2023 Algorithm Study Week 1

cheonwon · 8분 전 · 비공개

2023 Algorithm Study smarcle

자료구조 복습

큐, 이중원형연결리스트

각각 큐와 이중원형연결리스트를 이용해 풀었으며, 체감난이도는 풍선터트리기 코드가 좀 많이 어려웠다.

큐 <백준 10845번>

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef int element;

typedef struct QueueNode {
    element data;
    struct QueueNode* next;
}QueueNode;

typedef struct
{
    QueueNode* front;
    QueueNode* rear;
    int n;
}QueueType;

void init(QueueType* Q) {
    Q->front = Q->rear = NULL;
```

```
    Q->n = 0;
}
```

스택과 비교했을 때 큐의 가장 큰 특징은 선입 선출이며, 이를 가능하게 하는 것이 front와 rear의 존재이다(head와 tail이라고 하기도 함, 선언하기 나름인듯). 단일연결리스트로 구현하였으며, 큐의 세부 노드를 나타내는 큐노드 구조체, 큐의 구조를 나타낼 큐타입 구조체를 구현하고 이를 초기화하는 init함수를 구현했다.

```
int isEmpty(QueueType* Q) {
    return Q->front == NULL;
}

void push(QueueType* Q, element e)
{
    QueueNode* node = (QueueNode*)malloc(sizeof(QueueNode));
    node->data = e;
    node->next = NULL;

    if (isEmpty(Q))
        Q->front = Q->rear = node;
    else {
        Q->rear->next = node;
        Q->rear = node;
    }
    Q->n++;
}
```

init에서 front와 rear를 NULL로 초기화해줬기에 비어있으면 NULL이라는 조건을 통해 isEmpty를 구현한다. 이후 새로운 노드를 생성하고 이를 연결하는 push메소드를 구현한다. 단일연결이기에 코드가 간단하다.

```
element pop(QueueType* Q)
{
    if (isEmpty(Q)) {
        printf("-1\n");
        return 0;
    }
    QueueNode* p = Q->front;
    element e = p->data;
    Q->front = p->next;

    if (Q->front == NULL) //원소가 딱 한개밖에 없던 경우 삭제시 원소가 없는 NULL값이 되므로 초기상태(in
        Q->rear = NULL; //rear는 마지막 노드 그대로 가리키고 있는 상태가 되므로 끊어주기 위해

    free(p);
    printf("%d\n", e);
    Q->n--;
    return e;
}
```

큐의 경우 삽입은 rear(맨 끝)에서, 삭제는 front(맨 앞)에서 이뤄진다는 것을 고려하여 push와 pop을 구현한다. 이때, pop시 front만 움직이기에 rear는 삭제시에도 마지막 노드를 가리키는 상태 그대로 남아있을 수 있으므로, 둘 다 NULL로 바꿔줘야 한다고 한다.

```
element front(QueueType* Q)
{
    if (isEmpty(Q)) {
        printf("-1\n");
        return 0;
    }
    printf("%d\n", Q->front->data);
    return Q->front->data;
}

element back(QueueType* Q)
{
    if (isEmpty(Q)) {
        printf("-1\n");
        return 0;
    }
    printf("%d\n", Q->rear->data);
    return Q->rear->data;
}

element size(QueueType* Q)
{
    printf("%d\n", Q->n);
    return Q->rear->data;
}
```

맨 앞의 원소를 출력하는 front, 맨 뒤의 원소를 출력하는 back, 총 큐의 원소 갯수를 출력하는 size는 각각 큐 구조체를 구성하는 기본 요소들이니 별도의 코딩 없이 해당하는 값을 찾아서 return해준다.

단, 주의사항

이유는 모르겠는데 백준에 제출할 시 함수에서 return값으로 정수를 받아온뒤, main에서 출력하는 형식으로 하면 오류가 난다. 그래서 똑같은 역할을 하는 코드일지라도 백준에 제출할때는 함수 내부에서 그냥 직접 printf 하는 것을 추천한다. (이유는 잘 모르겠음.)

이후는 main코드이다.

```
int main()
{
    QueueType Q;
    init(&Q);

    int num;
    scanf("%d", &num);
```

```
        char sts[100];
        for (int i = 0; i < num; i++) {
            scanf("%s", &sts);
            getchar();
            if (strcmp(sts, "pop") == 0) {
                pop(&Q);
            }
            else if (strcmp(sts, "push") == 0) {
                int tmp;
                scanf("%d", &tmp);
                getchar();
                push(&Q, tmp);
            }
            else if (strcmp(sts, "front") == 0) {
                front(&Q);
            }
            else if (strcmp(sts, "back") == 0) {
                back(&Q);
            }
            else if (strcmp(sts, "size") == 0) {
                size(&Q);
            }
            else if (strcmp(sts, "empty") == 0) {
                printf("%d\n", isEmpty(&Q));
            }
        }

        return 0;
    }
```

출력 예시에 맞춰서 잘 출력되도록 작성해주기만 하면 끝이다.

<제출 화면>

제출 번호	아이디	문제	결과	메모리	시간	언어	코드 길이	제출한 시간
66500690	cheonwon	10845	맞았습니다!!	1380 KB	4 ms	C11 / 수정	2084 B	6일 전

솔직히 중간에 저 주의사항 하나 때문에 엄청 해맸다. 저거 하나만 고치니까 코드가 맞았다고 하는 것을 보고 솔직히 좀 허무했다.
명령 내부에 "~을 출력한다"라는 조건 자체가 들어가 있어서 그랬나 싶기도 하고... 왜 틀렸다고 하는지 원인은 아직까지 잘 모르겠다.

풍선 터뜨리기 <백준 2346번> - 풀이 1

```
#pragma warning(disable:4996)
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

typedef struct listnode {
    int n;
    int k;
    struct listnode* prev;
    struct listnode* next;
}listnode;

typedef struct {
    listnode* T;
}listtype;

listnode* node() {
    listnode* p = (listnode*)malloc(sizeof(listnode));
    p->next = p->prev = NULL;
    return p;
}

void init(listtype* L) {
    L->T = NULL;
}
```

이중연결리스트를 선언한다. 이때 이번에는 add가 뒤에서부터 들어가는 형태이므로 편의상 head가 아닌 tail을 사용했다. 원래는 head와 tail을 동시에 선언한 뒤, delete시 head와 tail의 위치를 옮겨가면서 head부터 몇번 이동하여 삭제하는 식으로 알고리즘을 작성했는데, head와 tail의 위치를 옮기는 것이 여의치 않은데다, 정상적으로 구현했다고 생각했음에도 불구하고 또 백준에서 오류가 자꾸 일어나서 tail만 사용하는 코드로 아예 새롭게 작성했다. 또, 리스트의 노드를 선언할 때에도 k가 하나가 더 들어가는데, 중간에 리스트의 노드가 삭제되어도 자기 자신의 번호(인덱스)를 유지할 수 있어야 하기 때문이다. n은 다음 노드로 이동할때, k는 출력 결과를 맞출때 사용한다고 보면 된다.

```
void add(listtype* L, int e, int k) {
    listnode* p = node();
    p->n = e;
    p->k = k;

    if (L->T == NULL) {
        L->T = p;
        L->T->next = L->T->prev = p;
    }
    else {
        p->prev = L->T;          //새로 삽입하는 노드를 먼저 연결
        p->next = L->T->next;
        L->T->next->prev = p;    //그 다음 원래 노드들 연결 변경(L->T->next = L->H역할)
        L->T->next = p;
        L->T = p;                //마지막 테일 변경
```

```
        }
    }
```

항상 head로 선언하다가 insertlast 메소드를 좀 더 쉽게 구현해 보겠다고 tail을 사용했는데, 익숙하지가 않아서 오히려 이부분이 생각보다 어려웠다. 특히 원형연결도 해주어야 하는데, 새로 만들어지는 노드의 prev를 L->T(현재 마지막 노드), next를 L->T->next(맨 처음 노드 - head역할)에 연결하고, 이후 L->T->next(맨 처음 노드 - head역할)를 연결한다는 발상이 쉽지 않았다. L->T->next가 head역할을 한다는 것을 기억하자.

```
int delete(listnode* d, int arr[], int n) { //처음엔 헤더와 테일을 이동시키는 방식으로 구현
    //but 백준 오류로 인해 삭제할 노드 자체의 위치를 움직이도록 함.
    if (d->next == d) {
        //d == NULL(노드가 없을때를 뜻함)을 인식하질 못함. visual의 특징인듯
        //그래서 노드가 하나만 있을때(노드의 next, prev가 자기자신을 가리키고 있을때)를 조건으로 사용, 예
        printf("%d", d->k);
        free(d);
        return 0;           //그래서 다른 조건 사용
    }
    else {
        d->next->prev = d->prev;        //d(삭제할 노드)의 연결을 끊음
        d->prev->next = d->next;
        int e = d->n;
        printf("%d ", d->k);

        listnode* p = d;
        if (e > 0) {
            for (int i = 0; i < e; i++) {
                p = p->next;
            }
        }
        else {
            e = 0 - e; //절댓값
            for (int i = 0; i < e; i++) {
                p = p->prev;
            }
        }

        free(d);

        return delete(p, arr, n);
    }
}
```

가장 어려웠던 delete이다. 원래는 헤더와 테일을 움직여서 헤더를 삭제의 기준점으로 잡고 헤더에서부터 순위를 이동했는데, 오류로 인해서 다른 방식을 고안했다. 이번에는 삭제할 노드 자체를 주고, 해당 노드를 삭제한 뒤 그 위치를 기준으로 아예 다음 위치로 이동한 뒤 그 삭제할 노드를 다시 집어넣는 방식으로 노드 간에 연결을 통해 구현했다.
재귀문을 사용하였으며, 종료 조건은 노드가 하나도 없을때로 구현하려 했으나 d==NULL을

인식하질 못해서 d->next == d(노드의 next, prev가 자기자신을 가리키고 있을때 - 원형연결리스트이므로 이건 노드가 자기자신밖에 없음을 뜻함)을 종료조건으로 해서 구현했다.
알고리즘 순서는 다음과 같다. 처음에 노드가 들어오면 그 삭제할 위치의 노드의 연결을 끊고, 삭제할 노드의 k를 출력한다. 이후 다음 위치로 이동하는 과정이 이어지는데, 이때 내부에 e값이 양수면 그만큼 오른쪽(next)로 이동하고, e값이 음수면 그만큼 왼쪽(prev)로 이동해야 한다.
음수일때는 또 for문의 조건을 만족하기 위해서 e값에 절댓값 (0-e)를 씌워서 양수로 만든 뒤 그 횟수만큼 움직여야 하며, 이 과정을 통해 삭제할 다음 위치의 노드를 찾아낸다.
이후 노드를 free해주고, 다음 위치를 찾았으면 그 노드를 넣어서 똑같은 과정을 반복하여 끝까지 삭제한다.

arr은 배열인데, 수정전의 코드를 수정하는 과정에서 미처 지우지 못한 잔재일 뿐이다. 딱히 코드상해서 하는 역할이 없으므로 지워도 무관하다.(main에서도 마찬가지)
원래는 arr이라는 리스트에 큐와 동일하게 값을 저장해두고, for문을 돌려서 delete할때의 값이 리스트에서 어디에 위치하는지 인덱스를 뽑아서 i + 1을 출력하는 형식으로 했었는데, 이렇게 될 경우 리스트의 값이 중복되면 인덱스가 정확하게 출력되지 않는 경우가 생겨서 k를 별도로 사용하는 방법으로 구현하게 되었다.

```
입력:
4
2 2 2 2
출력:
정답 - 1 3 2 4
출력 - 4 4 4 4 //이렇게 나오게 된다는 소리
```

우리는 풍선의 번호를 출력해야 한다는 것에 유의해야 한다.

```
int main() {
    listtype L;
    init(&L);

    int n;
    scanf("%d", &n);
    int arr[1000];

    for (int i = 0; i < n; i++) {
        int tmp;
        scanf("%d", &tmp);
        arr[i] = tmp;
        add(&L, tmp, i + 1);
    }

    delete(L.T->next, arr, n);

    return 0;
}
```

delete까지 구현하고 나면 딱히 어려운게 없다. 여기서는 함수 내부가 아니므로 L->T가 아니라 L.T로 연결되기에(L이 포인터 형식이 아니므로 - 함수는 포인터로 들어와서 포인터를 연결하는 L->T를, main에서는 L이 포인터가 아닌 그냥 구조체이므로 L.T로 연결) L.T->next로 넣어줘야 한다는 점을 제외하곤 특이사항이 없다.

<제출 화면>

풍선 터뜨리기 <백준 2346번> - 풀이 2

앞서서 head와 tail을 동시에 구현하고, 노드에서 바로 다음 노드를 찾아서 삭제하는 대신 헤드에서 위치만큼 움직여서 삭제하는 방법으로 구현을 시도했었다고 했는데, 백준오류라 생각했는데 k를 활용하지 않았던 것이 문제였던것 같다. 연결리스트에서 key값을 여러개를 활용할 수 있다는 점을 꼭 기억해 두자.

```
#pragma warning(disable:4996)
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

typedef struct listnode {
    int num;
    int k;
    struct listnode* prev;
    struct listnode* next;
}listnode;

typedef struct {
    listnode* H;
    listnode* T;
}listtype;

listnode* node() {
    listnode* node = (listnode*)malloc(sizeof(listnode));
    node->next = NULL;
    return node;
}
```

head와 tail을 동시에 사용하는 모습을 볼 수 있다.

```
void add(listtype* L, int num, int r, int k) {
    listnode* p = node();
    p->num = num;
    p->k = k;

    if (r == 1) {
        L->H = p;
        L->T = p;
        p->next = p; //자기자신
        p->prev = p; //자기자신
    }
    else {
        p->prev = L->T;
        p->next = L->H;
        L->T->next = p;
        L->H->prev = p;
        L->T = p;
    }
}
```

노드가 하나도 없을 때는 헤드와 테일이 모두 첫 노드를 가리키도록 하고, 노드의 prev, next가 모두 자기 자신을 가리키도록 조정한다. 이후 과정은 동일하다.(L->T->next가 L->H로 변해서 좀 더 직관적으로 이해하기 편할 뿐, 동일한 과정이다.)

```
int answer(listtype* L, int r, int n) {
    int e;
    int k;
    if (r > 0) {
        listnode* p = L->H;
        for (int i = 1; i < r; i++) {
            p = p->next;
        }
        e = p->num;
        k = p->k;
        p->prev->next = p->next;
        p->next->prev = p->prev;
        L->H = p->next; //H와 T가 가리키는 노드가 사라질 경우 옮겨줘야 함.
        L->T = p->prev; //이때 H와 T를 이렇게 변경해주면 바로바로 다음 delete변경 가능(문제 조건)
        free(p);
    }
    else {
        r = 0 - r; //절댓값으로 바꿔주기
        listnode* p = L->T;
        for (int i = 1; i < r; i++) {
            p = p->prev;
        }
        e = p->num;
        k = p->k;
        p->prev->next = p->next;
        p->next->prev = p->prev;
        L->H = p->next; //H와 T가 가리키는 노드가 사라질 경우 옮겨줘야 함.
        L->T = p->prev; //이때 H와 T를 이렇게 변경해주면 바로바로 다음 delete변경 가능(문제 조건)
        free(p);
    }
```

```
        printf("%d ", k);
        if (n == 1)
            return 0;
        return answer(L, e, n - 1);
    }
```

가장 어려운 부분인데, 여기서는 아까전과 달리 출발 위치가 달라 if문 내부에 코드를 따로따로 작성해주어야 한다.
인덱스의 정의에 가장 충실한 코드로, 삭제할 위치를 찾기 위해 e가 양수면 앞에서부터(head) 시작해서 오른쪽(next)으로, e가 음수면 뒤에서부터(tail) 시작해서 왼쪽(prev)으로 이동한다.
이동을 완료한 후 노드를 삭제할 위치를 찾으면 노드의 e와 k를 저장한 뒤 그 노드의 연결을 끊고 free한다. 이때, L->H는 삭제한 노드의 다음 노드를 가리키도록 옮겨주고, L->T는 삭제한 노드의 바로 전 노드로 옮겨준다. 이렇게 되면 삭제한 노드의 다음부터(L->H부터 L->T까지) 리스트가 새롭게 생성되며, 다음에 이 과정을 반복할때도 똑같이 e를 찾아서 이동후 삭제할 수 있다.

앞의 버전이 e번만큼 움직여서 다음에 삭제할 노드를 직접 찾았다면, 이 버전은 head와 tail을 움직여 보다 복잡하지만, 직관적으로 움직이도록 작성했다. 노드를 기준으로 상대위치로 삭제할 것인지, 아니면 연결리스트 전체를 기준으로 절대위치로 삭제할 것인지의 차이인것 같다.(-정확한 비유는 아니지만 앞의것은 상대위치, 뒤의것은 절대위치라 보면 될 것 같다.)
이후 k값을 출력하고, 이 과정을 반복하면 된다.

```
int main()
{
    listtype L;

    int tmp[1001];
    int n;

    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        int k;
        scanf("%d", &k);
        add(&L, k, i + 1, i + 1);
        tmp[i] = k;
    }

    answer(&L, 1, n);

    return 0;
}
```

실행하면 똑같은 결과가 나오며, 백준에서도 둘 다 똑같이 맞았다고 결과가 나온다.

<제출 화면>

2346 | cheonwon | 모든 언어 | 모든 결과 | 검색

제출 번호	아이디	문제	결과	메모리	시간	언어	코드 길이	제출한 시간
66819774	cheonwon	2346	맞았습니다!!	1116 KB	0 ms	C99 / 수정	1832 B	18분 전
66815750	cheonwon	2346	맞았습니다!!	1116 KB	0 ms	C99 / 수정	2093 B	1시간 전

근데 둘다 해본 입장에서 두 방법이 모두 가능하다면 첫번째 코드가 더 간편한 것 같다.

결론 - 연결리스트를 다루는 함수라고 해서 무조건 연결리스트를 입력값으로 집어넣어야 하는 것은 아니다. 각 노드를 입력값으로 넣는 게 더 편리할때도 있다.(특히 재귀문 할때 조심)

앞의 코드는 입력값을 노드로, 뒤의 코드는 입력값을 리스트로 했을 뿐이다. 결론적으로 정리해보면 그게 가장 큰 차이였던것 같다.

천승원

뭐든지 한걸음씩

이전 포스트
2023 Algorithm Study Week 2 - 과제

0개의 댓글

댓글을 작성하세요

댓글 작성


Powered by
Stellate