Assignment03

목차

1. 선택한 자료구조

Huffman code를 구현하기 위해서는 트리의 구조가 필요하기 때문에, 이번 Huffman code를 구현하기 위한 적절한 자료구조로 새로운 구조체를 선언하여 만든 linked list의 형태가 적합하다고 판단하였다. 이를 통하여 구현한 트리를 통해 encoding한 후 결과를 출력하는 것이 적절하다고 판단되어 새로운 구조체 node를 선언하였다. node의 구조는 다음과 같다.

```
6   typedef struct node* nodePointer;
7   typedef struct node{
8       char character;
9       int frequency;
10      int leaf;
11      nodePointer left, right; // for tree
12      nodePointer next; // for linked list
13  } node;
```

이 구조체는 총 6개의 element를 포함한다. 각각의 element가 담고 있는 정보는 다음과 같다.

char character: 어떤 문자인지에 대한 정보인지를 담고 있는 변수

int frequency: 입력으로 받은 파일에서 해당 문자가 나타나는 횟수를 담은 변수

int leaf: 해당 노드가 트리 상에서 리프 노드로 사용되는지의 정보를 담는 변수

nodePointer left, right: 해당 노드가 트리로 사용될 때 본인의 좌, 우 자식 노드의 주소

nodePointer next: 해당 노드가 linked list로 사용될 때 다음 노드를 가리키는 주소

노드의 포인터를 편하게 지정하기 위해 node*를 nodePointer로 새로 타입 선언을 해주었다.

2. main()

변수 선언 및 각 변수들의 데이터 타입은 다음과 같다.

```
21      // variables
22      int count, num, freq, len, depth;
23      int hash[128] = {0};
24      char character;
25      char* encode[128];
26      nodePointer list, tree, temp, temp2;
27      nodePointer* nodes;
28      nodePointer* queue;
```

파일을 여는 부분은 다음과 같다. 입력 파일인 hw3_input.txt와 결과 파일인 hw3_output1.txt 및 hw3_output3.txt을 stdio.h 헤더파일의 fopen 함수를 통해 각각 열어준 후, 이에 대한 파일 디스크립터를 받았다.

```
30      // open file
31      FILE* inputFile = fopen("hw3_input.txt", "r");
32      FILE* outputFile1 = fopen("hw3_output1.txt", "w");
33      FILE* outputFile2 = fopen("hw3_output2.txt", "w");
```

다음으로는 입력 파일로부터 파일의 내용물을 읽어오는 과정이다.

```
35      // read from input file
36      while ((character = fgetc(inputFile)) != EOF) hash[character] += 1;
```

여기에서 Hash table의 개념을 적용하였다. 입력 파일의 조건이 0~127의 ASCII 코드 값을 가지게 된다는 명확한 범위가 존재하므로, 각각의 아스키코드의 출현 빈도를 저장하는 길이가 128인 int형 배열 hash라는 변수를 선언하여, 이 안에 해당 배열의 인덱스를 ASCII 코드 번호로 갖는 문자의 출현 빈도를 저장하게 된다. 가령 A의 출현 빈도가 12회일 경우, A의 ASCII 코드는 65에 대응되기 때문에, hash[65]의 값이 12가 되게 된다.

다시 코드로 돌아와, 앞의 변수 선언 부분에서 해시 테이블의 선언과 함께 값이 모두 0으로 초기화되어 있으므로, 입력 문자를 하나씩 읽어가며 빈도를 늘려준다. 해당 문자가 나올 때마다 그 문자의 ASCII 코드 값을 인덱스로 가지는 hash 배열의 값을 1씩 늘여주면서, 파일을 전부 읽었을 경우 해시 테이블은 각 ASCII 코드 별 입력 문서 내에서의 출현 빈도를 가지게 된다.

다음은 Huffman code를 위한 트리를 만들기 위한 전처리 과정이다.

```
38      // preprocessing for making tree
39      count = 0;
40      for (int i = 0; i < 128; i++) {
41          if (hash[i]) count++;
42      }
43
44      num = 0;
45      nodes = (nodePointer*) malloc(sizeof(nodePointer) * count);
46      for (int i = 0; i < 128; i++) {
47          if (hash[i] > 0) nodes[num++] = makeNode(i, hash[i], NULL, NULL);
48      }
```

윗부분의 for 루프의 경우 빈도가 0이 아닌 문자의 개수를 세서, 등장한 총 문자의 개수를 정수형 변수 count 내에 저장한다. 밑의 for 루프의 경우 구해진 총 문자의 개수를 바탕으로 등장한 문자만큼의 노드를 만드는 과정이다. 이를 통하여 만들어진 노드들을 nodes라는 배열 내에 ASCII 코드 값을 기준으로 오름차순 정렬된 배열을 만들어 낸다. 노드들을 만들 때 문자와 빈도를 각각 노드의 character와 frequency element에 저장하고, 아직 트리 구조를 이루고 있지 않기에 좌우 자식 노드를 가리키는 포인터는 전부 NULL로 설정하였다.

이후 ASCII 코드 값을 기준으로 정렬된 배열 nodes를 출현 빈도를 기준으로 오름차순이 되도록 재정렬을 진행한다. 여기서 사용된 정렬 알고리즘은 버블 정렬이다.

```
50      // sort nodes by frequency in ascending order
51      for (int i = 0; i < num - 1; i++) {
52          for (int j = 0; j < num - i - 1; j++) {
53              if (nodes[j]->frequency > nodes[j + 1]->frequency) {
54                  temp = nodes[j];
55                  nodes[j] = nodes[j + 1];
56                  nodes[j + 1] = temp;
57              }
58          }
```

이 때 정렬 알고리즘인 버블 정렬은 stable한 정렬 알고리즘이기 때문에 같은 빈도를 가질 경우 기존의 값 배열 상태를 그대로 가지고 가게 되고, 따라서 동일 빈도를 갖는 문자들의 경우 ASCII 코드 값을 기준으로 오름차순 정렬을 유지하게 된다.

빈도수를 기준으로 오름차순 재정렬을 진행한 이후, 아래와 같이 linked list를 구성한다.

```
61      // make linked list
62      for (int i = 0; i < count - 1; i++) {
63          nodes[i]->next = nodes[i + 1];
64      }
65      list = nodes[0];
```

Huffman code는 트리 생성을 하는 과정에서 기존의 노드들을 자식 노드로 갖는 새로운 노드들이 발생한다. 이 노드들이 발생할 때마다 그 노드 또한 본인의 자식 노드들의 빈도를 더한 값을 빈도로 갖고, 배열에 들어갈 경우 정렬된 상태를 유지하기 위하여 적절한 자리를 찾아야 한다. 이 과정에서 적절한 자리를 찾더라도 그 자리 이후의 노드들을 전부 옆으로 밀어내야 하게 되는데, 이 과정에서 들게 되는 밀어내기의 비용을 없애기 위해 배열로 정리되어 있던 노드들을 linked list로 연결하게 된다. 이미 정렬되어 있기 때문에 앞 노드의 next element를 바로 다음 노드의 주소로 연결해주기만 하면 되는 과정을 거치게 된다. 이렇게 linked list로 될 경우 새로 발생할 노드를 안에 집어넣는 과정에서 이후 노드들을 모두 밀어내는 과정을 거치지 않고 단순하게 주소 지정만 변경해주는 식으로 하면 되기 때문에 밀어내기의 비용이 발생하지 않는다.

이렇게 linked list를 구성하고 난 이후, 이 리스트를 오름차순으로 따라가며 encoding을 위한 트리를 구성하게 된다. 트리 구성 부분의 코드는 다음과 같다.

```
76      // make tree
77      while(list->next) {
78          freq = list->frequency + (list->next)->frequency;
79          temp = makeNode(0, freq, list, list->next);
80          temp->leaf = 0;
81
82          list = list->next;
83          temp2 = list;
84          while (temp2->next) {
85              if (temp2->next->frequency < temp->frequency) temp2 = temp2->next;
86              else break;
87          }
88          temp->next = temp2->next;
89          temp2->next = temp;
90          list = list->next;
91
92          /*
93          // print for checking tree 1
94          temp2 = list;
95          while (temp2) {
96              printf("%c %d -> ", temp2->character, temp2->frequency);
97              temp2 = temp2->next;
98          }
99          printf("\n");
100         */
101     }
102     tree = list;
```

Linked list가 오름차순으로 이미 정렬되어 있기에, 이 중 가장 앞의 2개를 하나의 새로운 노드로 묶어주는 과정을 반복한다. 새로운 노드의 빈도는 합치려고 하는 두 노드의 빈도를 더한 값이 새로운 노드의 빈도로 들어가게 된다. 빈도를 계산한 이후 노드 생성 과정을 거치게 되는데, 새로 생성되는 노드는 임의로 문자 정보에는 garbage value인 0을 갖고 빈도는 새로 계산된 빈도를 가지게 되며, 왼쪽 자식 노드에 더 빈도가 적은 노드를 연결하고 오른쪽 노드에 나머지 노드가 연결한 형태가 된다. 또한 이 노드는 새로 생성된 노드이기에 한 문자의 정보만 가진 노드가 아닌 복합적으로 여러 노드들의 정보를 담고 있는 노드이기 때문에, 리프노드가 아님을 표시하기 위해 기본적으로 1로 설정되어 있던 leaf element를 0으로 바꿔주는 과정을 거치게 된다.

노드를 생성한 이후 생성된 노드를 다시 linked list에 정렬 순서가 유지되도록 삽입하는 과정을 거친다. Linked list를 맨 앞에서부터 탐색해 가며, 현재 위치 노드의 다음 노드의 빈도가 새로 삽입하려는 노드의 빈도보다 크면 현재 위치에 노드를 삽입하고, 아닐 경우 현재 위치를 다음 노드로 변경하는 과정을 반복하며 적절한 자리를 찾아 새로 생긴 노드를 적절한 위치에 연결한다. 연결이 완료된 이후에, 새 노드를 만드는 데에 사용했던 두 노드들을 linked list에서 끊어버린다.

이 과정을 linked list의 노드가 단 하나만 남을 때까지 진행한다. 마지막으로 남게 된 노드는 결국 모든 노드를 자식 노드로 갖는 트리의 루트 노드가 되기 때문에, 이 마지막으로 남은 노드를 트리의 루트 노드를 가리킬 포인터 tree에 할당하게 되며 트리 구성을 완료한다.

다음 과정으로는 encoding 및 결과 출력을 위한 트리의 최대 깊이를 찾는 과정이다.

```
119     // find max depth of tree(root is depth 0)
120     char* treeString = TreeToString(tree);
121     num = 0;
122     depth = 0;
123     len = strlen(treeString);
124     for (int i = 0; i < len; i++) {
125         if (treeString[i] == '(') num++;
126         else if (treeString[i] == ')') num--;
127         if (depth < num) depth = num;
128     }
129     //printf("%s\n", treeString);
130     //printf("%d\n", depth);
```

TreeToString 함수를 통해 트리를 최종 출력 형태로 스트링화한 다음 이 결과를 통해 최대 depth를 구하게 된다. 여기서 깊이는 앞에서부터 읽었을 때 (가 나온 횟수에서)가 나온 횟수를 뺀 값에 해당하게 되므로, 이 점을 이용하여 최대 깊이를 구한다.

최대 깊이를 구한 이후 encoding을 하는 과정은 다음과 같다.

```
132     // encoding
133     for (int i = 0; i < 128; i++) {
134         encode[i] = (char*) malloc(sizeof(char) * (depth + 1));
135         encode[i][0] = '\0';
136     }
137     encoding(tree, encode, "", depth);
```

encode 배열 또한 해시 테이블과 비슷한 원리에서 착안하였으며, 해당 인덱스를 ASCII 코드로 갖는 문자의 encoding 결과를 담을 배열이다. Encoding 결과의 최대 길이는 트리의 최대 깊이에 1을 더한 값과 같으므로('₩0'이 존재하기 때문) 그 사이즈만큼 메모리를 할당한 이후 초기화로는 아무런 값을 가지지 않는 스트링으로 초기화 해준다.

Encoding 과정은 encoding() 함수를 통하여 모든 과정이 이루어지기 때문에, 이 과정에 대한 설명은 뒤의 encoding() 함수의 설명 파트에서 후술하겠다.

Encoding이 완료된 이후 결과를 출력하는 과정을 가진다. 코드는 다음과 같다.

```
139     // print hw3_output2
140     for (int i = 0; i < 128; i++) {
141         if (strlen(encode[i])) fprintf(outputFile2, "%c: %s\n", i, encode[i]);
142     }
143
144     fseek(inputFile, 0, SEEK_SET);
145     while ((character = fgetc(inputFile)) != EOF) fprintf(outputFile2, "%c", character);
146     fprintf(outputFile2, "\n");
147
148     // print hw3_output1
149     fprintf(outputFile1, "%s\n", treeString);
150     fseek(inputFile, 0, SEEK_SET);
151     while ((character = fgetc(inputFile)) != EOF) fprintf(outputFile1, "%s", encode[character]);
152     fprintf(outputFile1, "\n");
```

ASCII 코드의 인코딩 결과를 담은 배열 encode를 순회하며 확인해가면서, encoding 결과가 길이가 0이 아닐 경우 변환 과정이 진행되었기 때문에, 변환된 코드를 hw3_output2.txt에 적게 된다. 이후 입력 파일의 seek point를 파일의 맨 앞으로 다시 설정하여, 입력 파일의 내용을 그대로 다시 hw3_output2.txt에 저장한다.

다음으로 hw3_output1.txt에는 스트링으로 변환한 트리의 구조를 우선 저장한 이후, 다시 입력 파일의 seek point를 파일의 맨 앞으로 다시 설정하여 문자를 하나씩 읽으며 변환한 문서의 전체 encoding 결과를 저장한다.

```
154     // memory deallocation
155     for (int i = 0; i < 128; i++) free(encode[i]);
156     freeTree(tree);
157     free(treeString);
158     free(nodes);
159
160     // close file
161     fclose(inputFile);
162     fclose(outputFile1);
163     fclose(outputFile2);
```

마지막으로 malloc() 함수를 통해 할당받은 메모리들을 모두 해제한 이후 파일을 닫으며 main 함수의 종료와 함께 전체 프로그램을 종료한다.

3. nodePointer makeNode()

makeNode 함수는 입력받은 파라미터들을 기준으로 새로운 노드를 생성하여 이 노드를 반환해주는 역할을 하는 함수이다. 인자들은 다음과 같다.

char character: 문자에 대한 정보를 담는 변수

int frequency: 해당 문자의 출현 빈도를 담는 변수

nodePointer left: 해당 노드가 트리로 사용될 때 본인의 왼쪽 자식 노드의 주소

nodePointer right: 해당 노드가 트리로 사용될 때 본인의 오른쪽 자식 노드의 주소

함수의 내용은 다음과 같다.

```
167 nodePointer makeNode(char character, int frequency, nodePointer left, nodePointer right) {
168     nodePointer temp;
169     temp = (nodePointer) malloc(sizeof(node));
170     temp->character = character;
171     temp->frequency = frequency;
172     temp->leaf = 1;
173     temp->left = left;
174     temp->right = right;
175     temp->next = NULL;
176     return temp;
```

함수의 구성은 간단하다. 노드의 초기 설정을 진행하는 부분이라 생각하면 된다. 먼저 노드를 생성한 이후에 메모리를 할당하고, 입력 받은 인자들을 바탕으로 문자 정보, 빈도, 좌우 자식 노드의 포인터를 각각의 element에 할당한다. 리프 노드임을 확인하는 변수 leaf를 1로 설정하고, linked list에서 쓰일 변수 next를 NULL로 초기화한 후 이 노드를 반환하면서 함수를 종료한다.

4. char* TreeToString()

이 함수는 트리를 받으면 재귀적인 방식을 통하여 트리의 구조를 스트링의 형태로 바꿔 표현해주는 함수이다. 입력받은 노드가 NULL일 경우엔 비어있는 스트링을 반환하게 하여 재귀 호출의 종료 부분을 담당하고 있으며, NULL이 아닐 경우 리프 노드일 경우와 아닐 경우의 두 가지로 나누어 스트링으로 변경하는 과정을 거친다. 트리 내의 노드를 탐색하는 과정은 재귀적인 방식을 통한 너비 우선 탐색(BFS)의 방식을 사용하였다.

리프 노드일 경우, 본인 노드의 문자만 적은 스트링을 반환하며 재귀 호출의 종료 조건에 해당하도록 설정한다. 만일 리프 노드가 아닐 경우, 왼쪽 자식 노드와 오른쪽 자식 노드로부터 각각 스트링으로 변환된 값을 얻어온 후 이 값을 이어붙이고, 그 붙인 결과 값을 반환하며 함수를 종료한다.

```
179  char* TreeToString(nodePointer root) {
180      if (!root) {
181          char* emptyString = (char*)malloc(1);
182          emptyString[0] = '\0';
183          return emptyString;
184      }
185
186      char* leftStr = TreeToString(root->left);
187      char* rightStr = TreeToString(root->right);
188
189      int size = strlen(leftStr) + strlen(rightStr) + 4; // for '(', ',', ')', '\0'
190      char* result = (char*) malloc(size);
191
192      if (root->leaf) snprintf(result, size, "%c", root->character);
193      else snprintf(result, size, "(%s,%s)", leftStr, rightStr);
194
195      free(leftStr);
196      free(rightStr);
197
198      return result;
199  }
```

5. void encoding()

encoding() 함수 또한 재귀적으로 구현된 함수로, 트리와 encoding 결과를 저장할 배열 그리고 인코딩 중간 결과를 저장하는 변수, 그리고 트리의 최대 깊이의 4개의 변수를 받아서 encoding 결과를 저장하게 해주는 함수이다. 노드의 순회 방식은 재귀적으로 구현된 너비 우선 탐색 방식(BFS)이다.

Encoding의 방식은 간단하다. 왼쪽 자식 노드로 갈 경우 0을 붙이고, 오른쪽 자식 노드로 갈 경우 1을 뒤에 붙이는 과정을 반복한다. 이 과정을 반복하다 리프 노드를 만나게 될 경우 지금까지의 0과 1을 이용하여 만들어 놓은 encoding 결과를 encoding 결과 저장 배열의 해당되는 인덱스 안에 저장한다.

```
201  void encoding(nodePointer root, char** arr, char* str, int depth) {
202      if (root->leaf) {
203          strcpy(arr[root->character], str);
204      }
205      else {
206          char* left = (char*) malloc(sizeof(char) * (depth + 1));
207          char* right = (char*) malloc(sizeof(char) * (depth + 1));
208
209          strcpy(left, str);
210          strcpy(right, str);
211
212          strcat(left, "0");
213          strcat(right, "1");
214
215          encoding(root->left, arr, left, depth);
216          encoding(root->right, arr, right, depth);
217
218          free(left);
219          free(right);
220      }
221  }
```

6. void freeTree()

freeTree 함수는 단순하게 트리 내의 모든 루프를 탐색하면서 모든 노드의 메모리를 해제해주는 함수이다. 이 또한 재귀적인 방식을 통해 구현된 함수이며, 코드는 다음과 같다.

```
223   void freeTree(nodePointer root) {
224       if (root == NULL) return;
225       freeTree(root->left);
226       freeTree(root->right);
227       free(root);
228   }
```

7. 시간 복잡도 분석

시간 복잡도의 분석 결과는 다음과 같다. 입력의 크기를 n, 입력 문서 내에 출현하는 단어들의 종류의 개수를 v라 가정하겠다.

1. 입력 파일로부터 내용 읽어오는 부분

   입력 파일로부터 모든 내용을 한 번씩 순회하며 값을 읽어들인 후 해시 테이블에 빈도를 바로 저장하기 때문에, 이 부분에서의 시간 복잡도는 $O(n)$이다.

2. 트리 생성을 위한 전처리 과정: 등장하는 문자 확인 및 노드 생성 부분

   해시 테이블의 크기는 128로 고정된 상수 값이며, 이 값은 입력의 크기에 영향을 받지 않는다. 따라서 이 해시 테이블을 한번 순회하며 등장하는 문자들을 확인하기 때문에 이 과정은 입력의 크기에 영향받지 않는 과정이고, 따라서 이 과정의 시간 복잡도는 $O(1)$이다.

3. 노드들을 빈도를 기준으로 오름차순 정렬하는 부분

   오름차순 정렬을 사용하는 부분에서 버블 정렬을 사용하였기 때문에 얼핏 보면 시간 복잡도가 $O(n^2)$일 것 같지만, 여기서 정렬해야 할 요소들의 개수는 최대 128개라는 상수의 상한선이 존재한다. 이 정렬 부분은 입력의 크기가 아닌 '출현하는 단어의 종류의 수'에 비례하게 되기 때문에, 시간 복잡도는 이의 제곱에 비례하게 되므로 시간 복잡도는 $O(v^2)$에 속하게 된다.

4. 오름차순 정렬된 배열을 Linked list로 변환하는 부분

   정렬된 배열을 처음부터 끝까지 순회하며 연결 과정을 진행하였기 때문에 시간 복잡도가 $O(n)$일 것 같지만, 여기서 연결해야 할 요소들의 개수는 최대 128개라는 상수의 상한선이 존재한다. 이 연결 부분 또한 입력의 크기가 아닌 '출현하는 단어의 종류의 수'에 비례하기 때문에, 이 부분의 시간 복잡도는 $O(n)$에 속하게 된다.

5. Linked list를 트리의 형태로 변환하는 부분

   Linked list를 트리의 형태로 바꾸는 부분 또한 입력의 크기에 비례하지 않고, '출현하는

단어의 종류의 수'에 비례하게 된다. 한 번의 노드 연결 작업으로 서브 트리를 생성하는 과정은 2개의 서브 트리를 linked list에서 빼서 새 서브 트리를 제작하고 나온 결과 서브 트리를 다시 linked list에 집어넣는 과정을 반복하기 때문에, 결과적으로 서브 트리 하나가 줄어드는 과정이므로 이 linked list에 남은 서브 트리가 하나가 될 때까지 작업하는 과정은 v에 선형으로 비례하는 시간 복잡도를 갖게 된다. 따라서 이 과정의 시간 복잡도는 O(v)에 속하게 된다.

6. 트리의 구조를 문자열로 정리하여 나타내는 부분 및 최대 깊이를 구하는 부분

   트리 구조를 문자열로 변환하는 과정 또한 모든 노드를 한 번씩 순회하되 너비 우선 탐색 방식을 사용하는 방식을 통해 순회하는데, 여기서 노드의 개수는 v에 비례하게 되므로 시간 복잡도는 v에 선형으로 비례하는 시간 복잡도를 가지게 된다. 따라서 시간 복잡도는 O(v)이다.

7. Encoding 과정

   Encoding 과정 또한 모든 노드를 한 번씩 순회하되 너비 우선 탐색 방식을 사용하여 순회하는데, 이 또한 노드의 개수에 선형으로 비례하고 노드는 v에 선형으로 비례하기 때문에 이 과정의 시간 복잡도는 O(v)에 속하게 된다.

8. hw3_output2.txt 파일에 저장하는 부분

   이 과정은 전체 해시 테이블을 한 번씩 순회하며 encoding 결과가 존재하는 부분만 파일에 출력하는 부분과 입력 파일의 내용을 그대로 복사해주는 부분이 존재한다. 전자의 과정은 시간 복잡도가 입력 크기와 상관없이 항상 128개의 배열을 전부 순회하기 때문에 상수 시간을 가지며, 입력 파일의 내용 복사 부분은 입력 파일의 크기에 비례하기 때문에 O(n)에 속하게 된다. 따라서 이 과정의 전체 시간 복잡도는 O(n)에 속하게 된다.

9. hw3_output1.txt 파일에 저장하는 부분

   이 과정은 트리를 스트링으로 형식화한 값이 이미 구해져 있는 상태에서 이 스트링을 파일에 저장하는 부분과 파일 입력 파일 전체를 인코딩한 결과를 저장하는 부분에 존재하게 된다. 이 과정의 시간 복잡도는 O(v + n)에 속하게 된다.

10. 메모리 해제 부분

    메모리 해제 부분의 경우 트리 내의 모든 노드들에 대해 한번씩 순회하며 메모리 해제를 진행하기 때문에 이 과정은 O(v)에 속하게 된다.

따라서 이 모든 과정을 종합해 보았을 때 전체 과정의 시간 복잡도는 $O(n + v^2)$에 속하게 되고, n과 v^2 중 더욱 지배적인 항을 따라서 시간 복잡도가 결정되게 된다.