자료구조 HW1

2016124145 양해찬

1.

```
#include <stdio.h>
#include <stdlib.h>
void transpose(int a[], int b[], int row, int column) //1차원 배열 주소
와 행, 열 정보를 인자로 받아 전치
{
    int i, j;
    int k = 0;
    for (i = 0;i < column;i++)
    {
        for (j = 0;j < row;j++)
        {
            b[row * i + j] = a[j * column + i];
            //b에 a행렬을 전치해서 저장
        }
    }
}
int main()
{
    int i, j, k = 0;
    int row, column;//row=a행렬 열, b 행렬 행 column=a행렬 행, b 행렬 열
    int* a;
    int* b;
    printf("Input the size of matrix: ");
    scanf_s("%d %d", &row, &column);
    a = (int*)malloc(sizeof(int) * row * column);//1차원 배열을 입력받기
위해 메모리 동적할당
    b = (int*)malloc(sizeof(int) * row * column);//a행렬을 전치해서 저장
하기 위한 배열
    printf("Input elements of the %d x %d matrix: ", row, column);//행
열 입력(a)
    for (i = 0;i < row * column;i++)//1차원 배열로 가정해 입력한다
    {
        scanf_s("%d", &a[i]);
    }
    transpose(a, b, row, column);//a행렬을 전치해서 b행렬에 저장
    int** arr = (int**)malloc(sizeof(int*) * column);//b행렬을 2차원 행
렬로 만들기 위해 세로방향으로 메모리 동적할당
    for (i = 0;i < column;i++)
```

```
    {
        arr[i] = (int*)malloc(sizeof(int) * row);//가로방향으로 메모리 동
적할당
        for (j = 0;j < row;j++)
        {
            arr[i][j] = b[k];//b행렬을 2차원 행렬로 변환
            k++;
        }
    }
    for (i = 0;i < column;i++)//2차원 행렬 출력
    {
        for (j = 0;j < row;j++)
        {
            printf("% d ", arr[i][j]);
        }
        printf("₩n");
    }
    for (i = 0;i < column;i++)
    {
        free(arr[i]);//가로방향 메모리 해제
    }
    free(arr);//세로방향 메모리 해제
    free(a);//a 행렬 메모리 해제
    free(b);//b 행렬 메모리 해제
}
```

Microsoft Visual Studio 디버그 콘솔

```
Input the size of matrix: 3 4
Input elements of the 3 x 4 matrix: 1 3 2 4 2 3 4 1 2 4 2 33
 1  2  2
 3  3  4
 2  4  2
 4  1  33

C:₩Users₩y9711₩source₩repos₩Project4₩Debug₩Project4.exe(프로세
```

2. (1), (2)

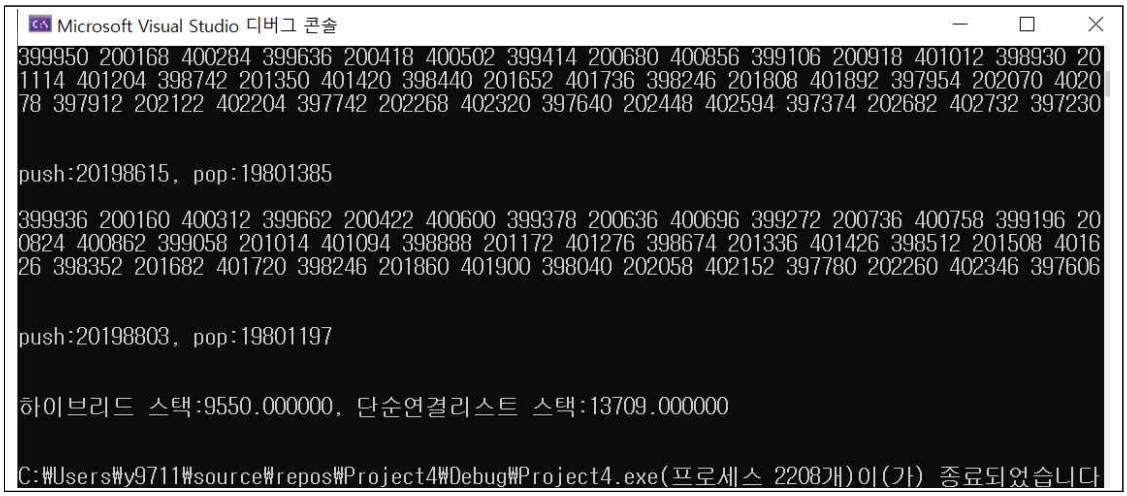
```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define MAX 1000000
typedef struct _node* nodeptr;
typedef struct _node {
    int value[10];
    nodeptr next;
} node;
typedef struct _node_l {//단순연결리스트용
    int key;
    struct _node_l* next_l;
} node_l;
node_l* head_l, * tail_l;
void init_stack_l(void) {//단순연결리스트용
    int key;
    head_l = (node_l*)malloc(sizeof(node_l));
    tail_l = (node_l*)malloc(sizeof(node_l));
    head_l->next_l = tail_l;
    tail_l->next_l = tail_l;
}
int push_l(int k) {//단순연결리스트용
    int key;
    node_l* t;
    if ((t = (node_l*)malloc(sizeof(node_l))) == NULL) {
        printf("\n Out of memory...");
        return -1;
    }
    t->key = k; t->next_l = head_l->next_l; head_l->next_l = t;
    return k;
}
int pop_l(void) {//단순연결리스트용
    int key;
    node_l* t;
    int i;
    if (head_l->next_l == tail_l) /* if empty */ {
        printf("\n Stack underflow.");
        return -1;
```

```
    }
    t = head_l->next_l; i = t->key;
    head_l->next_l = t->next_l; free(t);
    return i;
}
void clear_stack_l(void) {//단순연결리스트용
    int key;
    node_l* t, * s;
    t = head_l->next_l;
    while (t != tail_l) {
        s = t; t = t->next_l;
        free(s);
    }
    head_l->next_l = tail_l;
}
typedef struct {
    int num;
    nodeptr head;
} stack;
int pop(stack* stk)
{
    node* n;//head를 가리키기 위한 노드 n선언
    n = stk->head;
    int i;
    if ((stk->num % 10 == 0) && (stk->num != 0))//stk->num이 10의 배수(노
드가 꽉 참)일때
    {
        stk->head = n->next;//stk->head를 통과시킴
        stk->num--;//가장마지막에 들어간것이 가장 먼저 나와야하므로
        free(n);//n 메모리 해제
        return stk->head->value[9];//pop한 값을 return
    }
    else if (stk->num == 0)//stk->num이 0일 때(노드가 비었을 때)
    {
        free(n);
        free(stk);//빈 노드이므로 모두 해제해준다
    }
    else
```

```
    {
        i = stk->num % 10;
        stk->num--;
        return n->value[i - 1];//pop 한 값을 return
    }
}
void push(stack* stk, int v) {
    if (stk->num % 10 == 0) {
        node* n = (node*)malloc(sizeof(node));
        n->value[0] = v;
        stk->num++;
        n->next = stk->head;
        stk->head = n;
    }
    else {
        int i = stk->num % 10;
        stk->head->value[i] = v;
        stk->num++;
    }
}
stack* init_stack() {
    stack* s = malloc(sizeof(stack));
    s->num = 0;
    s->head = NULL;
    return s;
}
printstack(stack* s) {
    node* n = s->head;
    int num = s->num;
    while (num > 0) {
        printf("%d ", n->value[(num - 1) % 10]);
        num--;
        if (num % 10 == 0) n = n->next;
    }
    printf("\n");
}
int push_or_pop(int pushnum, int popnum, int max)
{
```

```
    // max 정도 쌓일 때까지는 push 가 주로 나옴.
    // 전에 push 였으면 push 가 가능성이 높고 pop 이었으면 pop 이 가능성이
높음.
    static double cont = 1.0;
    int j = rand();
    if (j % 10 + 1 > ((pushnum - popnum) * cont / max) * 10)
    {
        cont = 0.2;
        return 1;
    }
    else
    {
        cont = 5.0;
        return 0;
    }
}
int main()
{
    int i = 0;
    clock_t st1, st2, end1, end2;
    float t1, t2;
    int popNum = 0;
    int pushNum = 0;
    int v;
    st1 = clock();//시작시각
    stack* s = init_stack();
    while (i++ < MAX * 40)
    { /*100 만번씩 40 회 수행 */
        v = rand();
        if (push_or_pop(pushNum, popNum, MAX))
        {
            push(s, v);
            pushNum++;
        }
        else
        {
            pop(s);
            popNum++;
```

```
        }
        if (i % MAX == 0) printf("%d ", pushNum - popNum);
    }
    printf("\n\npush:%d, pop:%d\n\n", pushNum, popNum);
    end1 = clock();//종료시각
    init_stack_l();//stack 생성
    i = 0;
    popNum = 0;
    pushNum = 0;
    st2 = clock();//시작시각
    while (i++ < MAX * 40)
    { /*100 만번씩 40 회 수행 */
        v = rand();
        if (push_or_pop(pushNum, popNum, MAX))
        {
            push_l(v);
            pushNum++;
        }
        else
        {
            pop_l();
            popNum++;
        }
        if (i % MAX == 0) printf("%d ", pushNum - popNum);
    }
    printf("\n\npush:%d, pop:%d\n\n", pushNum, popNum);
    end2 = clock();//종료시각
    t1 = (float)(end1 - st1);
    t2 = (float)(end2 - st2);
    printf("\n하이브리드 스택:%lf, 단순연결리스트 스택:%lf\n\n", t1, t2);
}
```



2. (3)

10개	5개	20개
하이브리드 스택:9550.000000,	하이브리드 스택:10047.000000	하이브리드 스택:9070.000000

하이브리드스택보다 연결리스트가 많은 시간이 소요되는 이유는 push와 pop연산 시에 메모리의 할당과 반환이 이루어져야 하기 때문이다.
같은 이유로, 하나의 노드에 원소가 적게 저장 될 수록 메모리의 할당과 반환이 자주 일어나므로 시간이 많이 소요되게 된다.