Experiment 5, LRU algorithm simulation

Experimental purpose and requirements

Simulate the page replacement algorithm LRU with a high-level language to deepen the understanding of the LRU algorithm. Ask to output the change sequence of the stack.

Experimental principle

The basic principle is: if a certain page is accessed, it is likely to be accessed; on the contrary, if it has not been accessed for a long time, it is unlikely to be accessed in the near future.

Experiment code:

```
#include <iostream>
using namespace std;

#define OK 1
#define TRUE 1
#define ERROR -1
#define INIT_STACK_SIZE 100
#define SIZEINCREMENT 10
typedef int Elemtype;
typedef int Status;
typedef struct {
    Elemtype *base;
    Elemtype *top;
    int size;
}SqStack;

//initialization
Status InitStack(SqStack &s) {
    s.base = (Elemtype *)malloc(INIT_STACK_SIZE * sizeof(Elemtype));
    if (!s.base) exit(OVERFLOW);
    s.top = s.base;
    s.size = INIT_STACK_SIZE;
    return TRUE;
}

//get the top element of the stack
Status Gettop(SqStack s, Elemtype &e) {
    if (s.top == s.base) return ERROR;
    e = *(s.top - 1);
    return TRUE;
}
```

```
//Push stack
Status Push(SqStack &s, Elemtype e) {
    if (s.top - s.base >= s.size) {
        s.base = (Elemtype *)realloc(s.base, (s.size + SIZEINCREMENT) * sizeof(Elemtype));
        if (!s.base) return ERROR;
        s.top = s.base + s.size;
        s.size += SIZEINCREMENT;
    }
    *s.top++ = e;
    return TRUE;
}

//Pop stack
Status Pop(SqStack &s, Elemtype &e) {
    if (s.top == s.base) return ERROR;
    e = *--s.top;
    return TRUE;
}
Status Clear( SqStack & s ) { //Specially used in this question, clear the bottom of the
stack when there is only one element left in the stack
    if ( s.top == s.base ) return ERROR ;
    *-- s.top ;
    return TRUE ;
}

//Loop through the output
Status PrintStack( SqStack & s ) {
    Elemtype *i;
    if ( s.top == s.base ) return ERROR ;
    cout << "stack: " ;
    for (i = s.base ; i < s.top ; i++) {
        cout << *i << " " ;
    }
    return TRUE ;
}

//Corresponding to this question, only output 3 numbers
Status ShowStack(SqStack &s) {
    Elemtype *i;
    if (s.top == s.base) return ERROR;
    cout << "stack: ";
    for (i = s.top-1; i >s.top-4; i--) {
        cout << *i << " ";
    }
```

```
    return TRUE ;
}

// check if the element exists
int ExistStack( SqStack & s , int num , int square ) {
    if ( s .top == s .base) //if the stack is empty
    {
        return -1; //Stack is empty -> add target directly
    }
    Elemtype *i;
    for (i = s.base ; i < s.top ; i++) {
        if (*i == num ) {
            if (i == s.top - 1) {
                return 0; // exists, it is the top of the stack -> no operation
            }
            else
                return 1; // exists, not the top of the stack -> the target moves to the
top of the stack
        }
    }
    if ( s.top - s.base == square )
        return 2; //The stack is full, does not exist -> go to the bottom of the stack,
add the target to the top of the stack
    else
        return -1; //The stack is not full, does not exist -> directly add the target
}
int LRU( SqStack * s1 , int num , int square ) {
    SqStack s2, s3;
    InitStack(s2);
    InitStack(s3);
    int e = -1;
    int times = 0;

    //The stack is empty or the stack is not full, does not exist -> directly add the target
    if (ExistStack(* s1 , num , square ) == -1) {
        Push(* s1 , num );
    }

    //Exist, it is the top of the stack -> no operation
    else if (ExistStack(* s1 , num , square ) == 0) {
        NULL ;
    }

    // exists, not the top of the stack -> the target moves to the top of the stack
```

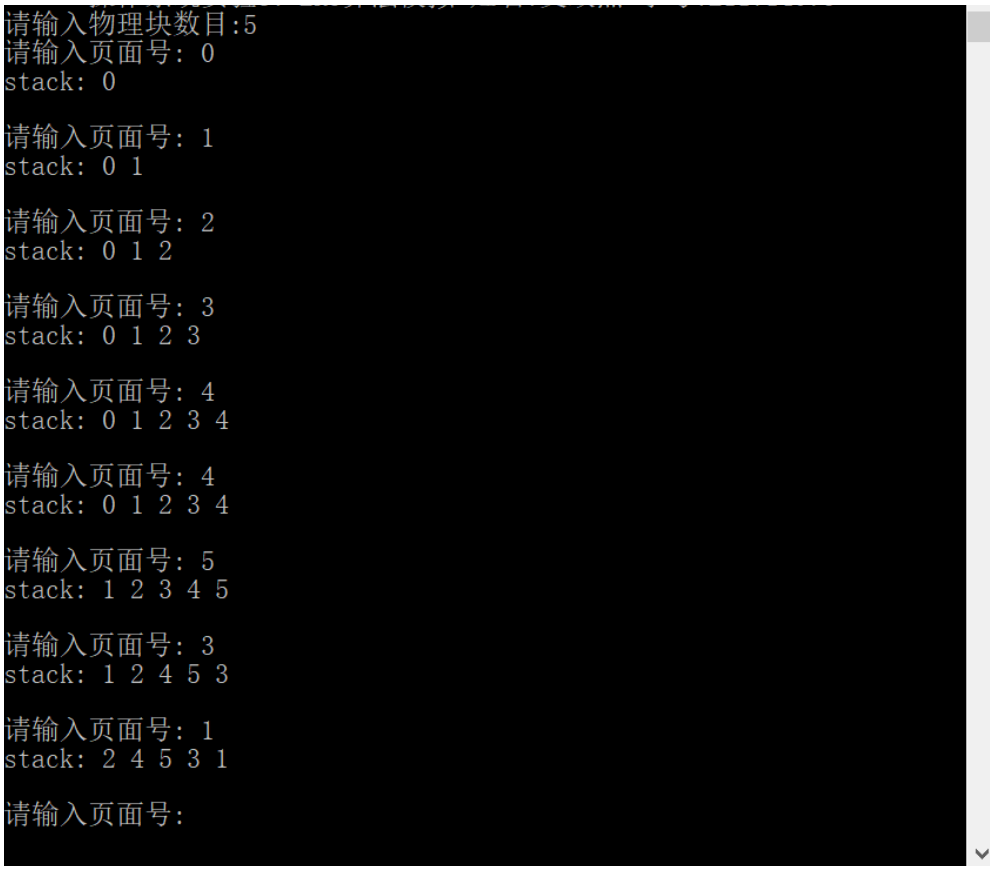
```
    else if (ExistStack(* s1 , num , square ) == 1)
    {
        /*
        //If the stack head is the target, no operation
        Gettop(*s1, e);
        if (e == num) {
            return OK;
        }
        */
        //If the target is after the stack head, take out the target and push it back on
the stack
        {
            e = -1;
            while (1) {
                Pop(* s1 , e);
                if (e == num )
                    break ;
                Push(s2, e);
                times++;
            }
            for ( int i = 0; i < times; i++) {
                Pop(s2, e);
                Push(* s1 , e);
            }

            Push(* s1 , num );
        }
    }

    //The stack is full, does not exist -> go to the bottom of the stack, add the target
to the top of the stack
    else if (ExistStack(* s1 , num , square ) == 2) {
        for ( int i = 1; i < square ; i++) {
            Pop(*s1, e);
            Push(s2, e);
        }
        Clear(*s1);
        for (int i = 1; i < square; i++) {
            Pop(s2, e);
            Push(*s1, e);
        }
        Push(*s1, num);
    }
```

```
    PrintStack(*s1);
    cout << endl;
    return 1;
}
int main() {
    cout <<"***** Operating System Experiment 5: LRU Algorithm Simulation Name: XXX Student
ID: AAA *****" << endl;
    SqStack s1;
    InitStack(s1);
    int square;
    cout << "Please enter the number of physical blocks:" ;
    cin >> square;
    int num;
    cout << "Please enter the page number: " ;
    while (1) {
        cin >> num;
        if (num == -1)
            break ;
        LRU(&s1, num, square);
        cout << endl << "Please enter the page number: " ;
    }
    return 0;
}
```

Screenshot of the program running: (As shown in the figure, taking the physical block number 5 as an example, various possible situations have been tested)



summary:

Problems encountered:

The biggest problem we encountered in this experiment is that if we start the experiment directly, the logic will be unclear, and the subsequent addition of code will lead to limited efficiency.

In order to better complete this experiment, before the experiment, we need to clarify the internal logic of this experiment, and optimize the logic method accordingly, so as to clarify the process, and then design the code, and make the program concise and clear, and improve the running speed .

The following is my combing of the algorithm logic of this question:

Finally, I found that there are four possible operation situations for this question, and I classified them one by one (-1, 0, 1, 2) and applied them to my own program design.