**Main**

#include<stdio.h>

#include<stdlib.h>

#include "binTree.h"

#include "Heap.h"

#pragma warning (disable:4996)

#define STUDENTLEN 5

#define COURSELEN 8

#define M 5

node\* students[M] = { NULL };

//heap

heapAvg\* h;

int hash(unsigned long Id, int size)

{

int s = 1;

Id /= 10;

while (Id)

{

if (Id % 10 != 0)

s \*= (Id % 10);

Id /= 10;

}

return s % M;

}

student\* getStudent(unsigned long id)

{

return binSearch(students[hash(id, M)], id);

}

void updateAll()

{

for (int i = 0; i < h->size; i++)

{

unsigned long tempId = h->key[i].id;

int indexInStudents = hash(tempId,M);

binSearch(students[indexInStudents], tempId) -> locationInHeap = i;

}

}

void main()

{

//students

student\* arrStudents[STUDENTLEN];

for (int i = 0; i < STUDENTLEN; i++)

arrStudents[i] = newStudent(100000000 + 2 \* 3 \* 4 \* 5 \* 6 \* 7 \* rand() % 100000000);

//courses

course arrCourses[COURSELEN];

for (int i = 0; i < COURSELEN; i++)

{

arrCourses[i].grade = rand() % 100;

arrCourses[i].numberOfCourse = rand() % 10;

}

//adding the courses to the students

for (int i = 0; i < STUDENTLEN; i++)

for (int j = 0; j < COURSELEN; j++)

if (rand()%2)

addCourse(arrStudents[i], arrCourses[j]);

//stdAvg

stdAvg arrStdAvg[STUDENTLEN];

for (int i = 0; i < STUDENTLEN; i++)

{

arrStdAvg[i].average = getAverage(arrStudents[i]);

arrStdAvg[i].id = arrStudents[i]->ID;

}

heapInit(h);

/\*heapInsert(h, f1);

heapInsert(h, f2);

heapInsert(h, f3);

heapInsert(h, f4);

heapInsert(h, f5);\*/

for (int i = 0; i < STUDENTLEN; i++)

{

h->key[i] = arrStdAvg[i];

h->size++;

}

printh(h);

for (int j = 0; j < h->size; j++)

heapifyUp(h, j);

for (int i = 0; i < M; i++)

insert(students[hash(arrStudents[i]->ID, M)], arrStudents[i]);

updateAll();

for (int i = 0; i < h->size; i++)

printStudent(arrStudents[i]);

printf("\n");

printh(h);

}

**Student.h**

#pragma once

#include<malloc.h>

struct Course

{

int numberOfCourse;

int grade;

} typedef course;

struct Student

{

unsigned long ID;

course\* arrayOfCourses;

int numOfCourses;

int MaxSizeOfCourses;

int locationInHeap;

} typedef student;

struct StudentdAvg

{

unsigned long id;

double average;

}typedef stdAvg;

student\* newStudent(long id)

{

student\* sptr = (student\*)malloc(sizeof(student));

sptr->ID = id;

sptr->locationInHeap = -1;

sptr->numOfCourses = 0;

sptr->MaxSizeOfCourses = 10;

sptr->arrayOfCourses = (course\*)malloc(sizeof(course) \* 10);

return sptr;

}

void addCourse(student\* student, course s)

{

if (student->numOfCourses % 10 == 0 && student->numOfCourses != 0)

{

student->MaxSizeOfCourses += 10;

student->arrayOfCourses = (course\*)realloc(student->arrayOfCourses, sizeof(course) \* student->MaxSizeOfCourses);

}

student->arrayOfCourses[student->numOfCourses] = s;

student->numOfCourses++;

}

double getAverage(student\* s)

{

double avg = 0;

for (int i = 0; i < s->numOfCourses; i++)

avg += s->arrayOfCourses[i].grade;

avg /= s->numOfCourses;

return avg;

}

void printCourses(student\* s)

{

for (int i = 0; i < s->numOfCourses; i++)

{

printf("\tcourse number:%d\n\tcourse grade: %d\n", s->arrayOfCourses[i].numberOfCourse, s->arrayOfCourses[i].grade);

if (i != s->numOfCourses - 1)

printf("\t-----\n");

}

}

void printStudent(student\* s)

{

printf("----------------------------\n");

printf("student ID: %ld\n", s->ID);

printf("name: moshe cohen :D\n");

printf("course:\n");

printCourses(s);

printf("location in heap: %d\n", s->locationInHeap);

printf("average: %lf\n----------------------------", getAverage(s));

}

**Heap.h**

#pragma once

#include<malloc.h>

struct HeapAverage

{

stdAvg\* key;

int size; //the heap real size (number of items)

int capacity; //the heap total capacity

}typedef heapAvg;

void heapInit(heapAvg\* &h)

{

h = (heapAvg\*)malloc(sizeof(heapAvg));

h->capacity = 10;

h->key = (stdAvg\*)malloc(sizeof(stdAvg) \* h->capacity);

h->size = 0;

}

void heapInsert(heapAvg\* &h, stdAvg avg)

{

if (h->size == h->capacity)

{

h->capacity += 10;

h->key = (stdAvg\*)realloc(h->key, sizeof(int) \* h->capacity);

}

int i = h->size;

while (i >= 1 && h->key[(i - 1) / 2].average <= avg.average)

{

h->key[i] = h->key[(i - 1) / 2];

i = (i - 1) / 2;

}

h->key[i] = avg;

h->size++;

}

void printh(heapAvg\* h)

{

for (int i = 0; i < h->size; i++)

printf("id: %ld\taverage: %lf\n", h->key[i].id,h->key[i].average);

}

void swap(heapAvg\* h, int x, int y)

{

stdAvg temp = h->key[x];

h->key[x] = h->key[y];

h->key[y] = temp;

}

void heapifyDown(heapAvg\* h, int i) {

int largest = i;

int switched = 1;

while (i < h->size && switched) {

int right = i \* 2 + 2;

int left = i \* 2 + 1;

if (left < h->size && h->key[left].average > h->key[i].average)

largest = left;

if (right < h->size && h->key[right].average > h->key[i].average)

largest = right;

if (largest != i)

{

int temp = h->key[i].average;

h->key[i] = h->key[largest];

h->key[largest].average = temp;

i = largest;

}

else

switched = 0;

}

}

void buildHeap(heapAvg\* h)

{

for (int i = (h->size - 1 - 1) / 2; i >= 0; i--)

heapifyDown(h, i);

}

int heapifyUp(heapAvg\* h, int i) {

while (i && h->key[(i - 1) / 2].average < h->key[i].average)

{

int temp = h->key[i].average;

h->key[i] = h->key[(i - 1) / 2];

h->key[(i - 1) / 2].average = temp;

i = (i - 1) / 2;

}

return i;

}

/\* reminder:

left = 2x+1

right = 2x+2

parent = (i-1)/2

\*/

**binTree.h**

#pragma once

#include<stdio.h>

#include "student.h"

student\* defS = newStudent(123); // default student, for checks only

struct Node

{

student\* key;

Node\* left;

Node\* right;

}typedef node;

node\* newNode(student\* s)

{

node\* n = (node\*)malloc(sizeof(node));

n->key = s;

n->left = NULL;

n->right = NULL;

return n;

}

void insert(node\*& n, Student\* s)

{

if (n == NULL)

{

n = newNode(s);

return;

}

node\* parent = NULL;

node\* x = n;

while (x != NULL)

{

parent = x;

if (s->ID <= x->key->ID)

x = x->left;

else x = x->right;

}

if (s->ID <= parent->key->ID) //put in left

parent->left = newNode(s);

else parent->right = newNode(s);//put in right

}

student\* binSearch(node\* root, unsigned long id)

{

if (root == NULL)

{

printf("ERROR binSearch\n");

return NULL;

}

node\* x = root;

while (x != NULL)

{

if (x->key->ID == id)

return x->key;

if (id <= x->key->ID)

x = x->left;

else x = x->right;

}

printf("ERROR!");

return x->key;

}