ASSIGNMENT#2

자료구조 02분반

20220357

구민정

1. Assignment #2-1: BST
   1. Node Insert(Node\* node, int data) 함수

나눠진 경우의 수는 각각 주석으로 설명되어 있다.

Node\* Insert(Node\* node, int data)

{

// 노드가 비어있을 경우에 새 노드를 만들어서 데이터를 넣고 반환

if (node == NULL)

{

Node\* newNode = (Node\*)malloc(sizeof(Node));

newNode->data = data;

newNode->Left = NULL;

newNode->Right = NULL;

return newNode;

}

else

{

// 이미 노드가 있는 경우에는 새로운 데이터를 노드에 맞게 삽입

/\*

TODO: Fill the conditions in Insert function

\*/

// 삽입할 데이터가 현재 노드의 데이터보다 작으면 왼쪽 자식 노드로 이동

if (data < node->data)

{

node->Left = Insert(node->Left, data);

return node;

}

/\*

TODO: Fill the conditions in Insert function

\*/

// 삽입할 데이터가 현재 노드의 데이터보다 크면 오른쪽 자식 노드로 이동

else if (data > node->data)

{

node->Right = Insert(node->Right, data);

return node;

}

// 삽입하려는 데이터가 이미 트리에 있는 경우에는 그대로 반환

else

{

return node;

}

}

}

위 함수의 흐름은 다음과 같다.

1) 새로운 데이터를 삽입하기 위해 함수를 호출한다.

2) 현재 노드가 NULL인 경우, 새로운 노드를 생성하고 해당 데이터를 삽입한다.

3) 새로운 데이터가 현재 노드의 데이터보다 작으면 왼쪽 하위 트리로 이동하여 재귀적으로 Insert 함수를 호출한다.

4) 새로운 데이터가 현재 노드의 데이터보다 크면 오른쪽 하위 트리로 이동하여 재귀적으로 Insert 함수를 호출한다.

5) 데이터가 삽입된 후에는 해당 노드를 반환한다.

* 1. int Search(Node\* node, int data) 함수

나눠진 경우의 수는 각각 주석으로 설명되어 있다.

int Search(Node\* node, int data)

{

// 노드가 비어있으면 해당 데이터가 트리에 없다는 메시지를 출력하고 NULL을 반환

if (node == NULL) {

printf("No Node %d ", data);

return NULL;

}

// 데이터를 찾으면 해당 데이터를 반환

if (node->data == data)

{

return node->data;

}

// 찾으려는 데이터와 현재 노드의 데이터를 비교하여 적절한 방향으로 이동

else

{

/\*

TODO: Implement the Search function conditions

\*/

if (data < node->data)

{

return Search(node->Left, data);

}

else if (data > node->data)

{

return Search(node->Right, data);

}

}

}

위 함수의 흐름은 다음과 같다.

1) 특정 데이터를 검색하기 위해 함수를 호출한다.

2) 현재 노드가 NULL인 경우, 해당 데이터가 트리에 없음을 나타내는 메시지를 출력하고 NULL을 반환한다.

3) 현재 노드의 데이터와 검색할 데이터를 비교한다.

4) 검색할 데이터가 현재 노드의 데이터와 일치하면 해당 데이터를 반환한다.

5) 검색할 데이터가 현재 노드의 데이터보다 작으면 왼쪽 하위 트리로 이동하여 재귀적으로 Search 함수를 호출한다.

6) 검색할 데이터가 현재 노드의 데이터보다 크면 오른쪽 하위 트리로 이동하여 재귀적으로 Search 함수를 호출한다.

* 1. Node\* Delete(Node\* node, int data) 함수

나눠진 경우의 수는 각각 주석으로 설명되어 있다.

Node\* Delete(Node\* node, int data)

{

if (node == NULL) return NULL;

if (node->data == data)

{

Node\* deleteNode = node;

/\*

TODO: Implement the Delete function conditions\*/

if (node->Left == NULL && node->Right == NULL) {

free(deleteNode);

return NULL;

}

else if (node->Left == NULL) {

Node\* temp = node->Right;

free(deleteNode);

return temp;

}

else if (node->Right == NULL) {

Node\* temp = node->Left;

free(deleteNode);

return temp;

}

else {

Node\* temp = node->Right;

while (temp->Left != NULL) {

temp = temp->Left;

}

node->data = temp->data;

node->Right = Delete(node->Right, temp->data);

}

return node;

}

else {

if (data < node->data) {

node->Left = Delete(node->Left, data);

}

else if (data > node->data) {

node->Right = Delete(node->Right, data);

}

}

return node;

}

위 함수의 흐름은 다음과 같다.

1) 특정 데이터를 삭제하기 위해 함수를 호출한다.

2) 현재 노드가 NULL이면 아무 작업도 수행하지 않고 NULL을 반환한다.

3) 현재 노드의 데이터와 삭제할 데이터를 비교한다.

4) 삭제할 데이터가 현재 노드의 데이터와 일치하면 해당 노드를 삭제한다.

5) 삭제할 노드의 자식 노드가 없는 경우, 그냥 삭제하고 NULL을 반환한다.

6) 하나의 자식을 가진 경우, 해당 자식을 현재 노드의 위치로 이동시킨다.

7) 두 개의 자식을 가진 경우, 오른쪽 하위 트리에서 가장 작은 값을 가진 노드를 찾아 해당 값을 현재 노드로 복사한 후, 해당 노드를 삭제한다.

8) 재귀적으로 자식 노드들을 업데이트하고 변경된 루트 노드를 반환한다.

D. 실행결과

위의 함수들로 해당 파일을 채워 넣으면 아래와 같은 결과가 나온다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

1. Assignment #2-1: TRIE
   1. void Insert(TrieNode\* root, const char\* key)

추가된 코드에 대한 설명은 주석으로 설명되어 있다.

// Function to insert a word into the trie

void Insert(TrieNode\* root, const char\* key) {

TrieNode\* crawler = root;

while (\*key) {

/\*

TODO: write a traversal on the key for inserting nodes in tree

\*/

int index = \*key - 'a'; // 현재 문자의 인덱스를 계산 ('a'의 ASCII 값 빼기)

// 자식 노드가 없다면 새로운 노드 생성

if (!crawler->children[index]) {

crawler->children[index] = createNode();

}

// 다음 노드로 이동

crawler = crawler->children[index];

key++;

}

// 단어의 끝을 표시

crawler->isEndOfWord = true;

}

위 함수의 흐름은 다음과 같다.

1. 새로운 키를 삽입하기 위해 함수를 호출한다.
2. 루트 노드에서 시작하여 각 문자를 따라 트라이를 탐색한다.
3. 현재 노드의 자식 노드 중 해당 문자가 없다면 새로운 노드를 생성하여 연결한다.
4. 다음 문자로 이동하며 계속해서 자식 노드를 탐색하거나 생성한다.
5. 키의 마지막 문자에 도달하면, 해당 노드를 단어의 끝으로 표시한다.
   1. bool search(TrieNode\* root, const char\* key)

추가된 코드에 대한 설명은 주석으로 설명되어 있다.

bool search(TrieNode\* root, const char\* key) {

TrieNode\* crawler = root;

while (\*key) {

/\*

TODO: write a traversal on the key for searching nodes in tree

\*/

// 현재 문자의 인덱스를 계산

int index = \*key - 'a';

// 자식 노드가 없다면 단어가 존재하지 않음

if (!crawler->children[index]) {

return false; // Word not found

}

// 다음 노드로 이동

crawler = crawler->children[index];

key++;

}

// 단어의 끝이 존재하는지 확인

return (crawler != NULL && crawler->isEndOfWord);

}

위 함수의 흐름은 다음과 같다.

1. 특정 키를 검색하기 위해 함수를 호출한다.
2. 루트 노드에서 시작하여 각 문자를 따라 트라이를 탐색한다.
3. 현재 노드의 자식 노드 중 해당 문자가 없다면, 트라이에 키가 없음을 나타내고 false를 반환한다.
4. 다음 문자로 이동하며 계속해서 자식 노드를 탐색한다.
5. 키의 마지막 문자에 도달하면, 현재 노드가 null이 아니고 단어의 끝으로 표시되어 있는지 확인한다. 조건을 만족하면 true를 반환한다.
   1. bool deleteHelper(TrieNode\* root, const char\* key, int depth)

추가된 코드에 대한 설명은 주석으로 설명되어 있다.

bool deleteHelper(TrieNode\* root, const char\* key, int depth) {

if (!root) {

return false;

}

// If last character of key is being processed

if (\*key == '\0') {

if (root->isEndOfWord) {

root->isEndOfWord = false;

// If root does not have any children (its not prefix of any other word)

if (isEmpty(root)) {

return true; // Delete the node

}

return false; // Do not delete the node

}

}

else {

int index = \*key - 'a';

if (deleteHelper(root->children[index], key + 1, depth + 1)) {

/\*

TODO: write a traversal on the key for searching nodes and deleting

Hint: recersive solutions are always in handy

\*/

root->children[index] = NULL;// 자식 노드 삭제

// 현재 노드가 단어의 끝이 아니고 자식 노드가 없는 경우 삭제

if (isEmpty(root) && root->isEndOfWord == false) {

free(root);

return true;

}

}

}

return false;

}

위 함수의 흐름은 다음과 같다.

1. 특정 키를 삭제하기 위해 재귀적으로 호출된다.
2. 루트가 null이면 false를 반환한다.
3. 현재 깊이가 키의 길이와 같으면, 해당 노드가 단어의 끝인지 확인한다. 단어의 끝이면 이를 false로 설정하고 자식 노드가 없는지 확인한다.
4. 자식 노드가 없으면 현재 노드를 삭제할 수 있으므로 true를 반환한다.
5. 현재 깊이가 키의 길이보다 작으면, 현재 문자에 해당하는 자식 노드로 이동하여 재귀적으로 삭제를 시도한다.
6. 자식 노드가 삭제되면, 현재 노드에서 해당 자식 노드 포인터를 null로 설정하고 현재 노드가 단어의 끝이 아니며 모든 자식 노드가 없는 경우에 true를 반환한다.
   1. void suggestions(TrieNode\* root, const char\* currentPrefix

추가된 코드에 대한 설명은 주석으로 설명되어 있다.

void suggestions(TrieNode\* root, const char\* currentPrefix) {

if (root->isEndOfWord) {

printf("%s\n", currentPrefix);

}

if (isEmpty(root)) {

return;

}

// 모든 알파벳을 순회하며 자식 노드가 있는 경우 새로운 접두사로 재귀 호출

for (int i = 0; i < ALPHABET\_SIZE; i++) {

if (root->children[i]) {

/\*

TODO: write a traversal on the prefix for searching nodes and completing

Hint: recersive solutions are always in handy

\*/

char newPrefix[100];

strcpy(newPrefix, currentPrefix);

newPrefix[strlen(currentPrefix)] = 'a' + i;

newPrefix[strlen(currentPrefix) + 1] = '\0';

suggestions(root->children[i], newPrefix);

}

}

}

위 함수의 흐름은 다음과 같다.

1. 현재 노드에서 시작하는 모든 단어를 출력하기 위해 함수를 호출한다.
2. 현재 노드가 단어의 끝이면, 현재 접두사를 출력한다.
3. 'a'부터 'z'까지 모든 문자를 확인하여, 자식 노드가 있는 경우 해당 문자로 접두사를 확장하고 재귀적으로 함수를 호출한다.
4. 재귀적으로 호출된 함수는 확장된 접두사를 사용하여 모든 가능한 단어를 탐색하고 출력한다.
   1. 실행결과

위의 함수들로 해당 파일을 채워 넣으면 아래와 같은 결과가 나온다.

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

자동 생성된 설명