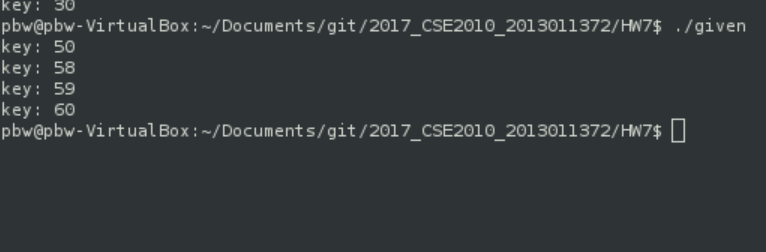
Hw7 B-tree

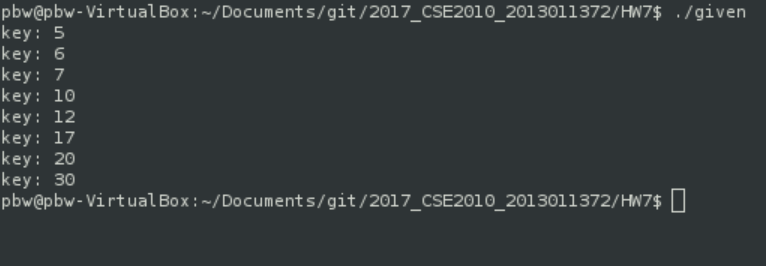
2013011372 박병욱

1. 과제 목적

* B-tree ADT(MAX\_KEYS:2 or 4)의 btInsert()함수를 구현한다. btInsert는 btInsertInternal 서브루틴을 가지고 있으며, btInsertInternal는 rearrage 서브 루틴을 가지고 있다. Rearrange()함수는 split된 btree를 연결해주고 재배열하는 역할을 한다.

1. 실행 결과

* Input.txt(MAX\_KEYS 2)
* 
* Input2.txt(MAX\_KEYS 4)



1. 소스 코드

#include <stdio.h>

#include <stdlib.h>

#include <assert.h>

#include <string.h>

#define MAX\_KEYS (2)

struct btNodeInternal {

int numKeys; /\* how many keys does this node contain? \*/

int keys[MAX\_KEYS];

struct btNode \*kids[MAX\_KEYS+1]; /\* kids[i] holds nodes < keys[i] \*/

};

struct btNode {

int isLeaf; /\* is this a leaf node? \*/

struct btNodeInternal keysAndKids;

};

/\* implementation of a B-tree \*/

typedef struct btNode \*bTree;

/\* create a new empty tree \*/

bTree btCreate(void);

/\* free a tree \*/

void btDestroy(bTree t);

/\* return nonzero if key is present in tree \*/

int btSearch(bTree t, int key);

/\* insert a new element into a tree \*/

void btInsert(bTree t, int key);

/\* print all keys of the tree in order \*/

void btPrintKeys(bTree t);

/\* changes the shape of the tree after btInsertInternal operation \*/

void rearrange(bTree b, bTree b2,int pos, int mid){

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

\* fill in the blank \*

\* STEP1: every key above pos moves up one space

memmove or copy(

Destination: &b->keys[pos+1],

Source: &b->keys[pos],

Size: sizeof(\*(b->keys)) \* (b->numKeys - pos));

\* STEP2: new kid goes in pos + 1 (location at pos +2)

memmove or copy(

Destination: &b->kids[pos+2],

Source: &b->kids[pos],

size: sizeof(\*(b->keys)) \* (b->numKeys - pos));

\* STEP3: rearrange bTree b

b->keys[pos] = mid;

b->kids[pos+1] = b2;

b->numKeys++;

END

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

memmove(&b->keysAndKids.keys[pos+1],&b->keysAndKids.keys[pos],\

sizeof(\*(b->keysAndKids.keys)) \* (b->keysAndKids.numKeys - pos));

memmove(&b->keysAndKids.kids[pos+2],&b->keysAndKids.kids[pos],\

sizeof(\*(b->keysAndKids.keys)) \* (b->keysAndKids.numKeys - pos));

b->keysAndKids.keys[pos] = mid;

b->keysAndKids.kids[pos+1] = b2;

b->keysAndKids.numKeys++;

}

void btPrintKeys(bTree p)

{

int i;

for (i = 0; i < p->keysAndKids.numKeys; i++)

{

if (p->isLeaf == 0)

{

btPrintKeys(p->keysAndKids.kids[i]);

}

printf("key: %d\n",p->keysAndKids.keys[i]);

}

if (p->isLeaf == 0)

{

btPrintKeys(p->keysAndKids.kids[i]);

}

}

bTree btCreate(void)

{

bTree b;

b = (bTree)malloc(sizeof(\*b));

assert(b);

b->isLeaf = 1;

b->keysAndKids.numKeys = 0;

return b;

}

void btDestroy(bTree b)

{

int i;

if(!b->isLeaf) {

for(i = 0; i < b->keysAndKids.numKeys + 1; i++) {

btDestroy(b->keysAndKids.kids[i]);

}

}

free(b);

}

/\* return smallest index i in sorted array such that key <= a[i] \*/

/\* (or n if there is no such index) \*/

static int searchKey(int n, int \*keys, int key)

{

int lo;

int hi;

int mid;

/\* invariant: keys[lo] < key <= keys[hi] \*/

lo = -1;

hi = n;

while(lo + 1 < hi) {

mid = (lo+hi)/2;

if(keys[mid] == key) {

return mid;

} else if(keys[mid] < key) {

lo = mid;

} else {

hi = mid;

}

}

return hi;

}

int btSearch(bTree b, int key)

{

int pos;

/\* have to check for empty tree \*/

if(b->keysAndKids.numKeys == 0) {

return 0;

}

/\* look for smallest position that key fits below \*/

pos = searchKey(b->keysAndKids.numKeys, b->keysAndKids.keys, key);

if(pos < b->keysAndKids.numKeys && b->keysAndKids.keys[pos] == key) {

return 1;

} else {

return(!b->isLeaf && btSearch(b->keysAndKids.kids[pos], key));

}

}

/\* insert a new key into a tree \*/

/\* returns new right sibling if the node splits \*/

/\* and puts the median in \*median \*/

/\* else returns 0 \*/

static bTree btInsertInternal(bTree b, int key, int \*median)

{

int pos;

int mid;

bTree b2;

pos = searchKey(b->keysAndKids.numKeys, b->keysAndKids.keys, key);

if(pos < b->keysAndKids.numKeys && b->keysAndKids.keys[pos] == key) {

/\* nothing to do \*/

return 0;

}

if(b->isLeaf) {

/\*\* everybody above pos moves up one space\*\*\*\*\*

\* \*

\* fill in the blank \*

\* \*

\*STEP1: \*

\*memory copy or move \*

Source: &b->keysAndKids.keys[pos], \*

Destination: &b->keysAndKids.keys[pos+1], \*

Size: b->keys \* numKeys - pos \*

\*

\*STEP2: \*

b->keys[pos] = key; \*

\*

\*STEP3: \*

b->numKeys++; \*

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

memmove( &b->keysAndKids.keys[pos+1], &b->keysAndKids.keys[pos], \

sizeof(b->keysAndKids.keys) \* (b->keysAndKids.numKeys - pos) );

b->keysAndKids.keys[pos] = key;

b->keysAndKids.numKeys++;

} else {

/\* insert in child \*/

b2 = btInsertInternal(b->keysAndKids.kids[pos], key, &mid);

/\* insert a new key in b \*/

if(b2) {

rearrange(b,b2,pos,mid);

}

}

/\* we waste a tiny bit of space by splitting now

\* instead of on next insert \*/

if(b->keysAndKids.numKeys >= MAX\_KEYS) {

mid = b->keysAndKids.numKeys/2;

\*median = b->keysAndKids.keys[mid];

/\* make a new node for keys > median \*/

/\* picture is:

\*

\* 3 5 7

\* A B C D

\*

\* becomes

\* (5)

\* 3 7

\* A B C D

\*/

b2 = (bTree)malloc(sizeof(\*b2));

b2->keysAndKids.numKeys = b->keysAndKids.numKeys - mid - 1;

b2->isLeaf = b->isLeaf;

memmove(b2->keysAndKids.keys, &b->keysAndKids.keys[mid+1]

, sizeof(\*(b->keysAndKids.keys)) \* b2->keysAndKids.numKeys);

if(!b->isLeaf) {

memmove(b2->keysAndKids.kids, &b->keysAndKids.kids[mid+1]

, sizeof(\*(b->keysAndKids.kids)) \* (b2->keysAndKids.numKeys + 1));

}

b->keysAndKids.numKeys = mid;

return b2;

} else {

return 0;

}

}

void btInsert(bTree b, int key)

{

bTree b1; /\* new left child \*/

bTree b2; /\* new right child \*/

int median;

b2 = btInsertInternal(b, key, &median);

if(b2) {

/\* basic issue here is that we are at the root \*/

/\* so if we split, we have to make a new root \*/

b1 = (bTree)malloc(sizeof(\*b));

assert(b1);

/\* copy root to b1 \*/

memmove(b1, b, sizeof(\*b));

/\* make root point to b1 and b2 \*/

b->isLeaf = 0;

b->keysAndKids.numKeys = 1;

b->keysAndKids.keys[0] = median;

b->keysAndKids.kids[0] = b1;

b->keysAndKids.kids[1] = b2;

}

}

int main(int argc, char \*\*argv)

{

bTree b;

char command;

int key;

FILE \*input;

input = fopen("input.txt", "r");

b = btCreate();

while (1) {

command = fgetc(input);

if (feof(input)) break;

switch (command) {

case 'i':

fscanf(input, "%d", &key);

btInsert(b, key);

break;

default:

break;

}

}

btPrintKeys(b);

btDestroy(b);

return 0;

}

1. 코드 설명

* Rearrange()

1. Keys[pos]부터 맨 뒤까지 있는 key들을 다 keys[pos+1]로 옮긴다.
2. Kids[pos]부터 맨 뒤까지 있는 kids들을 다 kids[pos+2]로 옮긴다.
3. Keys[pos]에 mid를 넣고, kids[pos+1]에 b2를 넣는다.numkeys를 증가한다.

* BtInsertInternal
* If(b->isLeaf)

1. Keys[pos]부터 맨 뒤까지 있는 key들을 다 keys[pos+1]로 옮긴다.
2. Keys[pos]에 key를 넣는다
3. Numkeys를 증가시킨다