자료구조 기말과제 / 2018920065 루안리치

INDEX

1. 소스코드와 설명

FIFO :

```
// 2018920065 루안리치 - FIFO

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define CACHE_SIZE 8192
#define TABLE_SIZE 997        // prime number

typedef struct cache_buffer{
    unsigned long blkno;
    int data;       // data buffer
    struct cache_buffer *next, *prev;
    struct cache_buffer *hash_next, *hash_prev;
}cache_buffer;

cache_buffer *hash_table[TABLE_SIZE];    // hash tables
cache_buffer *cache_head;    // cache_buffer's head, as a pointer

int ascii_change(int c){        // change ascii to demical number
    int value = c-48;
    if(value>=0){
        return value;
    }
    else{
        return 0;
    }
```

```
}

int sum_of(int c[]){        // make a number list to an integer value
    int sum=0;
    if(c[0]==1){
        for(int i=0;i<9;i++){
            for(int j=0;j<8-i;j++){
                c[j]=c[j]*10;
            }
        }
        for(int m=0;m<9;m++){
            sum=sum+c[m];
        }
    }
    else{
        for(int i=0;i<8;i++){
            for(int j=0;j<7-i;j++){
                c[j]=c[j]*10;
            }
        }
        for(int m=0;m<8;m++){
            sum=sum+c[m];
        }
    }
    return sum;
}

int hash_function(int key){     // for hash table
    return (key % TABLE_SIZE);
}

void init_buffer(){         // initialize cache buffer
    cache_head = (cache_buffer *)malloc(sizeof(cache_buffer));
    cache_head->next = cache_head;
    cache_head->prev = cache_head;
    cache_head->blkno = -1;
    cache_head->data = -1;
}
```

Back to index

```
void init_hash_table(){          // initialize hash_table[]
    for(int i=0;i<TABLE_SIZE;i++){
        hash_table[i] = (cache_buffer *)malloc(sizeof(cache_buffer));
        hash_table[i]->hash_next = hash_table[i];
        hash_table[i]->hash_prev = hash_table[i];
        hash_table[i]->blkno = -1;
        hash_table[i]->data = -1;
    }
}

void hash_chain_add(cache_buffer *newnode, int hash){    // set newnode's hash pointer
    newnode->hash_next = hash_table[hash];
    newnode->hash_prev = hash_table[hash]->hash_prev;
    hash_table[hash]->hash_prev->hash_next = newnode;
    hash_table[hash]->hash_prev = newnode;
}

void node_add(int value, unsigned long nbr){          // add an newnode
    cache_buffer *newnode = (cache_buffer *)malloc(sizeof(cache_buffer));  // malloc a new cache_buffer*
    newnode->next = cache_head;
    newnode->prev = cache_head->prev;
    cache_head->prev->next = newnode;              // used be : head->prev ⇔ head
    cache_head->prev = newnode;                    // now : head->prev ⇔ newnode ⇔ head

    int hash = hash_function(value);      // get the hash value
    hash_chain_add(newnode, hash);        // use hash_chain_add function

    newnode->blkno = nbr; // set newnode's blkno as number
    newnode->data = value;// set newnode's data as value
}

void node_chain_delete(cache_buffer *target){          // delete and set the pointers in cache buffer
    target->prev->next = target->next; // before : target->prev ⇔ target ⇔ target->next
    target->next->prev = target->prev; // after : target->prev ⇔ target->next
}

void hash_chain_delete(cache_buffer *target){          // delete and set the pointers in hash table
```

```
    target->hash_prev->hash_next = target->hash_next;   // similar to node_chain_delete
    target->hash_next->hash_prev = target->hash_prev;
}

void node_delete(){     // delete (head->next) node
    cache_buffer *node;
    node = cache_head->next;
    hash_chain_delete(node);// free hash pointer
    node_chain_delete(node);        // free buffer pointer
    free(node);
}

int search_hash(int value){     // searching function
    int hash = hash_function(value);

    cache_buffer *keynode;
    keynode = hash_table[hash]->hash_prev;
    while(keynode != hash_table[hash]){         // find the node that has data == value
        if(keynode->data != value){
            keynode = keynode->hash_prev;
        }
        else if(keynode->data == value){            // if found
            return 1;    // return found
        }
    }

    return 0;       // not found
}

void hash_chain_print(cache_buffer *ht, int i){         // print out hash table
    cache_buffer *node;                                 // like [1]->x->x->x->
    printf("[%d]->",i);
    for(node=ht->hash_next;node!=ht;node=node->hash_next){
        printf("%lu->",node->blkno);     // print blkno that easy to read
    }
    printf("₩n");
}
```

```
int main() {
    FILE *fp;
    int c;   // getc
    int cnt=0;   // record fseek
    int i;    // save_value[i]
    int size=0;     // cache size < 8192
    int check=0;     // check if c == eof = 1 -> stop
    int save_value[10];         // to get int value
    int value;
    int search_result;   // 1 for found, 0 for not found
    unsigned long nbr=0;      // node's number as blkno
    double hit=0, miss=0, total;
    double hit_r, miss_r;

    fp = fopen("/Users/ruan/test_trace.txt", "r");       // open file
    if(fp==NULL){
        printf("file reading error₩n");     // check
        return -1;
    }
    fseek(fp, 0, SEEK_SET);

    init_buffer();
    init_hash_table();      // init
    while(check==0){
        i=0;
        do{
            c=getc(fp);
            cnt++;

            if(c=='₩n'){      // a line end
                fseek(fp, cnt, SEEK_SET);
                value=sum_of(save_value);      // get the value
                search_result = search_hash(value);          // search

                if(search_result == 0){    // not found -> add to cache buffer
                    if(size<CACHE_SIZE){        // has empty place
                        node_add(value,nbr);  // value->hash, number personally
                    }
```

```
                    else{    // no empty place
                        node_delete();   // delete last node (cache_head->next)
                        size--;      // so a place release
                        node_add(value,nbr);  // add the node
                    }
                    size++;
                    nbr++;
                    miss++;
                }
                else{    // found
                    hit++;
                }
                break;
            }
            else if(c==EOF){
                    check = 1; // never repeat
            }
            else{
                save_value[i]=ascii_change(c);
                i++;
            }
        }while(c!=EOF);
    }

    printf("2018920065 루안리치 / 자료구조 기말과제₩n");
    printf("----------FIFO simulation----------₩n");
    total = hit+miss;
    hit_r = hit/total;
    miss_r = miss/total;
    printf("₩n₩n<<< hit ratio = %lf , miss ratio = %lf >>>₩n₩n",hit_r,miss_r);
    printf("total access = %.1lf, hit = %.1lf, miss = %.1lf₩n",total,hit,miss);
    printf("HIT RATIO = %lf₩n₩n₩n", hit_r);

    fclose(fp);
    return 0;
}
```

LRU :

```
// 2018920065 루안리치 - LRU
// green line means different with FIFO

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define CACHE_SIZE 8192
#define TABLE_SIZE 997        // prime number

typedef struct cache_buffer{
    unsigned long blkno;
    int data;       // data buffer
    struct cache_buffer *next, *prev;
    struct cache_buffer *hash_next, *hash_prev;
}cache_buffer;

cache_buffer *hash_table[TABLE_SIZE];    // hash tables
cache_buffer *cache_head;    // cache_buffer's head, as a pointer

int ascii_change(int c){          // change ascii to demical number
    int value = c-48;
    if(value>=0){
        return value;
    }
    else{
        return 0;
    }
}

int sum_of(int c[]){       // make a number list to an integer value
    int sum=0;
    if(c[0]==1){
        for(int i=0;i<9;i++){
            for(int j=0;j<8-i;j++){
                c[j]=c[j]*10;
```

```
            }
        }
        for(int m=0;m<9;m++){
            sum=sum+c[m];
        }
    }
    else{
        for(int i=0;i<8;i++){
            for(int j=0;j<7-i;j++){
                c[j]=c[j]*10;
            }
        }
        for(int m=0;m<8;m++){
            sum=sum+c[m];
        }
    }
    return sum;
}

int hash_function(int key){     // for hash table
    return (key % TABLE_SIZE);
}

void init_buffer(){         // initialize cache buffer
    cache_head = (cache_buffer *)malloc(sizeof(cache_buffer));
    cache_head->next = cache_head;
    cache_head->prev = cache_head;
    cache_head->blkno = -1;
    cache_head->data = -1;
}

void init_hash_table(){          // initialize hash_table[]
    for(int i=0;i<TABLE_SIZE;i++){
        hash_table[i] = (cache_buffer *)malloc(sizeof(cache_buffer));
        hash_table[i]->hash_next = hash_table[i];
        hash_table[i]->hash_prev = hash_table[i];
        hash_table[i]->blkno = -1;
        hash_table[i]->data = -1;
```

```
    }
}

void hash_chain_add(cache_buffer *newnode, int hash){     // set newnode's hash pointer
    newnode->hash_next = hash_table[hash];
    newnode->hash_prev = hash_table[hash]->hash_prev;
    hash_table[hash]->hash_prev->hash_next = newnode;
    hash_table[hash]->hash_prev = newnode;
}

void node_add(int value, unsigned long nbr){          // add an newnode
    cache_buffer *newnode = (cache_buffer *)malloc(sizeof(cache_buffer));  // malloc a new cache_buffer*
    newnode->next = cache_head;
    newnode->prev = cache_head->prev;
    cache_head->prev->next = newnode;                 // used be : head->prev ⇔ head
    cache_head->prev = newnode;                       // now : head->prev ⇔ newnode ⇔ head

    int hash = hash_function(value);      // get the hash value
    hash_chain_add(newnode, hash);        // use hash_chain_add function

    newnode->blkno = nbr; // set newnode's blkno as number
    newnode->data = value;// set newnode's data as value
}

void node_add_top(cache_buffer *target, int hash){   // similar to node_add but dosen't set blkno and data
    target->next = cache_head;                       // means only set the new pointer (both cache buffer and hash)
    target->prev = cache_head->prev;                 // to reset the pointers but not blkno and data
    cache_head->prev->next = target;                 // for LRU
    cache_head->prev = target;

    target->hash_next = hash_table[hash];
    target->hash_prev = hash_table[hash]->hash_prev;
    hash_table[hash]->hash_prev->hash_next = target;
    hash_table[hash]->hash_prev = target;
}

void node_chain_delete(cache_buffer *target){         // delete and set the pointers in cache buffer
    target->prev->next = target->next; // before : target->prev ⇔ target ⇔ target->next
```

```
    target->next->prev = target->prev; // after : target->prev ⇔ target->next
}

void hash_chain_delete(cache_buffer *target){          // delete and set the pointers in hash table
    target->hash_prev->hash_next = target->hash_next;  // similar to node_chain_delete
    target->hash_next->hash_prev = target->hash_prev;
}

void node_delete(){     // delete (head->next) node
    cache_buffer *node;
    node = cache_head->next;
    hash_chain_delete(node);// free hash pointer
    node_chain_delete(node);        // free buffer pointer
    free(node);
}

void move_to_top(cache_buffer *target, int hash){     // is used when LRU simulation
    hash_chain_delete(target);      // free hash pointer
    node_chain_delete(target);      // free buffer pointer
    node_add_top(target, hash);     // reset the pointers
}

int search_hash(int value){     // searching function
    int hash = hash_function(value);

    cache_buffer *keynode;
    keynode = hash_table[hash]->hash_prev;
    while(keynode != hash_table[hash]){         // find the node that has data == value
        if(keynode->data != value){
            keynode = keynode->hash_prev;
        }
        else if(keynode->data == value){              // if found
            move_to_top(keynode, hash);       // for LRU; if FIFO, delete this line
            return 1;     // return found
        }
    }

    return 0;       // not found
```

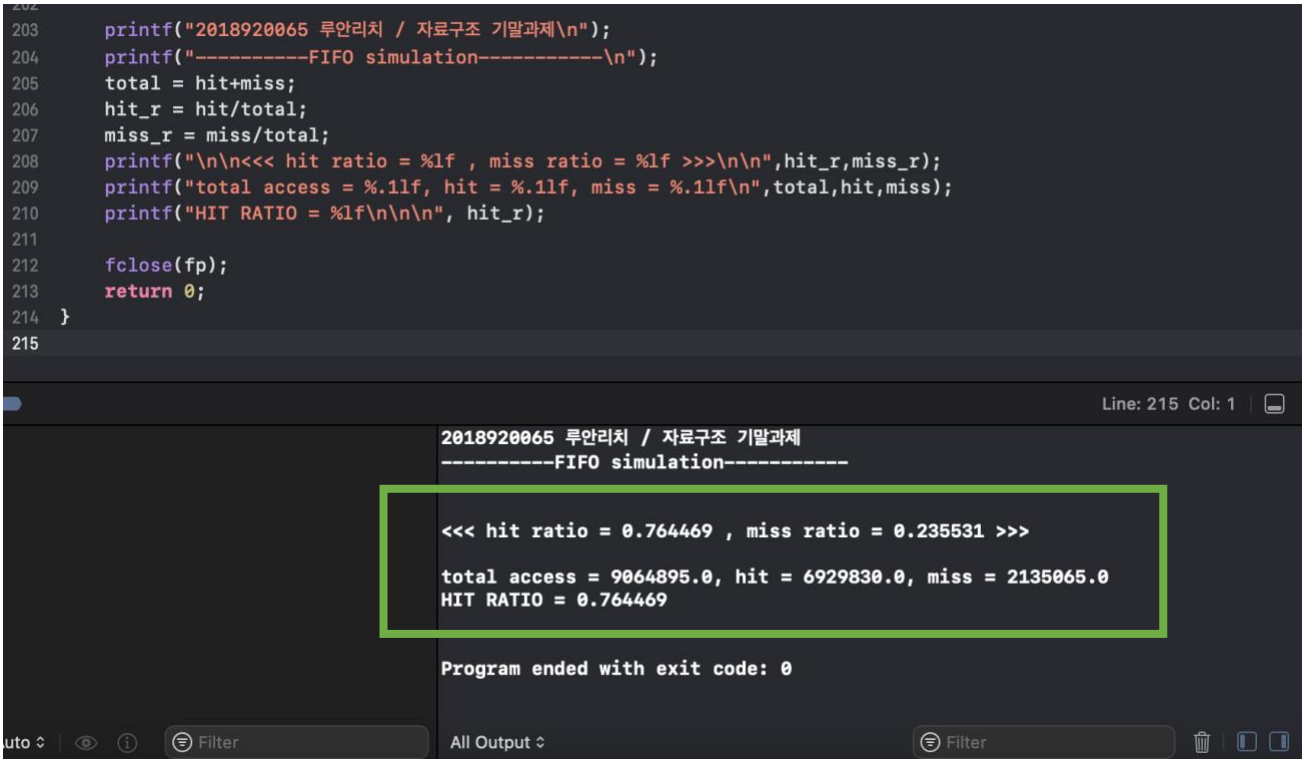
```
}

void hash_chain_print(cache_buffer *ht, int i){          // print out hash table
    cache_buffer *node;                                  // like [1]->x->x->x->
    printf("[%d]->",i);
    for(node=ht->hash_next;node!=ht;node=node->hash_next){
        printf("%lu->",node->blkno);     // print blkno that easy to read
    }
    printf("₩n");
}

int main() {
    FILE *fp;
    int c;   // getc
    int cnt=0;   // record fseek
    int i;    // save_value[i]
    int size=0;     // cache size < 8192
    int check=0;      // check if c == eof = 1 -> stop
    int save_value[10];         // to get int value
    int value;
    int search_result;    // 1 for found, 0 for not found
    unsigned long nbr=0;      // node's number as blkno
    double hit=0, miss=0, total;
    double hit_r, miss_r;

    time_t start, end;
    struct tm *timestart, *timeend;

    fp = fopen("/Users/ruan/test_trace.txt", "r");       // open file
    if(fp==NULL){
        printf("file reading error₩n");     // check
        return -1;
    }
    fseek(fp, 0, SEEK_SET);

    init_buffer();
    init_hash_table();     // init
```

```
while(check==0){
    i=0;
    do{
        c=getc(fp);
        cnt++;

        if(c=='₩n'){      // a line end
                fseek(fp, cnt, SEEK_SET);
                value=sum_of(save_value);      // get the value
                search_result = search_hash(value);          // search

                if(search_result == 0){    // not found -> add to cache buffer
                    if(size<CACHE_SIZE){        // has empty place
                        node_add(value,nbr);  // value->hash, number personally
                    }
                    else{    // no empty place
                        node_delete();   // delete last node (cache_head->next)
                        size--;      // so a place release
                        node_add(value,nbr);  // add the node
                    }
                    size++;
                    nbr++;
                    miss++;
                }
                else{    // found
                    hit++;
                }
                break;
        }
        else if(c==EOF){
                check = 1; // never repeat
        }
        else{
            save_value[i]=ascii_change(c);
            i++;
        }
    }while(c!=EOF);
}
```

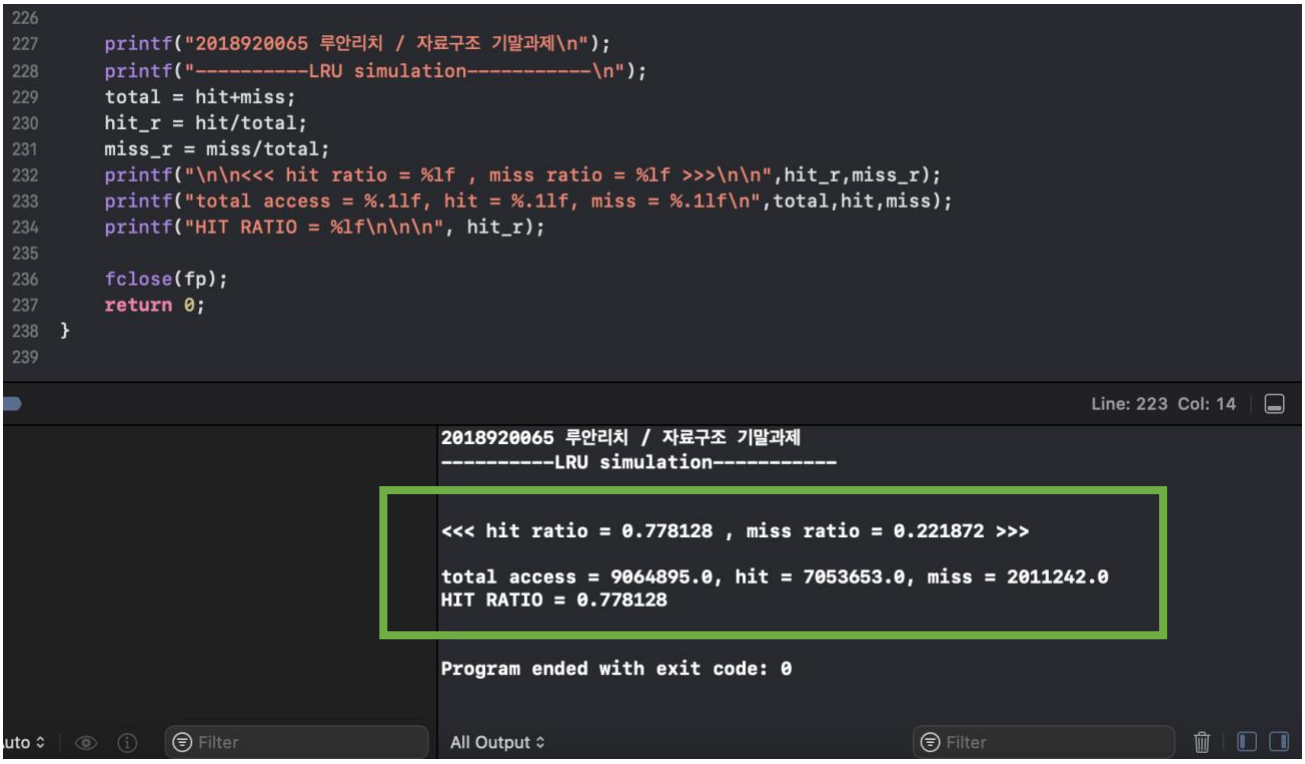
```
    printf("2018920065 루안리치 / 자료구조 기말과제₩n");
    printf("----------LRU simulation----------₩n");
    total = hit+miss;
    hit_r = hit/total;
    miss_r = miss/total;
    printf("₩n₩n<<< hit ratio = %lf , miss ratio = %lf >>>₩n₩n",hit_r,miss_r);
    printf("total access = %.1lf, hit = %.1lf, miss = %.1lf₩n",total,hit,miss);
    printf("HIT RATIO = %lf₩n₩n₩n", hit_r);

    fclose(fp);
    return 0;
}
```

2. 시뮬레이션 결과 화면

FIFO:



LRU:



3. LRU/FIFO 구현 및 시뮬레이션 결과보고

LRU :

데이터 들어오면 일단 hash_function 실행하고 해당 hash table 안에서 같은 값 갖는 노드 있는지 확인 →

1. 있다. Hit! → move that node to top → hit++ → next data
2. 없다
   a. Cache buffer is full → node_delete → node_add
   b. Cache buffer isn't full → node_add

FIFO :

데이터 들어오면 일단 hash_function 실행하고 해당 hash table 안에서 같은 값 갖는 노드 있는지 확인 →

1. 있다. Hit! → hit++ → next data
2. 없다
   a. Cache buffer is full → node_delete → node_add
   b. Cache buffer isn't full → node_add

따라서 **차이점은 hit 할 때 LRU 가 해당 노드를 맨 위로 옮긴다.**

	FIFO	LRU
HIT RATIO	76%	78%

해시로 구현하고 table_size 소수 997 로 설정하여 실제로 실행하는 결과 FIFO 의 HIT RATIO 는 0.764469 이고 LRU 의 HIT RATIO 는 0.778128 이다. 예상하고 같이 LRU 의 HIT RATIO 가 FIFO 의 HIT RATIO 보다 높다.