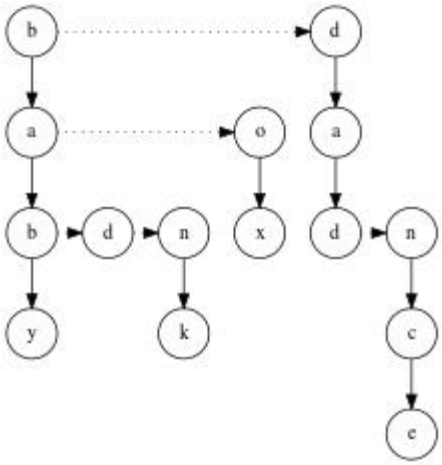
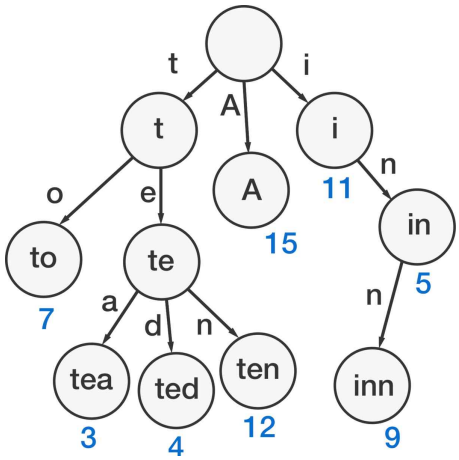
Programming Assignment #2

2017313264 홍진기

보충설명 : Trie



<그림 1>



<그림 2>

특정 단어에 대한 정보를 빠르게 탐색해, 올바른 파일로 접근하기 위해 **Trie**라는 트리 구조체를 이용한 메타데이터를 메모리 상에서 저장해 활용했습니다.
Trie는 일종의 트리 데이터 구조로써 노드 자체에 연관된 키를 저장하는 것이 아닌 트리상에서의 노드의 위치가 키의 값을 결정합니다. 이 경우 각 노드에 기본적으로 char형 변수를 저장하도록 해서 트리에 해당 단어가 있는지, 있다면 해당 단어에 대한 정보는 어떻게 되는지를 고속으로 탐색할 수 있습니다. 그림 1, 그림 2는 trie의 구조를 보여줍니다.

trie의 노드는 이웃한 노드들과 child-parent와 next의 관계를 가집니다. 우선 parent노드에 도달해 키 값을 검사해 만약 일치하면 child 노드로 내려보냅니다. 만약 key값이 일치하지 않는다면 next 노드를 탐색하도록 합니다.
이런 구조는 일종의 depth로도 생각할 수도 있습니다. child는 depth가 깊어지는 것이고, next는 같은 depth상에서 노드를 탐색하는 것입니다. 이 때, depth에서는 단어의 depth번째 문자를 비교합니다.
마지막으로, 어떤 단어가 입력으로 들어왔을 때, 그 단어에 대한 정보는 단어의 마지막 문자를 저장하고 있는 노드에 저장합니다.

그림 1로 예를 들어보겠습니다. "bank"라는 입력이 들어왔다고 가정합니다.
depth0 : "b" -> child node(depth1)
depth1 : "a" -> child node(depth2)
depth2 : "b" -> "d" -> "n" -> child node(depth3)
depth3 : "k"
다음과 같은 순서로 탐색을 해서 해당 단어가 있는지 빠르게 살펴볼 수 있습니다. 이 때, 그 단어에 대한 정보는 "k"에 저장되어있습니다.

이제 코드를 살펴보겠습니다.

```
24  typedef struct Node_{
25      struct Node_ *child;
26      struct Node_ *neighbor;
27      int latest;
28      int offset;
29      char data;
30  }Node;
```

Trie의 노드를 구현하기 위해 구조체를 선언했습니다. 구조체의 멤버는 각각 다음과 같은 정보를 저장합니다.

25 : struct Node_ *child : child노드의 주소를 저장합니다.

26 : struct Node_ *neighbor : next노드의 주소를 저장합니다.

27 : int latest : 해당 단어의 정보가 저장된 가장 최신 파일의 번호를 저장합니다.

28 : int offset : 해당 단어의 정보가 파일의 어느 위치에 있는지 offset을 저장합니다.

29 : char data : Trie구조를 위해 특정 문자를 저장합니다.

그리고 다음과 같은 전역변수들을 사용했습니다.

```
32  int Gsize;
33  int element_num;
34  int file_num;
35  Node *root = NULL;  //Root(first node) for trie tree
```

32 : int Gsize : 해쉬 테이블의 사이즈를 저장합니다.

33 : int element_num : 해쉬 테이블에 저장된 단어의 개수를 저장합니다.

34 : int file_num : 만들어진 파일의 개수를 저장합니다.

35 : Node *root : Trie의 시작 노드의 주소를 저장합니다. 처음에는 NULL로 초기화했습니다.

다음으로 Trie 구조체를 위한 함수들을 설명하겠습니다.

1. Node * find(char *word) : 문자열을 인자로 받아, 그 문자열이 Trie 내에 저장되어있는지 탐색합니다. 만약 존재한다면 그 정보를 담은 문자열의 마지막 문자에 해당하는 노드를 리턴합니다. 만약 존재하지 않는다면 0을 리턴합니다.

```
39 ▼ Node* find(char *word) {
40       int depth = 0;
41       Node *now = root;
42 ▼     while (1) {
43           if (now == NULL) return 0;
44           if (depth >= (int)strlen(word) - 1 && word[depth] == now->data) return now;
45           //for strlen == 1 word, I add condition 'word[depth] == now->data'
46           //if strlen == 1, first condition always becomes true
47
48           if (now->data != word[depth]) now = now->neighbor;
49 ▼         else {
50               now = now->child;
51               depth++;
52           }
53       }
54   }
```

43 : now가 NULL이 되었다는 것은 해당 단어가 저장되어 있지 않다는 것을 의미하므로 0을 리턴합니다.
44 : 적어도 depth가 단어 길이-1이 될 때까지 탐색을 계속합니다. -1이 들어간 이유는 depth는 0부터 시작하지만 단어 길이는 1부터 시작하기 때문입니다. 거기에 종료 조건으로 해당 노드에 저장된 문자가 단어의 문자와 일치해야 한다는 조건을 추가했습니다. 이는 만약 단어의 길이가 1일 경우 now가 root인 상태로 바로 함수가 종료되기 때문입니다.
48-52 : 함수가 실제로 동작하는 부분입니다. 노드 문자와 단어가 일치하면 child노드로 일치하지 않으면 next노드로 보냅니다.

2. void insert(char *word, int latest, int offset) : 문자열과 관련 정보를 받아 Trie에 저장합니다. 이 때, 만약 해당하는 문자열이 이미 존재하는 경우 latest와 offset 정보만 갱신해주고 그렇지 않다면 관련된 노드들을 추가로 생성하고 정보를 저장합니다.

```
56   void insert(char *word, int latest, int offset) {
57       if (root == NULL) {
58           root = (Node *)malloc(sizeof(Node));
59           *root = (Node) { NULL, NULL, -1, -1, word[0] };
60       } //If there is no data, initialize root
61
62       int depth = 0;
63       Node *now = root;
64       while (1) {
65           if (now->data != word[depth]) {
66               if (now->neighbor != NULL) now = now->neighbor;
67               else {
68                   Node *new = (Node *)malloc(sizeof(Node));
69                   *new = (Node) { NULL, NULL, -1, -1, word[depth] };
70
71                   now->neighbor = new;
72                   now = now->neighbor;
73               }
74           }
```

```
75          else {
76              if (depth >= (int)strlen(word) - 1) {
77                  now->latest = latest;
78                  now->offset = offset;
79                  return;
80              }//Termination condition
81
82              if (now->child != NULL) {
83                  now = now->child;
84                  depth++;
85              }
86              else {
87                  Node *new = (Node *)malloc(sizeof(Node));
88                  *new = (Node) { NULL, NULL, -1, -1, word[++depth] };
89
90                  now->child = new;
91                  now = now->child;
92              }
93          }
94      }
95  }
```

57-60 : 만약 Trie에 아무 정보도 저장되어 있지 않다면 root에 단어의 첫 문자를 갱신해줍니다. 이 때 child와 neighbor이 NULL인 것은 각각 child노드와 next노드가 존재하지 않는다는 것을 의미합니다. latest와 offset이 -1인 것은 아직 해당 단어가 입력된 적이 없다는 것을 의미합니다.

65-74 : 만약 노드의 데이터와 단어의 문자가 일치하지 않는다면 next노드를 탐색합니다. 이 때 next노드가 존재하지 않는다면 새로운 next노드를 생성한 후 해당 단어의 문자를 저장해줍니다.

76-80 : depth가 단어 길이-1이 되면 해당 노드에 정보를 갱신하고 함수를 종료합니다. 이 조건이 만족되었다는 것은 이미 존재하던 새로 만들었던 상관없이 입력된 단어의 마지막 문자에 해당하는 노드에 도착했다는 의미이기 때문입니다. 또한 노드의 데이터가 단어의 문자와 일치할 때만 종료 조건이 성립된다는 것을 압니다.

82-92 : 만약 노드의 데이터와 단어의 문자가 일치한다면 child노드로 내려보냅니다. child노드가 이미 존재한다면 이동하고 child노드가 없다면 새로 생성한 후 그 노드로 이동합니다. child노드로 내려가면서 depth는 1 증가해야합니다.

3. void Triefree(Node *now) :Trie를 만들 때 할당했던 메모리를 해제하는 함수입니다. 재귀를 사용해 Trie의 모든 노드를 순회하며 메모리를 해제합니다.

```
97  void Triefree(Node *now) {
98      if (now == NULL) return;
99      if (now->child == NULL && now->neighbor == NULL) {
100         free(now);
101         return;
102     }
103     Triefree(now->neighbor);
104     Triefree(now->child);
105     free(now);
106 }
```

98 : 재귀함수의 종료조건 now가 NULL이 되면 더 이상 탐색의 의미가 없으므로 함수를 종료합니다.
99-101 : 해당 조건을 만족한다는 것은 어떤 경로의 끝에 도달했다는 것이므로, 해당 노드를 해제하고 함수를 종료합니다.
103-104 : 재귀 용법. 각각 next와 child 노드로 뻗어나간다.
105 : 재귀가 끝나고 돌아오면서 메모리를 해제해야 하기 때문에 Triefree() 다음에 위치시켰다.

4. void db_put(db_t* db, char* key, int keylen, char* val, int vallen)

```
140 void db_put(db_t* db, char* key, int keylen, char* val, int vallen) {
141     int index = hash_func(key);
142     db_t *pos;
143 
144     //Check in Hash Table
145     for (pos = (db+index); pos->next != NULL; pos = pos->next)
146         if (!strcmp(pos->next->key, key)){
147             for (int i = 0; i < vallen; i++) pos->next->val[i] = val[i];
148             return;
149         }
150 
151     db_t *new = pos->next = (db_t*)malloc(sizeof(db_t));
152     new->next = NULL;
153     new->val = (char*)malloc(sizeof(char)*(vallen));
154     new->key = (char*)malloc(sizeof(char)*(keylen+1));
155     new->vallen = vallen;
156     new->keylen = keylen+1;
157     for (int i = 0; i < keylen; i++)  new->key[i] = key[i];
158     new->key[keylen] = '\0';//For strcmp
159     for (int i = 0; i < vallen; i++)  new->val[i] = val[i];
```

145-149 : 헤쉬 테이블 안에 키 값이 존재하는지 탐색합니다. 만약 찾았다면 함수를 종료합니다.
151-159 : 위에서 함수가 종료되지 않았다면 해쉬 테이블 안에 해당 키 값이 존재하지 않는다는 뜻입니다. 따라서 해당 키와 관련 정보를 저장할 노드를 만들고 이를 해쉬 테이블에 연결시켜줍니다. 이 때 key의 맨 뒤에 널 문자를 삽입해야만 이후 strcmp를 쓸 수 있습니다.

```
163     element_num++;
164     if (element_num >= Gsize){
165         file_num++;
166         char filename[50];
167         sprintf(filename, "./db/file-%d", file_num);
168         int fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0644);
169 
170         for (int i = 0 ; i < Gsize; i++){
171             db_t *pos;
172             db_t *tmp;
173             if ((db+i)->next ==NULL) continue;
174             for (pos = (db+i)->next; pos != NULL; pos = tmp){
175                 insert(pos->key, file_num, lseek(fd, 0, SEEK_CUR)); //
176                 int trashcan;
177                 trashcan = write(fd, &(pos->keylen), 4);
178                 trashcan = write(fd, pos->key, pos->keylen);
179                 trashcan = write(fd, pos->val, 4);
180                 trashcan++;
181                 tmp = pos->next;
182                 free(pos->val); free(pos->key); free(pos);
183             }
184             db[i].next = NULL;
185         }
186         close(fd);
187         element_num = 0;
188     }
189 }
```

163-164 : element_num는 해쉬 테이블에 저장되어 있는 단어의 수를 저장합니다. 만약 해쉬 테이블에 저장된 단어의 수가 해쉬 테이블의 사이즈와 같아지면 그 정보를 파일에 저장하고, 해쉬 테이블을 초기화하려고 합니다.

165-168 : file_num을 1 증가시키고 sprintf로 “file-%d" 형식의 문자열을 만들고 이것을 이름으로 하는 파일을 생성합니다. O_CREAT 과 O_TRUNC 옵션을 줘서 파일이 없다면 새로 만들고, 있다면 파일을 초기화한 후 데이터를 저장하도록 했습니다.

170-182 : 파일을 생성한 후, 해쉬 테이블에 저장된 정보를 파일로 옮깁니다. 옮기기 전 insert함수로 각각 key단어, 저장할 파일 정보, 그리고 offset을 저장합니다. 이 때 lseek의 리턴값을 통해 파일에 저장된 단어의 offset을 알 수 있습니다. 그 다음 파일에 정보를 저장합니다. 각각 키의 길이 4바이트, 키 단어 keylen바이트, 그리고 val값 4바이트를 할당해 저장합니다. 그리고 해쉬 테이블에 할당된 메모리들을 해제합니다.

184 : 그리고 엔트리의 next에 널 값을 저장해 해쉬 테이블을 초기화해줍니다.

5. char* db_get(db_t* db, char* key, int keylen, int* vallen)

```
193 char* db_get(db_t* db, char* key, int keylen, int* vallen) {
194     int index = hash_func(key);
195     db_t *pos;
196     for (pos = (db+index)->next; pos != NULL; pos = pos->next)
197         if (!strcmp(pos->key, key)) {
198             *vallen = pos->vallen;
199             char *val = (char*)malloc(sizeof(char)*(pos->vallen));
200             for (int i = 0; i < pos->vallen; i++) val[i] = pos->val[i];
201             return val;
202         }//Search in Hash Table
203
204     Node *found;
205     if ((found = find(key)) != 0){
206         if (found->latest == -1) return NULL;
207         char filename[50];
208         sprintf(filename, "./db/file-%d", found->latest);
209
210         int fd = open(filename, O_RDONLY);
211         lseek(fd, found->offset, SEEK_SET);
212         int kl;
213         char val[4];
214         char temp[1024];
215         int trashcan;
216         trashcan = read(fd, &kl, 4);
217         trashcan = read(fd, temp, kl);
218         trashcan = read(fd, val, 4);
219         trashcan++;
220
221         char *val2 = (char*)malloc(sizeof(char)*4);
222         for (int i = 0; i < 4; i++) val2[i] = val[i];
223         close(fd);
224         return val2;
225     }
226     return NULL;
227
228 }
```

196-202 : 해쉬 테이블이 우리가 찾는 키 값이 저장되어 있는지 확인해서 만약 있다면 그 키의 val값을 리턴합니다. 만약 이 구간을 통과한다면 해쉬 테이블에 찾는 정보가 없다는 뜻입니다.

205, 226 : Trie에 우리가 찾는 키 값의 정보가 있는지 확인합니다. 만약 없다면 이는 파일에도 저장되어 있지 않은 처음 들어온 단어라는 의미이므로 NULL값을 리턴합니다.

206 : 만약 latest값이 -1이라면 이는 단어의 일부분으로써 저장되어 있다는 뜻으로 진짜 그 단어가 저장되어 있지는 않다는 의미입니다. 예를 들어 “there”이라는 단어의 정보가 저장되어 있다면 “the"의 흔적은 남아있지만 그렇다고 "the”가 저장되어 있지는 않습니다. 이 때 "the“의 latest는 -1이 됩니다. 따라서 NULL을 리턴합니다.

207-219 : 이곳까지 도달했다는 것은 헤쉬테이블이 아닌 파일에 단어에 대한 정보가 존재한다는 뜻입니다. 따라서 Trie함수를 참조해 그 데이터가 파일의 어느 위치에 있는지를 파악한 후 read를 통해 정보를 읽어옵니다.

221-224 : 그 다음 그 단어의 val값을 리턴해줍니다.