code
12. typedef struct input_huff {
13.     char *data; // Character array (a ~ f)
14.     int *freq; // Frequency array
15.     int size; // Number of characters
16. } input_huff;
17.
18. // Structure for huffman binary tree
19. typedef struct TreeNode {
20.     char data; // Character (a ~ f)
21.     int key; // Frequency
22.     int bits[MAX_BIT]; // Huffman codeword
23.     int bit_size; // Huffman codeword's size
24.     struct TreeNode *l; // Left child of huffman binary tree
25.     struct TreeNode *r; // Right child of huffman binary tree
26. } TreeNode;
27.
28. // Structure for bits stream
29. typedef struct bits_stream {
30.     int *stream;
31.     int length;
32. } bits_stream;
33.
34. // Elements used in the heap
35. typedef struct element {
36.     TreeNode *ptree;
37.     int key; // frequency of each character
38. } element;
39.
40. // Heap
41. typedef struct HeapType {
42.     element heap[MAX_ELEMENT];
43.     int heap_size;
44. } HeapType;
```


input_huff: Structure to hold input data for Huffman coding.

TreeNode: Structure representing nodes in the Huffman binary tree.

HeapType: Structure representing a heap for Huffman tree construction.

element: Structure representing elements used in the heap.

bits_stream: Structure representing a stream of bits.

```
45.int **m_LUT, *m_bit_size;
46.int m_char_size = 6;
47.
48.// Initialization
49.void init(HeapType *h)
50.{
51.    h->heap_size = 0;
52.}
53.
54.int is_empty(HeapType *h)
55.{
56.    return h->heap_size == 0;
57.}
58.
59.void insert_min_heap(HeapType *h, element item)
60.{
61.    int i = ++(h->heap_size);
62.
63.    // compare it with the parent node in an order from the leaf to the root
64.    while ((i != 1) && (item.key < h->heap[i / 2].key)) {
65.        h->heap[i] = h->heap[i / 2];
66.        i /= 2;
67.    }
68.    h->heap[i] = item; // Insert new node
69.}
70.
71.element delete_min_heap(HeapType *h)
72.{
73.    int parent, child;
74.    element item, temp;
75.
76.    item = h->heap[1];
77.    temp = h->heap[(h->heap_size)--];
78.
79.    parent = 1;    child = 2;
80.    while (child <= h->heap_size) {
81.        if ((child < h->heap_size) && (h->heap[child].key > h->heap[child +
            1].key))
82.            child++;
```

```

83.     if (temp.key <= h->heap[child].key) break;
84.     h->heap[parent] = h->heap[child];
85.     parent = child;
86.     child *= 2;
87. }
88. h->heap[parent] = temp;
89. return item;
90.}
91.
92.// Node generation in binary tree
93.TreeNode *make_tree(TreeNode *left, TreeNode *right)
94.{
95.    TreeNode *node = (TreeNode *)malloc(sizeof(TreeNode));
96.    if (node == NULL) {
97.        fprintf(stderr, "Memory allocation error\n");
98.        exit(1);
99.    }
100.    node->l = left;
101.    node->r = right;
102.    return node;
103.}

```

(5) make_tree: Creates a new tree node with left and right children.

```

104.
105.    // Binary tree removal
106.    void destroy_tree(TreeNode *root)
107.    {
108.        if (root == NULL) return;
109.        destroy_tree(root->l);
110.        destroy_tree(root->r);
111.        free(root);
112.    }
113.
114.    // Huffman code generation
115.    element huffman_tree(input_huff *huff)
116.    {
117.        int i;
118.        TreeNode *node, *x;
119.        HeapType heap;
120.        element e, e1, e2;
121.        init(&heap);
122.
123.        int n = huff->size;
124.
125.        for (i = 0; i < n; i++) {
126.            node = make_tree(NULL, NULL);
127.            e.ptree = node;
128.            node->data = huff->data[i];
129.            e.key = node->key = huff->freq[i];

```

```

130.         memset(node->bits, 0, sizeof(int) * MAX_BIT);
131.         node->bit_size = 0;
132.
133.         insert_min_heap(&heap, e);
134.     }
135.
136.     for (i = 1; i < n; i++) {
137.         // Delete two nodes with minimum values
138.         e1 = delete_min_heap(&heap);
139.         e2 = delete_min_heap(&heap);
140.
141.         // Merge two nodes
142.         x = make_tree(e1.ptree, e2.ptree);
143.         e.ptree = x;
144.         x->data = '\0'; // Update: set data as a null character
145.         e.key = x->key = e1.key + e2.key;
146.         memset(x->bits, 0, sizeof(int) * MAX_BIT);
147.         x->bit_size = 0;
148.
149.         insert_min_heap(&heap, e);
150.     }
151.     e = delete_min_heap(&heap); // Final Huffman binary tree
152.
153.     return e;
154.     // destroy_tree(e.ptree);
155. }

```

(6) huffman_tree: Generates a Huffman binary tree from input data.

```

156.     // Generate the huffman codeword from the huffman binary tree
157.     // Hint: use the recursion for tree traversal
158.     // input: root node
159.     // output: m_LUT, m_bit_size
160. void huffman_traversal(TreeNode *node, int code[], int code_size)
161. {
162.     if (node == NULL) {
163.         return;
164.     }
165.
166.     if (node->l == NULL && node->r == NULL) {
167.         // Leaf node (contains data)
168.         int index = node->data - 'a'; // Assuming the data is a
lowercase letter
169.
170.         // Copy the code to m_LUT
171.         for (int i = 0; i < code_size; i++) {
172.             m_LUT[index][i] = code[i]; // a 면 0 번째줄, b 면 첫번째줄,
c 면 두번째줄...
173.         }
174.

```

```

175.         // Set the bit_size for the data
176.         m_bit_size[index] = code_size;
177.     }
178.
179.     // Traverse left with '0'
180.     code[code_size] = 0;
181.     huffman_traversal(node->l, code, code_size + 1);
182.
183.     // Traverse right with '1'
184.     code[code_size] = 1;
185.     huffman_traversal(node->r, code, code_size + 1);
186. }

```

(7) huffman_traversal: Recursive traversal of the Huffman binary tree to generate codewords.

```

187. int **mem_2D_int(int row, int col)
188. {
189.     int **m2 = (int **)malloc(sizeof(int *) * row);
190.     for (int i = 0; i < row; i++)
191.         m2[i] = (int *)malloc(sizeof(int) * col);
192.     return m2;
193. }
194.
195. void print_codeword()
196. {
197.     printf("* Huffman codeword\n");
198.     for (int i = 0; i < m_char_size; i++) {
199.         switch (i) {
200.             case 0:
201.                 printf("%c: ", 'a');
202.                 break;
203.             case 1:
204.                 printf("%c: ", 'b');
205.                 break;
206.             case 2:
207.                 printf("%c: ", 'c');
208.                 break;
209.             case 3:
210.                 printf("%c: ", 'd');
211.                 break;
212.             case 4:
213.                 printf("%c: ", 'e');
214.                 break;
215.             case 5:
216.                 printf("%c: ", 'f');
217.                 break;
218.         }
219.
220.         for (int j = 0; j < m_bit_size[i]; j++)
221.             printf("%d", m_LUT[i][j]);

```

```

222.
223.         printf("\n");
224.     }
225. }
226.
227.     // Input: 'str'
228.     // Output: 'bits_stream' (consisting of 0 or 1)
229.     // 'bits_stream' is generated using 'm_LUT' generated by the huffman
    binary tree
230.     // Return the total length of bits_stream
231. void huffman_encoding(char *str, bits_stream *bits_str)
232. {
233.     int interval = 0;
234.
235.     for (int i = 0; i < strlen(str); i++) {
236.         int index = str[i] - 'a';
237.         for (int j = 0; j < m_bit_size[index]; j++) {
238.             bits_str->stream[interval++] = m_LUT[index][j];
239.         }
240.     }
241.
242.     bits_str->length = interval;
243.
244.     printf("\n* Huffman encoding\n");
245.     printf("total length of bits stream: %d\n", interval);
246.     printf("bits stream: ");
247.     for (int i = 0; i < interval; i++)
248.         printf("%d", bits_str->stream[i]);
249.     printf("\n");
250. }
251.
252.     // input: 'bits_stream' and 'total_length'
253.     // output: 'decoded_str'
254. int huffman_decoding(bits_stream *bits_str, TreeNode *node, char
    *decoded_str)
255. {
256.     int index_char = 0;
257.     TreeNode *current_node = node;
258.
259.     for (int i = 0; i < bits_str->length; i++) {
260.         if (bits_str->stream[i] == 0) {
261.             current_node = current_node->l;
262.         }
263.         else {
264.             current_node = current_node->r;
265.         }
266.
267.         if (current_node->l == NULL && current_node->r == NULL) {

```

```

268.         // Leaf node, found a character
269.         decoded_str[index_char++] = current_node->data;
270.         current_node = node; // Reset to the root for the next
    character
271.     }
272. }
273.
274.     decoded_str[index_char] = '\0';
275.
276.     printf("\n* Huffman decoding\n");
277.     printf("total number of decoded chars: %d\n", index_char);
278.     printf("decoded chars: %s\n", decoded_str);
279.
280.     return index_char;
281. }

```

(8) huffman_encoding: Encodes a given string using Huffman codewords.

(9) huffman_decoding: Decodes a given bit stream using the Huffman binary tree.

```

282.     int main()
283.     {
284.         char data[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
285.         int freq[] = { 45, 13, 12, 16, 9, 5 };
286.
287.         input_huff *huff1 = (input_huff *)malloc(sizeof(input_huff));
288.         huff1->data = data;
289.         huff1->freq = freq;
290.         huff1->size = m_char_size;
291.
292.         // m_LUT: each row corresponds to the codeword for each character
293.         // m_bit_size: 1D array of codeword size for each character
294.         // For instance, a = 0, b = 101, ...
295.         // 1st row of 'm_LUT': 0 0 ... 0
296.         // 2nd row of 'm_LUT': 1 0 1 ... 0
297.         // m_bit_size = {1, 3, ...}
298.         m_LUT = mem_2D_int(m_char_size, MAX_BIT);
299.         m_bit_size = (int *)malloc(sizeof(int) * m_char_size);
300.
301.         // Generate the huffman binary tree on heap
302.         // 'element_root': element containing the root node
303.         element element_root = huffman_tree(huff1);
304.
305.         // Generate the huffman codeword from the huffman binary tree
306.         int code[MAX_BIT];
307.         huffman_traversal(element_root.ptree, code, 0);
308.
309.         //printf out the huffman codeword
310.         print_codeword();
311.

```

```

312.         //example of input data
313.         char str[MAX_CHAR] = { "abacdebaf" };
314.         char decoded_str[MAX_CHAR];
315.
316.         printf("\n* input chars: ");
317.         for (int i = 0; i < strlen(str); i++)
318.             printf("%c", str[i]);
319.         printf("\n");
320.
321.         //start encoding
322.         bits_stream *bits_str1 = (bits_stream
    *)malloc(sizeof(bits_stream));
323.         bits_str1->stream = (int *)malloc(sizeof(int) * MAX_BIT *
    MAX_CHAR);
324.         memset(bits_str1->stream, -1, sizeof(int) * MAX_BIT * MAX_CHAR);
325.         bits_str1->length = 0;
326.
327.         huffman_encoding(str, bits_str1);
328.
329.         //start decoding
330.         int decoded_char_length = huffman_decoding(bits_str1,
    element_root.ptree, decoded_str);
331.
332.         return 0;
333.     }
334.

```

2. Experimental Result

```

a: 0
b: 101
c: 100
d: 111
e: 1101
f: 1100
total length of bits stream: 23
bits stream: 01010100111110110101100
total number of decoded chars: 9
decoded chars: abacdebaf

```

```

* Huffman codeword
a: 0
b: 101
c: 100
d: 111
e: 1101
f: 1100

* input chars: abacdebaf

* Huffman encoding
total length of bits stream: 23
bits stream: 01010100111110110101100

* Huffman decoding
total number of decoded chars: 9
decoded chars: abacdebaf

```

Execution result

3. Code Analysis

The overall time complexity of Huffman tree construction is $O(n \log n)$. The main loop in `huffman_tree()` iterates n times, where n is the number of characters. In each iteration, `insert_min_heap()` is called, which has a time complexity of $O(\log n)$. Thus, the overall time complexity

is $O(n \log n)$.

The function `huffman_traversal()` performs a recursive traversal of the Huffman tree to generate codewords. Each traversal involves visiting a node and appending a bit to the codeword. In the worst case, each node is visited once, making it a linear time operation. ($O(n)$)

The function `huffman_encoding()` converts input characters into Huffman codewords. It linearly scans each character in the input string, and for each character, it appends the corresponding codeword bits to the bit stream. Its time complexity is $O(m)$, where m is the length of the input string.