ГУАП

КАФЕДРА № 43

ОТЧЕТ
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

стр. преподаватель		Е.О. Шумова
должность, уч. степень, звание	подпись, дата	инициалы, фамилия

ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ №7

КАЧЕСТВО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ. ОЦЕНКА КАЧЕСТВА ПРОГРАММНОГО ПРОДУКТА.

по курсу: ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ ИНФОРМАЦИОННЫХ СИСТЕМ

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ ГР. № 4831		К.А. Корнющенков
	подпись, дата	инициалы, фамилия

Санкт-Петербург 2020

Цель работы

Целью данной работы является изучение методологии оценки качества программного продукта на основе одной из существующих методик.

Задание

Оценить качество приложенного программного продукта по перечисленным в методике параметрам.

Оценки качества программного продукта

1 Показатели качества

1.1 Показатели “завершенность”
- Использование всех исходных данных в вычислениях.
- Проверка переменных – границ циклов на допустимый диапазон.
- Проверка исходных данных на допустимый диапазон.

1.2 Показатели “стандартизация”
- Одинаковое представление физических и математических констант.
- Не одинаковые имена для разных по смыслу переменных.
- Одинаковые имена для одинаковых по смыслу переменных.
- Общее функциональное назначение всех элементов массива.

1.3 Показатели “рациональность”
- Оптимизация часто используемых подпрограмм и фрагментов исходного кода.

1.4 Показатели “доступность”
- Исключение использования чисел, подверженных изменениям (например, A*3.14).

1.5 Показатели “коммуникативность”
- Четкость и полезность сообщений об ошибках.
- Не требование указания количества входных данных.

1.6 Показатели “структурированность”
- Существование хотя бы одной точки выхода из подпрограммы.
- Соответствие оверлейной структуры и последовательности выполнения программ.
- Соответствие подпрограмм их функциональному назначению.

1.7 Показатели “информативность”
- Существование комментария для каждого модуля (назначение , входы, выходы, метод).
- Описание зависимостей модулей.
- Соответствие имен объектов их назначению.

1.8 Показатели “осмысленность”
- Все операторы выполнимы при тестировании.
- Выполнение вычислений, не относящихся к циклу ,вне его.

1.9 Показатели “открытость”
- Одно присваивание в одной строке.

- Один оператор в одной строке.

2 Расчетные показатели качества.

2.1 Показатели “надежность”

Устойчивость к искажающим воздействиям :

P(1)=1-D/K
где :
D – число экспериментов, в которых искажающее воздействие приводило к отказу.
K – число экспериментов с искажающим воздействием.

P(1) = 1-1/6 = 0.83;

Вероятность безотказной работы :

P=1-Q/N
где :
Q – число зарегистрированных отказов.
N – число экспериментов.

P = 1-1/5 = 0,8;

Среднее время восстановления :

$$Q_b = \frac{T_b^{\text{доп}}}{T_b}, \text{если } T_b > T_b^{\text{доп}}$$

$$Q_b = 1, \text{если } T_b \leq T_b^{\text{доп}}$$

Где:
$T_b^{\text{доп}}$ – допустимое среднее время восстановления,
T_b – среднее время восстановления,
$T_b = \frac{1}{N} * \sum T_b$
N – число восстановлений,
T_{bi} – время восстановления после отказа.

$T_b = \frac{1}{4} * 1 = 0{,}25$
$T_b^{\text{доп}} = 0{,}2$
$Q_b = \frac{0{,}2}{0{,}25} = 0{,}8$

Оценка продолжительности преобразования входных данных в выходные :

$$Q_n = \frac{T_{ni}^{\text{доп}}}{T_{ni}}, \text{если } T_{ni} > T_{ni}^{\text{доп}}$$

$$Q_{ni} = 1, \text{если } T_{ni} \leq T_{ni}^{\text{доп}}$$

Где:

$T_{ni}^{\text{доп}}$ – допустимая продолжительность преобразования i-го входного набора данных,

T_{ni} –фактическая продолжительность преобразования i-го входного набора данных.

$T_{ni}^{\text{доп}} = 0{,}2$
$T_{ni} = 0{,}8$
$Q_n = \frac{0{,}2}{0{,}8} = \frac{1}{4}$

2.2 Показатели “сопровождаемость”

Оценка простоты программы по числу точек входа и выхода :

W=1/(D+1)*(F+1)

где :
D – общее число точек входа,
F – общее число точек выхода.

W = 1/(1 + 1)*(2 + 1) = 1/8

Оценка простоты по числу условных операторов :

K=(1-A/B)

где :
A – общее число точек входа ,
B – общее число точек выхода.

K = (1-1/4) = 0,75

Отношение количества тестированных модулей к общему количеству модулей:

Q_{T^M}/Q_{O^M}

где :
Q_{T^M} – количество тестированных модулей,
Q_{O^M} – общее количество модулей.

$\frac{Q_{T^M}}{Q_{O^M}} = \frac{12}{15} = 0{,}8$

Отношение количества тестированных логических блоков к общему количеству логических блоков:

$Q_{T^Б}/Q_{O^Б}$

где :
$Q_{T^Б}$ – количество тестированных модулей,
$Q_{O^Б}$ – общее количество модулей.

$$\frac{Q_{T^{\text{Б}}}}{Q_{O^{\text{Б}}}} = \frac{1}{4} = 0.25$$

3 Экспериментальные показатели качества

3.1 Показатели “надежность”
- контроль полноты входных данных,
- контроль корректности входных данных,
- контроль непротиворечивости входных данных,

3.2 Показатели “сопровождаемость”
- описание интерфейса с пользователем,
- возможность управления подробностью получаемых выходных данных.

3.3 Показатели “эффективность”
- время выполнения,
- время реакции на действия пользователя,
- оценка числа потенциальных пользователей,
- оценка числа функции,

3.4 Показатели “корректность”
- отсутствие ошибок в описании действий пользователя, генерации, настройки.

Выводы

В ходе выполнения работы изучили методологии оценки качества программного продукта на основе одной из существующих методик.

ПРИЛОЖЕНИЕ А

Ссылка на источник кода - https://github.com/xtaci/algorithms/blob/master/include/btree.h

```
#ifndef
ALGO_BTREE_H__
                #define ALGO_BTREE_H__

                #include <stdio.h>
                #include <assert.h>
                #include <stdint.h>
                #include <stdlib.h>
                #include <string.h>
                #include <fcntl.h>
                #include <unistd.h>
                #include <memory>

                #define BLOCKSIZE     4096
                #define T                     255
                #define LEAF          0x0001
                #define ONDISK        0x0002
                #define MARKFREE      0x0004

                namespace alg {
                class BTree {
                private:
                // 4K node, 4096 bytes to write
                // t = 255
                struct node_t {
                uint16_t n;                         // num key
                uint16_t flag;              // flags
                uint32_t offset;            // lseek offset related to file beginning
                char padding[12];           // padding to 4096
                int32_t key[509];           // key
                int32_t c[510];                     // childs pointers (represented as
                file offsets)
                } __attribute__ ((packed));
                typedef struct node_t *node;

                public:
                // node and index
                struct Res {
                uint32_t offset;
                int32_t idx;
                };
                private:
                int fd;
                private:
                BTree(const BTree &);
```

```
BTree& operator=(const BTree&);
public:
BTree(const char * path) {
fd = open(path, O_RDWR|O_CREAT, 0640);
if (fd == -1)
return;
node x = (node)ALLOCBLK();
int n = read(fd,x,BLOCKSIZE);
if (n != BLOCKSIZE) { // init new btree
x->flag |= LEAF;
WRITE(x);
}
}

~BTree() {
close(fd);
}

Res Search(int32_t x) {
node root = ROOT();
return search(root, x);
}

void Insert(int32_t k) {
node r = ROOT();
if (r->n == 2*T - 1) {
// place the old root node to the end of the file
r->flag &= ~ONDISK;
WRITE(r);
// new root
node s = (node)ALLOCBLK();
s->flag &= ~LEAF;
s->flag |= ONDISK;    // write to offset 0
s->offset = 0;
s->n = 0;
s->c[0] = r->offset;
split_child(s, 0);    // split_child with write s
insert_nonfull(s, k);
} else {
insert_nonfull(r, k);
}
}

void DeleteKey(int32_t k) {
node root = ROOT();
delete_op(root, k);
}
```

```
private:
/**
* search a key, returns node and index
*/
Res search(node x, int32_t k) {
int32_t i = 0;
Res ret;
while (i<x->n && k > x->key[i]) i++;

if (i<x->n && k == x->key[i]) {     // search in [0,n-1]
ret.offset = x->offset;
ret.idx = i;
return ret;
} else if (x->flag & LEAF) { // leaf, no more childs
ret.offset = 0;
ret.idx = -1;
return ret;
} else {
std::auto_ptr<node_t> xi(READ(x, i));      // search in a child
return search(xi.get(), k);
}
}

/**
* insert into non-full node
*/
void insert_nonfull(node x, int32_t k) {
int32_t i = x->n-1;
if (x->flag & LEAF) { // insert into this leaf
while (i>=0 && k < x->key[i]) {     // right shift to
x->key[i+1] = x->key[i];     // make place for k
i = i - 1;
}
x->key[i+1] = k;
x->n = x->n + 1;
WRITE(x);
} else {
while(i>=0 && k < x->key[i]) {
i = i-1;
}
i=i+1;
node xi = READ(x, i);        // insert the key into one child.
if (xi->n == 2*T-1) {
split_child(x, i);
if (k > x->key[i]) {
i = i+1;
}
// NOTICE!
```

```
// reload x[i] after split_child.
xi = READ(x, i);
}
insert_nonfull(xi, k);
delete xi;
}
}

/**
* split a node into 2.
*/
void split_child(node x, int32_t i) {
std::auto_ptr<node_t> z((node)ALLOCBLK());
std::auto_ptr<node_t> y(READ(x, i));
z->flag &= ~LEAF;
z->flag |= (y->flag & LEAF);
z->n = T - 1;

int32_t j;
for (j=0;j<T-1;j++) { // init z, t-1 keys
z->key[j] = y->key[j+T];
}

if (!(y->flag & LEAF)) {     // if not leaf, copy childs too.
for (j=0;j<T;j++) {
z->c[j] = y->c[j+T];
}
}

y->n = T-1;    // shrink y to t-1 elements
WRITE(y.get());
WRITE(z.get());

for (j=x->n;j>=i+1;j--) { // make place for the new child in x
x->c[j+1] = x->c[j];
}

x->c[i+1] = z->offset; // make z the child of x
for (j=x->n-1;j>=i;j--) { // move keys in x
x->key[j+1] = x->key[j];
}
x->key[i] = y->key[T-1];     // copy the middle element of y into x
x->n = x->n+1;
WRITE(x);
}

/**
* recursive deletion.
```

```
*/
void delete_op(node x, int32_t k) {
int32_t i;
/*
int t;
printf("key:%d n:%d\n",k, x->n);
for (t=0;t<x->n;t++) {
printf("=%d=", x->key[t]);
}
printf("\n");
*/

if (x->n == 0) {      // emtpy node
return;
}

i = x->n - 1;
while (i>=0 && k < x->key[i]) { // search the key.
i = i - 1;
}

if (i >= 0 && x->key[i] == k) {     // key exists in this node.
if (x->flag & LEAF) {
//printf("in case 1 [%d] [%d]\n", i,x->n);
case1(x, i, k);
} else {
//printf("in case 2 [%d] [%d]\n", i,x->n);
case2(x, i, k);
}
} else {
// case 3. on x.c[i+1]
case3(x, i+1, k);
}
}

/**
* case 1.
* If the key k is in node x and x is a leaf, delete the key k from x.
*/
void case1(node x, int32_t i, int32_t k) {
int j;
for (j = i;j<x->n-1;j++) {   // shifting the keys only, no childs
available.
x->key[j] = x->key[j+1];
}
x->n = x->n - 1;
WRITE(x);
}
```

```
void case2(node x, int32_t i, int32_t k) {
// case 2a:
// If the child y that precedes k in node x has at least t
// keys, then find the predecessor k0 of k in the subtree
// rooted at y. Recursively delete k0, and replace k by k0 in x.
// (We can find k0 and delete it in a single downward pass.)
std::auto_ptr<node_t> y(READ(x, i));
if (y->n >= T) {
int32_t k0 = y->key[y->n-1];
//printf("case2a %d %d\n", k0, x->key[i]);
x->key[i] = k0;
WRITE(x);
delete_op(y.get(), k0);
return;
}

// case 2b.
// If y has fewer than t keys, then, symmetrically, examine
// the child z that follows k in node x. If z has at least t keys,
// then find the successor k0 of k in the subtree rooted at z.
// Recursively delete k0, and replace k by k0 in x. (We can find k0
// and delete it in a single downward pass.)
std::auto_ptr<node_t> z(READ(x, i+1));
if (z->n >= T) {
int32_t k0 = z->key[0];
//printf("case2b %d %d\n", k0, x->key[i]);
x->key[i] = k0;
WRITE(x);
delete_op(z.get(), k0);
return;
}

// case 2c:
// Otherwise, if both y and z have only t-1 keys,
// merge k and all of z into y,  so that x loses both k and the
// pointer to z, and y now contains 2t - 1 keys.
// Then free z and recursively delete k from y.
if (y->n == T-1 && z->n == T-1) {
//printf("case2c");
// merge k & z into y
y->key[y->n] = k;

int j;
for (j=0;j<z->n;j++) {              // merge keys of z
y->key[y->n+1+j] = z->key[j];
}
for (j=0;j<z->n+1;j++) {     // merge childs of z
```

```
y->c[y->n+1+j] = z->c[j];
}

// mark free z
z->flag |= MARKFREE;
y->n = y->n + z->n + 1; // size after merge
WRITE(z.get());
WRITE(y.get());

for (j=i;j<x->n-1;j++) { // delete k from node x
x->key[j] = x->key[j+1];
}

for (j=i+1;j<x->n;j++){      // delete pointer to z --> (i+1)th
x->c[j] = x->c[j+1];
}
x->n = x->n - 1;
WRITE(x);

// recursive delete k
delete_op(y.get(), k);
return;
}

// cannot reach here
assert(false);
}

void case3(node x, int32_t i, int32_t k) {
std::auto_ptr<node_t> ci(READ(x, i));
if (ci->n > T-1) {    // ready to delete in child.
delete_op(ci.get(), k);
return;
}

// case 3a.
// If x.c[i] has only t - 1 keys but has an immediate sibling with at
least t keys,
// give x.c[i] an extra key by moving a key from x down into x.c[i],
moving a
// key from x.c[i]’s immediate left or right sibling up into x, and moving
the
// appropriate child pointer from the sibling into x.c[i].
std::auto_ptr<node_t> left(READ(x, i-1));
if (i-1>=0 && left->n >= T) {
// printf("case3a, left");
// right shift keys and childs of x.c[i] to make place for a key
// right shift ci childs
```

```
int j;
for (j=ci->n-1;j>0;j--) {
ci->key[j] = ci->key[j-1];
}

for (j=ci->n;j>0;j--) {
ci->c[j] = ci->c[j-1];
}
ci->n = ci->n+1;
ci->key[0] = x->key[i-1];                 // copy key from x[i-1] to
ci[0]
ci->c[0] = left->c[left->n];        // copy child from left last child.
x->key[i] = left->key[left->n-1];   // copy left last key into x[i]
left->n = left->n-1;                        // decrease left size

WRITE(ci.get());
WRITE(x);
WRITE(left.get());
delete_op(ci.get(), k);
return;
}

// case 3a. right sibling
std::auto_ptr<node_t> right(READ(x, i+1));
if (i+1<=x->n && right->n >= T) {
// printf("case3a, right");
ci->key[ci->n] = x->key[i];         // append key from x
ci->c[ci->n+1] = right->c[0];       // append child from right
ci->n = ci->n+1;
x->key[i] = right->key[0];          // subsitute key in x

int j;
for (j=0;j<right->n-1;j++) { // remove key[0] from right sibling
right->key[j] = right->key[j+1];
}

for (j=0;j<right->n;j++) { // and also the child c[0] of the right
sibling.
right->c[j] = right->c[j+1];
}
right->n = right->n - 1;     // reduce the size of the right sibling.

WRITE(ci.get());
WRITE(x);
WRITE(right.get());
delete_op(ci.get(), k);      // recursive delete key in x.c[i]
return;
}
```

```
// case 3b.
// If x.c[i] and both of x.c[i]’s immediate siblings have t-1 keys, merge
x.c[i]
// with one sibling, which involves moving a key from x down into the new
// merged node to become the median key for that node.
if ((i-1<0 ||left->n == T-1) && (i+1 <=x->n || right->n == T-1)) {
if (left->n == T-1) {
// copy x[i] to left
left->key[left->n] = x->key[i];
left->n = left->n + 1;

// remove key[i] from x and also the child
// shrink the size & set the child-0 to left
delete_i(x, i);

int j;
// append x.c[i] into left sibling
for (j=0;j<ci->n;j++) {
left->key[left->n + j] = ci->key[j];
}

for (j=0;j<ci->n+1;j++) {
left->c[left->n + j] = ci->c[j];
}
left->n += ci->n;     // left became 2T-1
ci->flag |= MARKFREE; // free ci
ci->n = 0;
WRITE(ci.get());
WRITE(x);
// root check
if (x->n == 0 && x->offset ==0) {
left->flag |= MARKFREE;
WRITE(left.get());
left->flag &= ~MARKFREE;
left->offset = 0;
}
WRITE(left.get());
delete_op(left.get(), k);
return;
} else if (right->n == T-1) {
// copy x[i] to x.c[i]
ci->key[ci->n] = x->key[i];
ci->n = ci->n + 1;
// remove key[i] from x and also the child
// shrink the size & set the child-0 to ci
delete_i(x, i);
```

```
int j;
// append right sibling into x.c[i]
for (j=0;j<right->n;j++) {
ci->key[ci->n + j] = right->key[j];
}

for (j=0;j<right->n+1;j++) {
ci->c[ci->n + j] = right->c[j];
}
ci->n += right->n;                  // ci became 2T-1
right->flag |= MARKFREE;     // free right
right->n = 0;
WRITE(right.get());
WRITE(x);
// root check
if (x->n == 0 && x->offset ==0) {
ci->flag |= MARKFREE;
WRITE(ci.get());
ci->flag &= ~MARKFREE;
ci->offset = 0;
}
WRITE(ci.get());
delete_op(ci.get(), k);
return;
}
}
}

/**
* delete ith key & child.
*/
void delete_i(node x, int32_t i)  {
int j;
for (j=i;j<x->n-1;j++) {
x->key[j] = x->key[j+1];
}

for (j=i+1;j<x->n;j++) {
x->c[j] = x->c[j+1];
}
x->n = x->n - 1;
}

/**
* Allocate empty node struct.
* A better allocator should be consider in practice, such as
* re-cycling the freed up blocks on disk, so used blocks
* should be traced in some data strucuture, file header maybe.
```

```
*/
void * ALLOCBLK() {
node x = new node_t;
x->n = 0;
x->offset = 0;
x->flag = 0;
memset(x->key, 0, sizeof(x->key));
memset(x->c, 0, sizeof(x->c));
memset(x->padding, 0xcc, sizeof(x->padding));
return x;
}

/**
* Load the root block
*/
node ROOT() {
void *root = ALLOCBLK();
lseek(fd, 0, SEEK_SET);
read(fd, root, BLOCKSIZE);
return (node)root;
}

/**
* Read a 4K-block from disk, and returns the node struct.
*/
node READ(node x, int32_t i) {
void *xi = ALLOCBLK();
if (i >=0 && i <= x->n) {
lseek(fd, x->c[i], SEEK_SET);
read(fd, xi, BLOCKSIZE);
}
return (node)xi;
}

/**
*       update a node struct to file, create if offset is -1.
*/
void WRITE(node x) {
if (x->flag & ONDISK) {
lseek(fd, x->offset, SEEK_SET);
} else {
x->offset = lseek(fd,0, SEEK_END);
}
x->flag |= ONDISK;
write(fd, x, BLOCKSIZE);
}
};
}
```

```
#endif //
```