자료구조 과제 7

물리학과 20182326 이선민

1. 사전 탐색 트리 만들기

```
[~/2022_1/Data_structure/Assignment/assignment_7]$ ./bst        *[mast
사전 파일을 모두 읽었습니다. 48406개의 단어가 있습니다.
A 트리의 전체 높이는 37 입니다. A트리의 노드 수는 48406개 입니다.
단어를 입력하세요 hello
(레벨 : 24) int.여보
단어를 입력하세요 no
(레벨 : 13) a.무의
단어를 입력하세요 iwis
(레벨 : 30) ad.확실히
단어를 입력하세요 iterative
(레벨 : 27) adj.반복되는
단어를 입력하세요 0
search time: 36 ms
search time: 22 ms
search time: 31 ms
search time: 24 ms
```

- 사전 파일을 모두 읽어 이진 탐색 트리에 담은 후 총 노드의 개수와 트리의 높이를 세어 표시 하였다.
- 이진탐색 트리로 만든 경우 트리의 높이는 37, 총 노드 개수는 48406개가 나왔다.

이진 탐색 트리의 구현

```
42 void Insert(Node **tree, char word[ARR_SIZE])
43 {
44         if (*tree == NULL)
45         {
46                 *tree = Create_node(word);
47                 return;
48         }
49         Node *parent = NULL;
50         Node *tmp = *tree;
51         Node *newNode = Create_node(word);
52         while (tmp)
53         {
54                 parent = tmp;
55                 if (strcmp(newNode->eng,tmp->eng) < 0)
56                         tmp = tmp->left;
57                 else
58                         tmp = tmp->right;
59         }
60         if (strcmp(newNode->eng, parent->eng) < 0)
61                 parent->left = newNode;
62         else
63                 parent->right = newNode;
64 }
65
```

- 이진 탐색 트리는 문장 전체를 한문장을 파일에 읽어와 그 한 줄을 전부 함수로 넘겨 주는 형태로 만들었다.
- 단어가 하나씩 넘어오면 Create_node라는 함수에서 하나의 노드로 만들고 영어글자를 기준으로 대소 구분을 하여 노드를 트리에 하나씩 추가해주었다.

노드의 형태

```
 8 typedef struct _node{
 9         char eng[ARR_SIZE];
10         char kor[ARR_SIZE];
11         struct _node * right;
12         struct _node * left;
13 } Node;
```

- 노드의 형태는 이와 같이 트리를 위해 필요한 right, left 를 가리키는 포인터와 영어 단어를 담은 문자열 배열, 한글 뜻을 담은 문자열을 선언해 두었다.

트리 전체 사이즈

- 트리 전체 사이즈는
  전역변수로 선언했다.

```
14
15 int Tree_size = 0;
16
```

노드 생성

```
17 Node *Create_node(char word[ARR_SIZE]) {
18         Node *node = (Node *)malloc(sizeof(Node));
19         if (node == NULL)
20                 return 0;
21         int i = 0;
22         int j = 0;
23         for (i = 0; i < ARR_SIZE; i++)
24         {
25                 node->eng[i] = '\0';
26                 node->kor[i] = '\0';
27         }
28         i = 0;
29         while (word[i] != ':')
30                 node->eng[j++] = word[i++];
31         node->eng[--j] = '\0';
32         j = 0;
33         i++;
34         while (word[i] != '\n' && word[i])
35                 node->kor[j++] = word[++i];
36         node->kor[--j] = '\0';
37         node->right = NULL;
38         node->left = NULL;
39         return node;
40 }
```

- 노드는 이 함수에서 한 문장을 받아 : 을 기준으로 영어와 한글을 구분 하고 각각 저장한다.
- Right와 left는 NULL로 초기화 한 후 노드를 리턴해준다.

단어 검색 기능

```
77 Node *Search_word(Node *tree, char eng[ARR_SIZE], int *count)
78 {
79         (*count)++;
80         if (tree == NULL)
81                 return 0;
82         if (strcmp(eng, tree->eng) == 0)
83                 return tree;
84         if (strcmp(eng, tree->eng) < 0)
85                 return Search_word(tree->left, eng, count);
86         else
87                 return Search_word(tree->right, eng, count);
88 }
```

- 단어 검색해주는 함수는 트리와 검색할 단어, 그리고 count 정수 포인터를 매개변수로 받는다.
- 재귀 함수로 구성하였는데, count는 검색할 단어의 레벨을 찾기 위해 일부러 포인터 변수로 받았다.

트리 전체 높이, 트리 전체 노드 개수 찾기.

```
101 int get_height(Node *tree)
102 {
103         int height = 0;
104         if (tree)
105         {
106                 if (tree->left == NULL && tree->right == NULL)
107                         height = 1;
108                 else
109                 {
110                         int le = get_height(tree->left);
111                         int ri = get_height(tree->right);
112                         if (le >= ri)
113                                 height = le + 1;
114                         else
115                                 height = ri + 1;
116                 }
117         }
118         return height;
119 }
120
121 void get_tree_size(Node *tree)
122 {
123         if (tree)
124         {
125                 Tree_size++;
126                 get_tree_size(tree->left);
127                 get_tree_size(tree->right);
128         }
129 }
```

- 트리 전체 높이와 전체 노드 개수찾는 함수는 재귀 함수로 구성하였다.

검색 시간 측정

```
while (1)
{
        int level = 0;
        printf("단어를 입력하세요 ");
        scanf("%s",word);
        if (strcmp(word,"0") == 0)
                break;
        start = clock();
        Node *to_find = Search_word(tree, word, &level);
        end = clock();
        if (to_find == NULL)
                printf("찾는 단어가 없습니다.\n");
        else if (to_find != NULL)
                printf("(레벨 : %d) %s\n",level, to_find->kor)

        double runtime = (double)(end - start);
        search_time[j++] = runtime;
}
```

- 검색시간은 time.h 라이브러리를 import하여, clock함수를 활용해 검색시간을 측정 했다.

실행 결과

```
[~/2022_1/Data_structure/Assignment/assignment_7]$ ./bst        *[mast
사전 파일을 모두 읽었습니다. 48406개의 단어가 있습니다.
A 트리의 전체 높이는 37 입니다. A트리의 노드 수는 48406개 입니다.
단어를 입력하세요 hello
(레벨 : 24) int.여보
단어를 입력하세요 no
(레벨 : 13) a.무의
단어를 입력하세요 iwis
(레벨 : 30) ad.확실히
단어를 입력하세요 iterative
(레벨 : 27) adj.반복되는
단어를 입력하세요 0
search time: 36 ms
search time: 22 ms
search time: 31 ms
search time: 24 ms
```

- 결과를 보면 레벨이 높을 수록 검색시간이 증가하는 추세임을 알 수 있다.

2. 효과적인 사전 탐색 트리 만들기.

```
16 Node *g_arr[48406];
17 int Tree_size = 0;
18 int g_arr_index = 0;
19
```

- B트리의 경우 이전에 만들어 놓았던 이진 탐색 트리를 활용해 g_arr라는 배열에 정렬된 전체 노드를 저장 시킨 후 이 배열에서 중간 노드를 새 트리에 삽입 시키고, 그다음 재귀적인 형태로 중간 기준으로 앞 구간에서 중간 노드 새 트리에 삽입, 중간 기준으로 뒤 구간 중간 노드 새트리에 삽입, 이런식으로 만들었다.

Main 함수

```
Node *tree = NULL;
Node *sort_arr[48406];
char buffer[ARR_SIZE];
clock_t start, end;
double search_time[ARR_SIZE];
FILE *fp = fopen("randdict_utf8.TXT", "r");
int index = 0;
int i = 0;
while (i < ARR_SIZE)
        buffer[i++] = '\0';
while (!feof(fp))
{
        fgets(buffer, sizeof(buffer), fp);
        if (feof(fp)) break;
        fflush(fp);
        Node *newNode = Create_node(buffer);
        Insert(&tree, newNode);
}
fclose(fp);
```

- Main함수 를 보면 이 부분은 그전 코드와 같다.
- 파일을 읽어온 후 이진 탐색 트리를 생성한다.

트리 배열에 담기

```
TreeToArray(tree);
get_tree_size(tree);
```

```
82
83 void TreeToArray(Node *tree) {
84     if (tree)
85     {
86             TreeToArray(tree->left);
87             g_arr[g_arr_index++] = tree;
88             TreeToArray(tree->right);
89     }
90 }
```

- TreeToArray는 트리의 모든 노드를 정렬된 순서로 g_arr에 담는 것이다.
- Get_tree_size는 앞에서 사용했던 것과 같다.
- (트리 전체 노드 개수 세기)

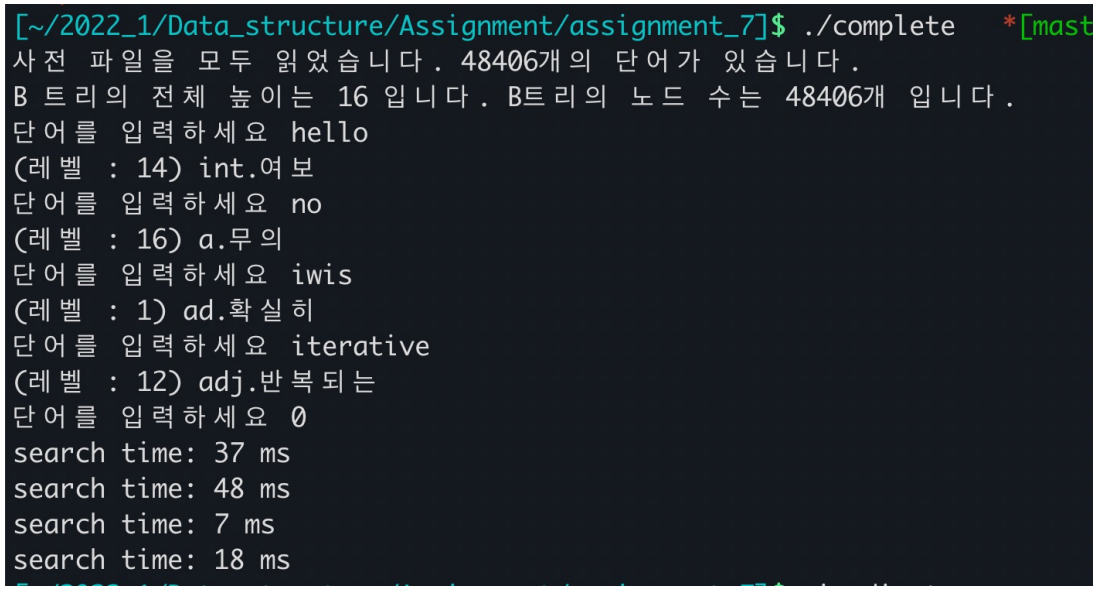
배열에 담긴 노드 다시 완전 이진 트리에 담기

```
    Node *complete = NULL;
    MakeCompleteTree(&complete, 0, Tree_size-1);
    g_arr[0]->left = NULL;
    g_arr[0]->right = NULL;
    Insert(&complete, g_arr[0]);
```

```
67
68 void MakeCompleteTree(Node **comp, int start, int end)
69 {
70         int index = (end + start) / 2 + ((end + start) % 2);
71         if (index < 0 || index > (Tree_size-1) || end <= start)
72                 return;
73         g_arr[index]->left = NULL;
74         g_arr[index]->right = NULL;
75         Insert(comp, g_arr[index]);
76         MakeCompleteTree(comp, start, index-1);
77         MakeCompleteTree(comp, index, end);
78 }
79
```

- 이제 main함수에서 complete이라는 새로운 트리의 루트를 만들어서 MakeCompleteTree함수를 호출한다.
- 이 함수는 재귀로 돌면서 처음과 끝 인덱스의 중간 노드를 새로운 트리 comp에 담는다.
- 이 알고리즘 대로 하면은 배열의 맨 처음 노드가 트리에 담기지 않아서 main함수에서 따로 담아주었다.

실행 결과.



- 노드 전체 개수는 이전과 같고 b트리 전체 높이는 완전 이진 탐색 트리이기 때문에 16이다.
- 실행 시간을 보면 마찬가지로 레벨이 높을 수록 탐색 시간이 길고, 단어를 여러차례 검색해보면 대부분 레벨이 14에서 16사이였다.

결과 비교

```
[~/2022_1/Data_structure/Assignment/assignment_7]$ ./complete    *[master]
사전 파일을 모두 읽었습니다. 48406개의 단어가 있습니다.
B 트리의 전체 높이는 16 입니다. B트리의 노드 수는 48406개 입니다.
단어를 입력하세요 hello
(레벨 : 14) int.여보
단어를 입력하세요 no
(레벨 : 16) a.무의
단어를 입력하세요 iwis
(레벨 : 1) ad.확실히
단어를 입력하세요 iterative
(레벨 : 12) adj.반복되는
단어를 입력하세요 0
search time: 37 ms
search time: 48 ms
search time: 7 ms
search time: 18 ms
```

```
[~/2022_1/Data_structure/Assignment/assignment_7]$ ./bst         *[master]
사전 파일을 모두 읽었습니다. 48406개의 단어가 있습니다.
A 트리의 전체 높이는 37 입니다. A트리의 노드 수는 48406개 입니다.
단어를 입력하세요 hello
(레벨 : 24) int.여보
단어를 입력하세요 no
(레벨 : 13) a.무의
단어를 입력하세요 iwis
(레벨 : 30) ad.확실히
단어를 입력하세요 iterative
(레벨 : 27) adj.반복되는
단어를 입력하세요 0
search time: 39 ms
search time: 28 ms
search time: 29 ms
search time: 37 ms
```

- 결과를 비교해보면 B트리의 경우가 전체 트리 레벨이 낮아 탐색시간이 더 단축된 것을 확인할 수 있다.
- 그런데 예상했던 것과 달리 탐색 시간이 눈에 띌 정도로 큰 차이는 없었다.