// 비트리 탐색, 삽입, 삭제 프로그램

// 입력: 2 개의 회사명 파일 - Com\_names1.txt, Com\_names2.txt

// (주의: 회사명은 중간에 space 글자를 포함할 수 도 있음.)

// 먼저 두 파일 내의 모든 회사명을 각 레코드로 하여 넣고,

// 그 다음 명령문 실행 루프로 실험함.

#define \_CRT\_SECURE\_NO\_WARNINGS

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#define MAXK 2 // MAXK는 2d 임

#define HALFK MAXK/2 // d (capacity order) 이다.

#define MAX 100 // 스택// 비트리 탐색, 삽입, 삭제 프로그램

// 입력: 2 개의 회사명 파일 - Com\_names1.txt, Com\_names2.txt

// (주의: 회사명은 중간에 space 글자를 포함할 수 도 있음.)

// 먼저 두 파일 내의 모든 회사명을 각 레코드로 하여 넣고,

// 그 다음 명령문 실행 루프로 실험함.

#define \_CRT\_SECURE\_NO\_WARNINGS

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#define MAXK 2 // MAXK는 2d 임

#define HALFK MAXK/2 // d (capacity order) 이다.

#define MAX 100 // 스택 최대 원소수.

#define false 0

#define true 1

typedef struct element { // 레코드 정의. 회사명과 그 길이로 구성됨

char name[100]; // 회사명

int nleng; // 회사명 길이

}ele;

typedef struct node\* nodeptr; // \*nodeptr은 node의 형태를 갖는 포인터이다.

typedef struct node { // 일반 노드

int fill\_cnt;

ele rec[MAXK];

nodeptr ptr[MAXK + 1];

}node;

typedef struct big\_node { // 일반노드보다 레코드와 포인터가 하나씩 더 큰 빅노드 [합병시 사용]

nodeptr ptr[MAXK + 2];

ele rec[MAXK + 1];

}big\_node;

typedef struct two\_Bn {

nodeptr ptr[2 \* MAXK + 1];

ele rec[2 \* MAXK];

} two\_Bnode;

nodeptr root = NULL; // root는 전역 변수로 사용한다.

//nodeptr temp;

nodeptr stack[MAX]; //stack의 max값은 100

int top = -1;

void push(nodeptr node);

nodeptr pop();

void insert\_btree(); // file 전체를 삽입 이 함수는 insert\_arec을 호출한다.

int insert\_arec(ele in\_rec); // 레코드를 하나 삽입한다.

nodeptr retrieve(char\*, int\*); // 키값을 넣어 검색한다. [깊이와 함께 출력]

int seq\_scan\_btree(nodeptr curr); // 전체 레코드들을 출력한다.

int B\_tree\_deletion(char\*); // Function to delete a record with a given key.

int total\_height = 0; // 전체 높이

int main(void) {

char input, line[200], \* res\_gets;

ele a\_insert;

char name\_s[100];

int num, lleng;

int i, k, r, find;

nodeptr tp;

insert\_btree();

// 명령문 수행 루프.

while (1) {

fflush(stdin); // buffer clear

printf("\n명령을 넣으시오\n");

printf("insert: i 이름 / delete : d 이름 / retrieve : r 이름 / 전체출력: p / 종료: e >> ");

res\_gets = gets(line);

if (!res\_gets)

break; // no input.

lleng = strlen(line);

if (lleng <= 0)

continue; // empty line input. try again.

i = 1;

if (line[0] == 'E' || line[0] == 'e') { // exit program

printf("종료명령이 들어왔음!\n\n"); return 0;

}

else if (line[0] == 'I' || line[0] == 'i') { // 레코드 한개 삽입 명령.

// 먼저 회사명을 가져옴.

k = 0;

while (line[i] == ' ') i++;

while (i < lleng) {

name\_s[k] = line[i]; k++; i++;

}

name\_s[k] = '\0';

if (strlen(name\_s) == 0)

continue;

strcpy(a\_insert.name, name\_s);

a\_insert.nleng = strlen(name\_s);

top = -1;

r = insert\_arec(a\_insert); // 레코드 하나만 삽입하는 함수를 호출

if (r == 0)

printf("삽입 실패.\n", r);

else

printf("삽입 성공.\n", r);

}

else if (line[0] == 'D' || line[0] == 'd') { // 삭제 명령 수행

k = 0;

while (line[i] == ' ') i++;

while (i < lleng) {

name\_s[k] = line[i]; k++; i++;

}

name\_s[k] = '\0';

if (strlen(name\_s) == 0)

continue;

r = B\_tree\_deletion(name\_s);

if (r == 0)

printf("삭제요청 받은 키의 레코드가 존재하지 않음.\n", r);

else

printf("삭제 성공.\n");

}

else if (line[0] == 'R' || line[0] == 'r') { // 탐색 명령 수행

k = 0;

while (line[i] == ' ') i++;

while (i < lleng) {

name\_s[k] = line[i]; k++; i++;

}

name\_s[k] = '\0';

if (strlen(name\_s) == 0)

continue;

tp = retrieve(name\_s, &i);

if (tp)

printf("탐색성공. Name: %s, 길이: %d\n", tp->rec[i].name, tp->rec[i].nleng);

else

printf("탐색요청 받은 키를 가진 레코드가 존재하지 않음.\n");

}

else if (line[0] == 'P' || line[0] == 'p') // 프린트 명령 수행. 모든 레코드를 키 순서로 출력.

seq\_scan\_btree(root);

else

printf("input error \n");

} //while

} //main

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* stack 관련 함수들 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void push(nodeptr node) {

if (top > MAX) /// >>> MAX-1 이상( >= ) 이어야 하지 않나?

{

printf("stack is full\n");

return;

}

top++;

stack[top] = node;

}

nodeptr pop() {

int temp = 0;

if (top < 0)

{

printf("stack is empty.\n");

return 0;

}

temp = top; ///>>> 현재 top 이 가리키는 곳에 가장 최근의 데이타가 저장되어 있다. 따라서,

top--;

return stack[temp];

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// 레코드 하나를 삽입하는 함수이다.

// 반환값: 0: 삽입실패, 1, 2: 삽입성공 (1: 층증가 없이, 2:한 층 더 늘어 남.)

int insert\_arec(ele in\_rec) { //하나의 레코드를 삽입 key = 회사명

int i, j;

nodeptr curr, child, new\_ptr, tptr = NULL;

ele empty = { "\0",0 };

big\_node bnode;

if (!root) { // root가 NULL이면 btree가 비어있음. 맨 첫 노드를 만들어 여기에 넣는다.

root = /\* Fill your code \*/; // nodeptr형태로 node크기만큼 할당받아 시작주소는 root가 가짐

root->rec[0] = in\_rec; // key값을 root->rec[0]에 넣는다.

root->/\* Fill your code \*/ = root->/\* Fill your code \*/ = NULL; // p0과 p1에 NULL을 넣는다.

root->fill\_cnt = 1;

return 1; // 첫 노드를 만들어 넣고 종료함.

}

//root is not null

curr = root;

// 아래 빈 곳은 in\_rec 이 들어가면 좋을 리프노드를 찾아 curr가 가리키게 하는 부분이 와야 함!!

/\*

Fill

your

code

\*/

do {

// curr node is not full

if (curr->fill\_cnt < MAXK) {

for (i = 0; i < curr->fill\_cnt; i++)

if (strcmp(in\_rec.name, /\* Fill your code \*/ < 0)

break;

for (j = curr->fill\_cnt; j > i; j--) {

curr->ptr[j + 1] = /\* Fill your code \*/;

curr->rec[j] = /\* Fill your code \*/;

}

curr->rec[i] = /\* Fill your code \*/;

curr->ptr[i + 1] = /\* Fill your code \*/;

curr->fill\_cnt++;

return 1; // 삽입성공 (종류 1: 루트의 추가 없이 가능함).

}

else {

// curr node is full

for (i = 0; i < MAXK; i++) {

if (strcmp(in\_rec.name, /\* Fill your code \*/) < 0)

break;

}

bnode.ptr[0] = /\* Fill your code \*/;

for (j = 0; j < i; j++) {

bnode.rec[j] = /\* Fill your code \*/;

bnode.ptr[j + 1] = /\* Fill your code \*/;

}

bnode.rec[j] = /\* Fill your code \*/;

bnode.ptr[j + 1] = /\* Fill your code \*/;

j++;

while (i < MAXK) {

bnode.rec[j] = /\* Fill your code \*/;

bnode.ptr[j + 1] = /\* Fill your code \*/;

j++;

i++;

}

// 아래 빈 곳은 big node 를 3 부분으로 나누어 전반부는 curr 에, 가운데 레코드는 in\_rec에,

// 후반부는 새 노드에 넣고, child가 이 새 노드를 가리키게 하는 부분이 와야 함!!

/\*

Fill

your

code

\*/

if (top >= 0) { // 스택이 emtpy 가 아닐 경우

curr = pop(); // curr 의 부모로 curr를 변경함.

}

else { // 스택이 empty 임 (즉 curr 는 root 노드임.) 새 root 노드를 만들어 curr 의 부모로 함.

tptr = /\* Fill your code \*/;

tptr->rec[0] = /\* Fill your code \*/;

tptr->ptr[0] = /\* Fill your code \*/;

tptr->ptr[1] = /\* Fill your code \*/;

tptr->fill\_cnt = /\* Fill your code \*/;

root = /\* Fill your code \*/;

total\_height++;

return 2; // 삽입 성공 (종류 2: 새 루트가 생김)

} // else.

} // else.

} while (1);

return 0; // 이 문장을 수행할 경우는 없다.

} //함수 insert\_arec

void insert\_btree() { //파일전체의 레코드를 삽입 ->insert\_arec 을 호출

FILE\* fp;

ele data;

char name\_i[20], line[200];

char\* ret\_fgets;

int num, r;

double score;

int n = 0, lineleng;

int check, count = 0;

fp = fopen("Com\_names1.txt", "r");

if (fp == NULL) {

printf("Cannot open this file : Com\_names1.txt\n");

scanf("%d", &check);

} //if

root = NULL;

while (1) {

ret\_fgets = fgets(line, 200, fp);

if (!ret\_fgets)

break; // 파일 모두 읽음.

lineleng = strlen(line); // line 의 마지막 글자는 newline 글자임.

if (lineleng - 1 >= 100)

continue; // 회사명이 너무 길어서 무시

line[lineleng - 1] = '\0'; // 마지막 newline 글자 제거.

strcpy(data.name, line); // 삽입할 레코드 준비.

data.nleng = strlen(line);

top = -1; // 스택의 빈상태로 초기화.

r = insert\_arec(data); // 한 레코드를 비트리에 삽입한다.

if (r != 0)

count++; // 삽입성공 카운트 증가.

} //while

fp = fopen("Com\_names2.txt", "r");

if (fp == NULL) {

printf("Cannot open this file : Com\_names2.txt\n");

scanf("%d", &check);

} //if

while (1) {

ret\_fgets = fgets(line, 200, fp);

if (!ret\_fgets)

break; // 파일 모두 읽음.

lineleng = strlen(line); // line 의 마지막 글자는 newline 글자임.

if (lineleng - 1 >= 100)

continue; // 회사명이 너무 길어서 무시

line[lineleng - 1] = '\0'; // 마지막 newline 글자 제거.

strcpy(data.name, line); // 삽입할 레코드 준비.

data.nleng = strlen(line);

top = -1; // 스택의 빈상태로 초기화.

r = insert\_arec(data); // 한 레코드를 비트리에 삽입한다.

if (r != 0)

count++; // 삽입성공 카운트 증가.

} //while

printf("삽입 성공한 레코드 수 = %d \n\n", count);

fclose(fp);

} // 함수 insert\_btree

nodeptr retrieve(char\* skey, int\* idx\_found) { //검색 함수

nodeptr curr = root;

nodeptr P;

int i;

do {

for (i = 0; i < curr->fill\_cnt; i++) {

if (strcmp(skey, curr->rec[i].name) < 0)

break;

else if (strcmp(skey, curr->rec[i].name) == 0) {

\*idx\_found = i;

return curr;

}

else

; // do next i.

} // for i=

P = curr->ptr[i];

if (P) {

curr = P;

}

} while (P);

return NULL;

}//retrieve

int seq\_scan\_btree(nodeptr curr) {

int check\_stack = 0;

int i;

int n;

if (curr)

{

n = curr->fill\_cnt;

for (i = 0; i < n; i++) {

seq\_scan\_btree(curr->ptr[i]);

printf("Name : %s\n", curr->rec[i].name);

}

seq\_scan\_btree(curr->ptr[i]);

} //if(curr)

else if (!curr)

{

return 0;

}

return 0;

} //seq\_scan\_btree

// 좌측형제노드 내용, 부모의 레코드, 우측형제 내용을 받아서 재분배를 하는 함수

// wcase: curr 가 좌측형제와 재분배이면 ‘L’, curr 가 우측형제와 재분배이면 ‘R’.

// j : father 안에서 curr를 가리키는 포인터의 인덱스임.

void redistribution(nodeptr father, nodeptr l\_sibling, nodeptr r\_sibling, char wcase, int j) {

int i, k, m, n, h;

two\_Bnode twoB; // twobnode(bnode의 2배의 공간)

if (wcase == 'L')

j = j - 1;

else if (wcase == 'R')

j = j;

//copy l\_sibling's content, intermediate key in father, r\_sibling's content to twobnode;

for (i = 0; i < /\* Fill your code \*/; i++) {

twoB.ptr[i] = /\* Fill your code \*/;

twoB.rec[i] = /\* Fill your code \*/;

}

twoB.ptr[i] = /\* Fill your code \*/;

// 주의: j 에 father 에서의 l\_sibling 에 대한 index 가 들어 있음.

twoB.rec[i] = /\* Fill your code \*/; // 부모에서의 중간 키를 가져옴.

i++;

for (k = 0; k < /\* Fill your code \*/; k++, i++) {

twoB.ptr[i] = /\* Fill your code \*/;

twoB.rec[i] =/\* Fill your code \*/;

}

twoB.ptr[i] = /\* Fill your code \*/;

//Split twobnode into first half, middle record, second half;

h = i / 2; // h is the index of middle record.

//copy first half to left node;

for (n = 0; n < h; n++) {

l\_sibling->ptr[n] = /\* Fill your code \*/;

l\_sibling->rec[n] = /\* Fill your code \*/;

}

l\_sibling->ptr[n] = /\* Fill your code \*/;

l\_sibling->fill\_cnt = /\* Fill your code \*/;

//copy second half to r\_sibling;

n++;

for (m = 0; m < (i - h - 1); m++, n++) {

r\_sibling->ptr[m] = /\* Fill your code \*/;

r\_sibling->rec[m] = /\* Fill your code \*/;

}

r\_sibling->ptr[m] = /\* Fill your code \*/;

r\_sibling->fill\_cnt = /\* Fill your code \*/;

//move the middle record to father ;

father->rec[j] = /\* Fill your code \*/;

} // end of redistribution

int B\_tree\_deletion(char\* out\_key) {

nodeptr curr, r\_sibling, l\_sibling, father, Pt, leftptr, rightptr;

int i, j, k, r\_OK, l\_OK, found = 0, finished = 0;

curr = root;

// Step (0): search for a record (to be deleted) whose key equals out\_key.

do {

for (i = 0; i < /\* Fill your code \*/; i++)

if (strcmp(out\_key, /\* Fill your code \*/->rec[i].name) < 0)

break;

else if (strcmp(out\_key, /\* Fill your code \*/->rec[i].name) == 0) {

found = 1; break;

}

if (found == 1)

break; // 주의: 변수 i에 찾은 위치가 들어 있음.

else { // curr에 없다. child로 내려 가야 한다.

Pt = /\* Fill your code \*/;

if (Pt) {

push(/\* Fill your code \*/);

curr = Pt;

}

else

break;

}

} while (!found);

if (!found) {

return 0;

}

// Comes here when the key is found. It is in curr's node. i has index of rec to delete.

// Step (1): find successor of d\_rec.

if (curr->ptr[0]) { // curr node is not a leaf node

// We need to find successor of out\_key ;

Pt = /\* Fill your code \*/;

push(/\* Fill your code \*/);

// 가장 왼쪽 포인터를 따라내려 간다.

while (/\* Fill your code \*/) {

push(/\* Fill your code \*/);

Pt = /\* Fill your code \*/;

}

curr->rec[i] = Pt->rec[0];

curr = Pt;

i = 0;

} //end if

// curr 노드에서 index 가 i 인 레코드와 그 우측 포인터를 삭제하여야 한다.

finished = false;

do {

// Step (2):

//Remove record of index i and a pointer to its right from curr's node;

for (j = i + 1; j < /\* Fill your code \*/; j++) {

curr->rec[j - 1] = /\* Fill your code \*/;

/\* Fill your code \*/ = curr->ptr[j + 1];

}

curr->fill\_cnt = curr->fill\_cnt - 1;

// Step (3):

if (curr == root) {

if (curr->fill\_cnt == /\* Fill your code \*/) {

root = /\* Fill your code \*/;

free(curr);

}

return 1; // deletion succeeded.

}

// Step (4):

// curr is not the root.

if (curr->fill\_cnt >= HALFK) { return 2; } // Finish deletion with success.

// Now, curr violates minimum capacity constraint.

// Step (5):

father = pop(); // bring father of curr.

// r-sibling = pointer to right sibling of curr' node;

// l-sibling = pointer to left sibling of curr's node;

for (j = 0; j <= father->fill\_cnt; j++)

if (father->ptr[j] == curr) // find ptr of father which goes down to curr.

break;

if (j >= 1)

l\_sibling = father->ptr[j - 1];

else

l\_sibling = NULL;

if (j < father->fill\_cnt)

r\_sibling = father->ptr[j + 1];

else

r\_sibling = NULL;

// 주의: father 의 ptr[j] 가 curr 과 같음

// r\_sibling or l\_sibling 중 하나가 d 보다 많은 레코드 가지면 재분배 가능함!

r\_OK = 0; l\_OK = 0;

if (r\_sibling && r\_sibling->fill\_cnt > HALFK)

r\_OK = 1;

else if (l\_sibling && l\_sibling->fill\_cnt > HALFK)

l\_OK = 1;

if (r\_OK || l\_OK) {

//if (r\_sibling has more than d keys) {

if (r\_OK) {

redistribution(father, curr, r\_sibling, 'R', j);

}

else if (l\_OK) {

redistribution(father, l\_sibling, curr, 'L', j);

}

printf("Redistribution has been done.\n");

return 3; // Deletion succeeded with redistribution.

}

else { // Step 6: merging (합병이 필요함)

// Let leftptr be a pointer to left one of curr and sibling chosen to merge ;

// Let rightptr point to the right one of curr and sibling chosen to merge ;

if (r\_sibling) {

leftptr = curr; rightptr = /\* Fill your code \*/;

} // r\_sibling exists.

else {

leftptr = l\_sibling; rightptr = /\* Fill your code \*/;

} // surely l\_sibling exists.

// 아래 빈 곳은 leftptr, rightptr 두 형제를 leftptr 형제로 합병하는 부분이 와야 함!!

// 주의: 변수 i 가 두 형제 사이의 father 내의 중간 레코드를 가리키게 해 놓아야 함.

/\*

Fill

your

code

\*/

curr = father;

// Note that i has index of record in father to be deleted.

// Deletion of this record and pointer to its right will be done at start of next iteration.

} // else.

} while (!finished); // end of do-while 문.

} // B\_tree\_deletion

최대 원소수.

#define false 0

#define true 1

typedef struct element { // 레코드 정의. 회사명과 그 길이로 구성됨

char name[100]; // 회사명

int nleng; // 회사명 길이

}ele;

typedef struct node\* nodeptr; // \*nodeptr은 node의 형태를 갖는 포인터이다.

typedef struct node { // 일반 노드

int fill\_cnt;

ele rec[MAXK];

nodeptr ptr[MAXK + 1];

}node;

typedef struct big\_node { // 일반노드보다 레코드와 포인터가 하나씩 더 큰 빅노드 [합병시 사용]

nodeptr ptr[MAXK + 2];

ele rec[MAXK + 1];

}big\_node;

typedef struct two\_Bn {

nodeptr ptr[2 \* MAXK + 1];

ele rec[2 \* MAXK];

} two\_Bnode;

nodeptr root = NULL; // root는 전역 변수로 사용한다.

//nodeptr temp;

nodeptr stack[MAX]; //stack의 max값은 100

int top = -1;

void push(nodeptr node);

nodeptr pop();

void insert\_btree(); // file 전체를 삽입 이 함수는 insert\_arec을 호출한다.

int insert\_arec(ele in\_rec); // 레코드를 하나 삽입한다.

nodeptr retrieve(char\*, int\*); // 키값을 넣어 검색한다. [깊이와 함께 출력]

int seq\_scan\_btree(nodeptr curr); // 전체 레코드들을 출력한다.

int B\_tree\_deletion(char\*); // Function to delete a record with a given key.

int total\_height = 0; // 전체 높이

int main(void) {

char input, line[200], \* res\_gets;

ele a\_insert;

char name\_s[100];

int num, lleng;

int i, k, r, find;

nodeptr tp;

insert\_btree();

// 명령문 수행 루프.

while (1) {

fflush(stdin); // buffer clear

printf("\n명령을 넣으시오\n");

printf("insert: i 이름 / delete : d 이름 / retrieve : r 이름 / 전체출력: p / 종료: e >> ");

res\_gets = gets(line);

if (!res\_gets)

break; // no input.

lleng = strlen(line);

if (lleng <= 0)

continue; // empty line input. try again.

i = 1;

if (line[0] == 'E' || line[0] == 'e') { // exit program

printf("종료명령이 들어왔음!\n\n"); return 0;

}

else if (line[0] == 'I' || line[0] == 'i') { // 레코드 한개 삽입 명령.

// 먼저 회사명을 가져옴.

k = 0;

while (line[i] == ' ') i++;

while (i < lleng) {

name\_s[k] = line[i]; k++; i++;

}

name\_s[k] = '\0';

if (strlen(name\_s) == 0)

continue;

strcpy(a\_insert.name, name\_s);

a\_insert.nleng = strlen(name\_s);

top = -1;

r = insert\_arec(a\_insert); // 레코드 하나만 삽입하는 함수를 호출

if (r == 0)

printf("삽입 실패.\n", r);

else

printf("삽입 성공.\n", r);

}

else if (line[0] == 'D' || line[0] == 'd') { // 삭제 명령 수행

k = 0;

while (line[i] == ' ') i++;

while (i < lleng) {

name\_s[k] = line[i]; k++; i++;

}

name\_s[k] = '\0';

if (strlen(name\_s) == 0)

continue;

r = B\_tree\_deletion(name\_s);

if (r == 0)

printf("삭제요청 받은 키의 레코드가 존재하지 않음.\n", r);

else

printf("삭제 성공.\n");

}

else if (line[0] == 'R' || line[0] == 'r') { // 탐색 명령 수행

k = 0;

while (line[i] == ' ') i++;

while (i < lleng) {

name\_s[k] = line[i]; k++; i++;

}

name\_s[k] = '\0';

if (strlen(name\_s) == 0)

continue;

tp = retrieve(name\_s, &i);

if (tp)

printf("탐색성공. Name: %s, 길이: %d\n", tp->rec[i].name, tp->rec[i].nleng);

else

printf("탐색요청 받은 키를 가진 레코드가 존재하지 않음.\n");

}

else if (line[0] == 'P' || line[0] == 'p') // 프린트 명령 수행. 모든 레코드를 키 순서로 출력.

seq\_scan\_btree(root);

else

printf("input error \n");

} //while

} //main

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* stack 관련 함수들 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void push(nodeptr node) {

if (top > MAX) /// >>> MAX-1 이상( >= ) 이어야 하지 않나?

{

printf("stack is full\n");

return;

}

top++;

stack[top] = node;

}

nodeptr pop() {

int temp = 0;

if (top < 0)

{

printf("stack is empty.\n");

return 0;

}

temp = top; ///>>> 현재 top 이 가리키는 곳에 가장 최근의 데이타가 저장되어 있다. 따라서,

top--;

return stack[temp];

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// 레코드 하나를 삽입하는 함수이다.

// 반환값: 0: 삽입실패, 1, 2: 삽입성공 (1: 층증가 없이, 2:한 층 더 늘어 남.)

int insert\_arec(ele in\_rec) { //하나의 레코드를 삽입 key = 회사명

int i, j;

nodeptr curr, child, new\_ptr, tptr = NULL;

ele empty = { "\0",0 };

big\_node bnode;

if (!root) { // root가 NULL이면 btree가 비어있음. 맨 첫 노드를 만들어 여기에 넣는다.

root = /\* Fill your code \*/; // nodeptr형태로 node크기만큼 할당받아 시작주소는 root가 가짐

root->rec[0] = in\_rec; // key값을 root->rec[0]에 넣는다.

root->/\* Fill your code \*/ = root->/\* Fill your code \*/ = NULL; // p0과 p1에 NULL을 넣는다.

root->fill\_cnt = 1;

return 1; // 첫 노드를 만들어 넣고 종료함.

}

//root is not null

curr = root;

// 아래 빈 곳은 in\_rec 이 들어가면 좋을 리프노드를 찾아 curr가 가리키게 하는 부분이 와야 함!!

/\*

Fill

your

code

\*/

do {

// curr node is not full

if (curr->fill\_cnt < MAXK) {

for (i = 0; i < curr->fill\_cnt; i++)

if (strcmp(in\_rec.name, /\* Fill your code \*/ < 0)

break;

for (j = curr->fill\_cnt; j > i; j--) {

curr->ptr[j + 1] = /\* Fill your code \*/;

curr->rec[j] = /\* Fill your code \*/;

}

curr->rec[i] = /\* Fill your code \*/;

curr->ptr[i + 1] = /\* Fill your code \*/;

curr->fill\_cnt++;

return 1; // 삽입성공 (종류 1: 루트의 추가 없이 가능함).

}

else {

// curr node is full

for (i = 0; i < MAXK; i++) {

if (strcmp(in\_rec.name, /\* Fill your code \*/) < 0)

break;

}

bnode.ptr[0] = /\* Fill your code \*/;

for (j = 0; j < i; j++) {

bnode.rec[j] = /\* Fill your code \*/;

bnode.ptr[j + 1] = /\* Fill your code \*/;

}

bnode.rec[j] = /\* Fill your code \*/;

bnode.ptr[j + 1] = /\* Fill your code \*/;

j++;

while (i < MAXK) {

bnode.rec[j] = /\* Fill your code \*/;

bnode.ptr[j + 1] = /\* Fill your code \*/;

j++;

i++;

}

// 아래 빈 곳은 big node 를 3 부분으로 나누어 전반부는 curr 에, 가운데 레코드는 in\_rec에,

// 후반부는 새 노드에 넣고, child가 이 새 노드를 가리키게 하는 부분이 와야 함!!

/\*

Fill

your

code

\*/

if (top >= 0) { // 스택이 emtpy 가 아닐 경우

curr = pop(); // curr 의 부모로 curr를 변경함.

}

else { // 스택이 empty 임 (즉 curr 는 root 노드임.) 새 root 노드를 만들어 curr 의 부모로 함.

tptr = /\* Fill your code \*/;

tptr->rec[0] = /\* Fill your code \*/;

tptr->ptr[0] = /\* Fill your code \*/;

tptr->ptr[1] = /\* Fill your code \*/;

tptr->fill\_cnt = /\* Fill your code \*/;

root = /\* Fill your code \*/;

total\_height++;

return 2; // 삽입 성공 (종류 2: 새 루트가 생김)

} // else.

} // else.

} while (1);

return 0; // 이 문장을 수행할 경우는 없다.

} //함수 insert\_arec

void insert\_btree() { //파일전체의 레코드를 삽입 ->insert\_arec 을 호출

FILE\* fp;

ele data;

char name\_i[20], line[200];

char\* ret\_fgets;

int num, r;

double score;

int n = 0, lineleng;

int check, count = 0;

fp = fopen("Com\_names1.txt", "r");

if (fp == NULL) {

printf("Cannot open this file : Com\_names1.txt\n");

scanf("%d", &check);

} //if

root = NULL;

while (1) {

ret\_fgets = fgets(line, 200, fp);

if (!ret\_fgets)

break; // 파일 모두 읽음.

lineleng = strlen(line); // line 의 마지막 글자는 newline 글자임.

if (lineleng - 1 >= 100)

continue; // 회사명이 너무 길어서 무시

line[lineleng - 1] = '\0'; // 마지막 newline 글자 제거.

strcpy(data.name, line); // 삽입할 레코드 준비.

data.nleng = strlen(line);

top = -1; // 스택의 빈상태로 초기화.

r = insert\_arec(data); // 한 레코드를 비트리에 삽입한다.

if (r != 0)

count++; // 삽입성공 카운트 증가.

} //while

fp = fopen("Com\_names2.txt", "r");

if (fp == NULL) {

printf("Cannot open this file : Com\_names2.txt\n");

scanf("%d", &check);

} //if

while (1) {

ret\_fgets = fgets(line, 200, fp);

if (!ret\_fgets)

break; // 파일 모두 읽음.

lineleng = strlen(line); // line 의 마지막 글자는 newline 글자임.

if (lineleng - 1 >= 100)

continue; // 회사명이 너무 길어서 무시

line[lineleng - 1] = '\0'; // 마지막 newline 글자 제거.

strcpy(data.name, line); // 삽입할 레코드 준비.

data.nleng = strlen(line);

top = -1; // 스택의 빈상태로 초기화.

r = insert\_arec(data); // 한 레코드를 비트리에 삽입한다.

if (r != 0)

count++; // 삽입성공 카운트 증가.

} //while

printf("삽입 성공한 레코드 수 = %d \n\n", count);

fclose(fp);

} // 함수 insert\_btree

nodeptr retrieve(char\* skey, int\* idx\_found) { //검색 함수

nodeptr curr = root;

nodeptr P;

int i;

do {

for (i = 0; i < curr->fill\_cnt; i++) {

if (strcmp(skey, curr->rec[i].name) < 0)

break;

else if (strcmp(skey, curr->rec[i].name) == 0) {

\*idx\_found = i;

return curr;

}

else

; // do next i.

} // for i=

P = curr->ptr[i];

if (P) {

curr = P;

}

} while (P);

return NULL;

}//retrieve

int seq\_scan\_btree(nodeptr curr) {

int check\_stack = 0;

int i;

int n;

if (curr)

{

n = curr->fill\_cnt;

for (i = 0; i < n; i++) {

seq\_scan\_btree(curr->ptr[i]);

printf("Name : %s\n", curr->rec[i].name);

}

seq\_scan\_btree(curr->ptr[i]);

} //if(curr)

else if (!curr)

{

return 0;

}

return 0;

} //seq\_scan\_btree

// 좌측형제노드 내용, 부모의 레코드, 우측형제 내용을 받아서 재분배를 하는 함수

// wcase: curr 가 좌측형제와 재분배이면 ‘L’, curr 가 우측형제와 재분배이면 ‘R’.

// j : father 안에서 curr를 가리키는 포인터의 인덱스임.

void redistribution(nodeptr father, nodeptr l\_sibling, nodeptr r\_sibling, char wcase, int j) {

int i, k, m, n, h;

two\_Bnode twoB; // twobnode(bnode의 2배의 공간)

if (wcase == 'L')

j = j - 1;

else if (wcase == 'R')

j = j;

//copy l\_sibling's content, intermediate key in father, r\_sibling's content to twobnode;

for (i = 0; i < /\* Fill your code \*/; i++) {

twoB.ptr[i] = /\* Fill your code \*/;

twoB.rec[i] = /\* Fill your code \*/;

}

twoB.ptr[i] = /\* Fill your code \*/;

// 주의: j 에 father 에서의 l\_sibling 에 대한 index 가 들어 있음.

twoB.rec[i] = /\* Fill your code \*/; // 부모에서의 중간 키를 가져옴.

i++;

for (k = 0; k < /\* Fill your code \*/; k++, i++) {

twoB.ptr[i] = /\* Fill your code \*/;

twoB.rec[i] =/\* Fill your code \*/;

}

twoB.ptr[i] = /\* Fill your code \*/;

//Split twobnode into first half, middle record, second half;

h = i / 2; // h is the index of middle record.

//copy first half to left node;

for (n = 0; n < h; n++) {

l\_sibling->ptr[n] = /\* Fill your code \*/;

l\_sibling->rec[n] = /\* Fill your code \*/;

}

l\_sibling->ptr[n] = /\* Fill your code \*/;

l\_sibling->fill\_cnt = /\* Fill your code \*/;

//copy second half to r\_sibling;

n++;

for (m = 0; m < (i - h - 1); m++, n++) {

r\_sibling->ptr[m] = /\* Fill your code \*/;

r\_sibling->rec[m] = /\* Fill your code \*/;

}

r\_sibling->ptr[m] = /\* Fill your code \*/;

r\_sibling->fill\_cnt = /\* Fill your code \*/;

//move the middle record to father ;

father->rec[j] = /\* Fill your code \*/;

} // end of redistribution

int B\_tree\_deletion(char\* out\_key) {

nodeptr curr, r\_sibling, l\_sibling, father, Pt, leftptr, rightptr;

int i, j, k, r\_OK, l\_OK, found = 0, finished = 0;

curr = root;

// Step (0): search for a record (to be deleted) whose key equals out\_key.

do {

for (i = 0; i < /\* Fill your code \*/; i++)

if (strcmp(out\_key, /\* Fill your code \*/->rec[i].name) < 0)

break;

else if (strcmp(out\_key, /\* Fill your code \*/->rec[i].name) == 0) {

found = 1; break;

}

if (found == 1)

break; // 주의: 변수 i에 찾은 위치가 들어 있음.

else { // curr에 없다. child로 내려 가야 한다.

Pt = /\* Fill your code \*/;

if (Pt) {

push(/\* Fill your code \*/);

curr = Pt;

}

else

break;

}

} while (!found);

if (!found) {

return 0;

}

// Comes here when the key is found. It is in curr's node. i has index of rec to delete.

// Step (1): find successor of d\_rec.

if (curr->ptr[0]) { // curr node is not a leaf node

// We need to find successor of out\_key ;

Pt = /\* Fill your code \*/;

push(/\* Fill your code \*/);

// 가장 왼쪽 포인터를 따라내려 간다.

while (/\* Fill your code \*/) {

push(/\* Fill your code \*/);

Pt = /\* Fill your code \*/;

}

curr->rec[i] = Pt->rec[0];

curr = Pt;

i = 0;

} //end if

// curr 노드에서 index 가 i 인 레코드와 그 우측 포인터를 삭제하여야 한다.

finished = false;

do {

// Step (2):

//Remove record of index i and a pointer to its right from curr's node;

for (j = i + 1; j < /\* Fill your code \*/; j++) {

curr->rec[j - 1] = /\* Fill your code \*/;

/\* Fill your code \*/ = curr->ptr[j + 1];

}

curr->fill\_cnt = curr->fill\_cnt - 1;

// Step (3):

if (curr == root) {

if (curr->fill\_cnt == /\* Fill your code \*/) {

root = /\* Fill your code \*/;

free(curr);

}

return 1; // deletion succeeded.

}

// Step (4):

// curr is not the root.

if (curr->fill\_cnt >= HALFK) { return 2; } // Finish deletion with success.

// Now, curr violates minimum capacity constraint.

// Step (5):

father = pop(); // bring father of curr.

// r-sibling = pointer to right sibling of curr' node;

// l-sibling = pointer to left sibling of curr's node;

for (j = 0; j <= father->fill\_cnt; j++)

if (father->ptr[j] == curr) // find ptr of father which goes down to curr.

break;

if (j >= 1)

l\_sibling = father->ptr[j - 1];

else

l\_sibling = NULL;

if (j < father->fill\_cnt)

r\_sibling = father->ptr[j + 1];

else

r\_sibling = NULL;

// 주의: father 의 ptr[j] 가 curr 과 같음

// r\_sibling or l\_sibling 중 하나가 d 보다 많은 레코드 가지면 재분배 가능함!

r\_OK = 0; l\_OK = 0;

if (r\_sibling && r\_sibling->fill\_cnt > HALFK)

r\_OK = 1;

else if (l\_sibling && l\_sibling->fill\_cnt > HALFK)

l\_OK = 1;

if (r\_OK || l\_OK) {

//if (r\_sibling has more than d keys) {

if (r\_OK) {

redistribution(father, curr, r\_sibling, 'R', j);

}

else if (l\_OK) {

redistribution(father, l\_sibling, curr, 'L', j);

}

printf("Redistribution has been done.\n");

return 3; // Deletion succeeded with redistribution.

}

else { // Step 6: merging (합병이 필요함)

// Let leftptr be a pointer to left one of curr and sibling chosen to merge ;

// Let rightptr point to the right one of curr and sibling chosen to merge ;

if (r\_sibling) {

leftptr = curr; rightptr = /\* Fill your code \*/;

} // r\_sibling exists.

else {

leftptr = l\_sibling; rightptr = /\* Fill your code \*/;

} // surely l\_sibling exists.

// 아래 빈 곳은 leftptr, rightptr 두 형제를 leftptr 형제로 합병하는 부분이 와야 함!!

// 주의: 변수 i 가 두 형제 사이의 father 내의 중간 레코드를 가리키게 해 놓아야 함.

/\*

Fill

your

code

\*/

curr = father;

// Note that i has index of record in father to be deleted.

// Deletion of this record and pointer to its right will be done at start of next iteration.

} // else.

} while (!finished); // end of do-while 문.

} // B\_tree\_deletion