『제 24기 청학동』Tutoring 4주 주간 학습 보고서

학습활동 참여 점검			
팀명	2+3=5		
1회차 일시	2022.04.12 18:00 ~ 21:00	1회차 장소	Discord
1회차 참석자	신수빈,변장무,이준호,이동현,권재현		
1회차 불참자			
2회차 일시		2회차 장소	
2회차 참석자			
2회차 불참자			
3회차 일시		3회차 장소	
3회차 참석자			
3회차 불참자			
4회차 일시		4회차 장소	
4회차 참석자			
4회차 불참자			
5회차 일시		5회차 장소	
5회차 참석자			
5회차 불참자			
학습목표	리스트 - 1) 배열 리스트 // 연결리스트 - 1) 단순 연결 리스트, 2) 원형 연결 리스트, 3) 이중 연결 리스트, 4) 연결 리스트로 구현한 스택, 5) 연결 리스트로 구현한 큐		
과제물	4주차 과제 없음		

『제 24기 청학동』Tutoring 4주 주간 학습 보고서

학습활동 참여 점검	
팀명	2+3=5

리스트

리스트에는 항목들이 차례대로 저장되어 있다.

리스트의 항목들은 순서 또는 위치를 가진다.

각 항목 간에 순서의 개념이 없는 집합과는 별개의 개념이다.

L = (item, item, item, ..., item)

리스트 ADT

insert(list, pos, item) : pos 위치에 요소를 추가한다.
insert_last(list, item) : 맨 끝에 요소를 추가한다.
insert_first(list, item) : 맨 처음에 요소를 추가한다.
delete(list, pos) : pos 위치의 요소를 제거한다.
clear(list) : 리스트의 모든 요소를 제거한다.
replace(list, pos, item) : pos 위치의 값을 요소로 교체
is_in_list(list, item) : 리스트 내에 아이템이 존재하는지 검사(T/F)
get_entry(list, pos) : pos 위치의 요소를 반환한다.
get_length(list) : 리스트의 길이를 구한다.
is_empty(list) : 리스트가 비었는지를 검사한다.
is_full(list) : 리스트가 꽉 찼는지를 검사한다.
print_list(list) : 리스트의 모든 요소를 표시한다.

1) 배열로 구현된 리스트

배열을 이용하여 리스트를 구현하면 순차적인 메모리 공간이 할당된다.
이것을 리스트의 순차적 표현(Sequential Representation)이라고 한다.

장점
구현이 간단하고 속도가 빠르다.

단점
삽입, 삭제 시 나머지 데이터를 밀어내거나 당겨서 재저장하는 오버헤드가 발생한다.
리스트의 크기가 고정된다.

배열 리스트 프로그램

```
#include
#include

#define MAX_LIST_SIZE 100

typedefintelement;
typedefstruct
{
    element array[MAX_LIST_SIZE];
    int size;
} ArrayListType;

void error(char *message)
```

```
{
    fprintf(stderr, "%s₩n", message);
    exit(1);
}

voidinit(ArrayListType *L)
{
    L -> size = 0;
}

intis_empty(ArrayListType *L)
{
    return L -> size == 0;
}

intis_full(ArrayListType *L)
{
    return L -> size == MAX_LIST_SIZE;
}

elementget_entry(ArrayListType *L, int pos)
{
    if(pos < 0 || pos >= L -> size)
        error("위치 오류");
    return L -> array[pos];
}

voidprint_list(ArrayListType *L)
{
    int i;
    for(i=0;isize;i++)
        printf("%d->", L -> array[i]);
    printf("₩n");
}

voidinsert_last(ArrayListType *L, element item)
{
    if(L -> size >= MAX_LIST_SIZE)
    {
        error("리스트 오버플로우");
    }
    L -> array[L->size++] = item;
}

voidinsert(ArrayListType *L, int pos, element item)
{
    int i;

    if(!is_full(L) && (pos >= 0) && (pos <= L -> size))
    {
        for(i=(L->size-1);i>=pos;i--)
            L -> array[i+1] = L -> array[i];
        L -> array[pos] = item;
        L -> size++;
    }
}

elementdelete(ArrayListType *L, int pos)
{
    element item;
    int i;

    if(pos < 0 || pos >= L -> size)
        error("위치 오류");
    item = L -> array[pos];
```

```
    for(i=pos;i<(L->size-1);i++)
        L -> array[i] = L -> array[i+1];
    L -> size--;

    return item;
}

intmain(void)
{
    ArrayListType list;

    init(&list);
    insert(&list, 0, 10);
    print_list(&list);
    insert(&list, 0, 20);
    print_list(&list);
    insert(&list, 0, 30);
    print_list(&list);
    insert_last(&list, 40);
    print_list(&list);
    delete(&list, 0);
    print_list(&list);

    return 0;
}
```

연결리스트

하나의 프로그램 안에는 동시에 여러 개의 연결 리스트가 존재할 수 있다.

장점

삭제나 삽입 시에 앞뒤에 있는 데이터들을 이동할 필요가 없이 줄만 변경시켜주면 된다.
리스트의 크기가 동적으로 변할 수 있다.

단점

(배열에 비해) 상대적으로 구현이 어렵고 오류가 나기 쉽다.
데이터뿐만 아니라 포인터도 저장하여야 하므로 메모리 공간을 많이 사용한다.

1) 단순 연결 리스트(Singly Linked List)

단순 연결 리스트는 하나의 방향으로만 연결되어 있는 연결 리스트이다.
체인(Chain)이라고도 한다.
마지막 노드의 링크는 NULL값을 가진다.

노드의 정의

노드는 자기 참조 구조체를 이용하여 정의된다.
자기참조 구조체란 자기 자신을 참조하는 포인터를 포함하는 구조체이다.
구조체 안에는 데이터를 저장하는 data 필드와 포인터가 저장되어 있는 link 필드가 존재한다.
data 필드는 element 타입의 데이터를 저장하고 있다.
link 필드는 ListNode를 가리키는 포인터로 정의되며 다음 노드의 주소가 저장된다.
단순 연결 리스트는 헤드 포인터만 있으면 모든 노드를 찾을 수 있다.

노드의 생성

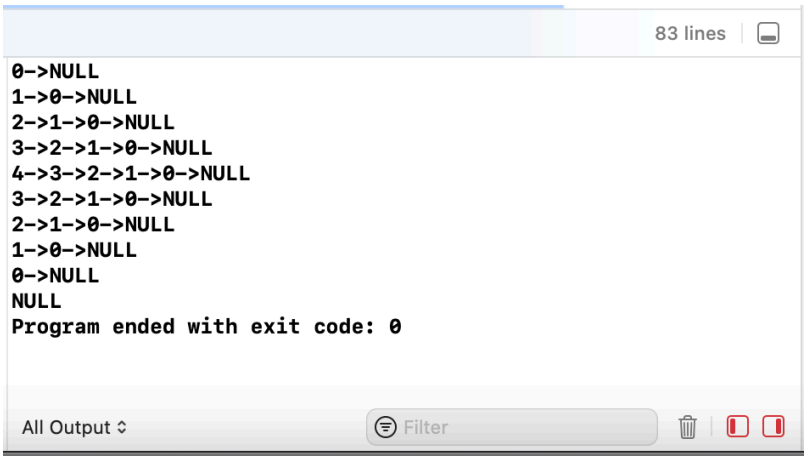
연결 리스트에서는 필요할 때마다 동적 메모리 할당을 이용하여 노드를 동적으로 생성한다.
malloc() 함수를 이용하여 노드의 크기만큼의 동적 메모리를 할당 받는다.
이 동적 메모리가 하나의 노드가 된다.
동적 메모리의 주소를 헤드 포인터에 저장한다.

단순 연결 리스트 프로그램

```
#include

#include


typedefintelement;

typedefstructListNode

{

    element data;

    struct ListNode *link;

} ListNode;


void error(char *message)

{

    fprintf(stderr, "%s₩n", message);

    exit(1);

}


ListNode* insert_first(ListNode *head, int value)

{

    ListNode *p = (ListNode *) malloc(sizeof(ListNode));

    p -> data = value;

    p -> link = head;

    head = p;


    return head;

}


ListNode* insert(ListNode *head, ListNode *pre, element value)

{
```

```
    ListNode *p = (ListNode *)malloc(sizeof(ListNode));
    p -> data = value;
    p -> link = pre -> link;
    pre -> link = p;

    return head;
}

ListNode* delete_first(ListNode *head)
{
    ListNode *removed;
    if(head == NULL) return NULL;
    removed = head;
    head = removed -> link;
    free(removed);

    return head;
}

ListNode* delete(ListNode *head, ListNode *pre)
{
    ListNode *removed;
    removed = pre -> link;
    pre -> link = removed -> link;
    free(removed);

    return head;
}

voidprint_list(ListNode* head)
{
   for (ListNode* p = head; p != NULL; p = p->link)
       printf("%d->", p->data);
   printf("NULL ₩n");
}

intmain(void)
{
```

```
    ListNode *head = NULL;
    int i;

    for(i=0;i<5;i++)
    {
        head = insert_first(head, i);
        print_list(head);
    }

    for(i=0;i<5;i++)
    {
        head = delete_first(head);
        print_list(head);
    }

    return 0;
}
```



단순 연결 리스트로 구현한 다항식 덧셈 프로그램

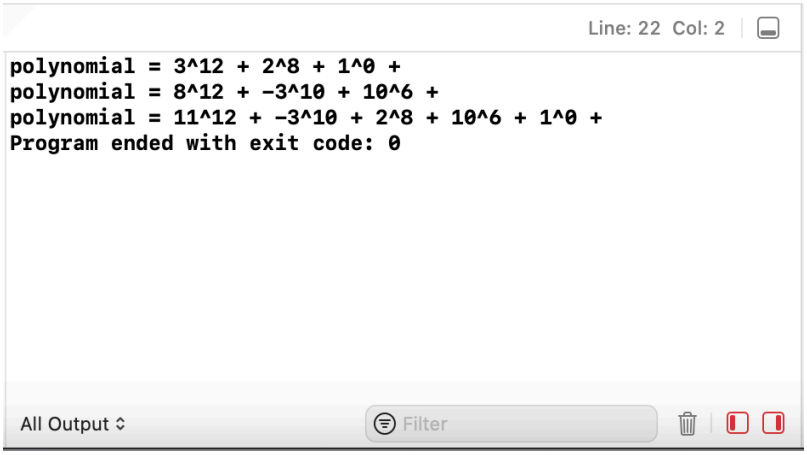
```
#include
#include

typedefstructListNode
{
    int coef;
    int expon;
    struct ListNode *link;
} ListNode;
```

```
typedefstructListType
{
    int size;
    ListNode *head;
    ListNode *tail;
} ListType;

void error(char *message)
{
    fprintf(stderr, "%s₩n", message);
    exit(1);
}

ListType* create()
{
    ListType *plist = (ListType *)malloc(sizeof(ListType));
    plist->size = 0;
    plist->head = plist->tail = NULL;

    return plist;
}

void insert_last(ListType* plist, int coef, int expon)
{
    ListNode* tmp = (ListNode *)malloc(sizeof(ListNode));
    if(tmp == NULL) error("메모리 할당 에러");
    tmp -> coef = coef;
    tmp -> expon = expon;
    tmp -> link = NULL;

    if(plist->tail == NULL)
        plist->head = plist->tail = tmp;

    else
    {
        plist->tail->link = tmp;
        plist->tail = tmp;
```

```
    }
    plist->size++;
}

void poly_add(ListType* plist1, ListType* plist2, ListType* plist3)
{
    ListNode* a = plist1->head;
    ListNode* b = plist2->head;
    int sum;

    while(a && b)
    {
        if(a->expon == b->expon)
        {
            sum = a->coef + b->coef;
            if(sum != 0) insert_last(plist3, sum, a->expon);
            a = a->link;
            b = b->link;
        }

        else if (a->expon > b->expon)
        {
            insert_last(plist3, a->coef, a->expon);
            a = a->link;
        }

        else
        {
            insert_last(plist3, b->coef, b->expon);
            b = b->link;
        }
    }

    for(;a!=NULL;a=a->link)
        insert_last(plist3, a->coef, a->expon);
    for(;b!=NULL;b=b->link)
        insert_last(plist3, b->coef, b->expon);
```

```
}

void print_poly(ListType* plist)
{
    ListNode* p = plist->head;

    printf("polynomial = ");
    for(;p;p=p->link)
        printf("%d^%d + ", p->coef, p->expon);
    printf("₩n");
}

intmain(void)
{
    ListType *list1, *list2, *list3;

    list1 = create();
    list2 = create();
    list3 = create();

    insert_last(list1, 3, 12);
    insert_last(list1, 2, 8);
    insert_last(list1, 1, 0);

    insert_last(list2, 8, 12);
    insert_last(list2, -3, 10);
    insert_last(list2, 10, 6);

    print_poly(list1);
    print_poly(list2);

    poly_add(list1, list2, list3);
    print_poly(list3);

    free(list1);
    free(list2);
    free(list3);
```

```
}
```



2) 원형 연결 리스트(Circular Linked List)

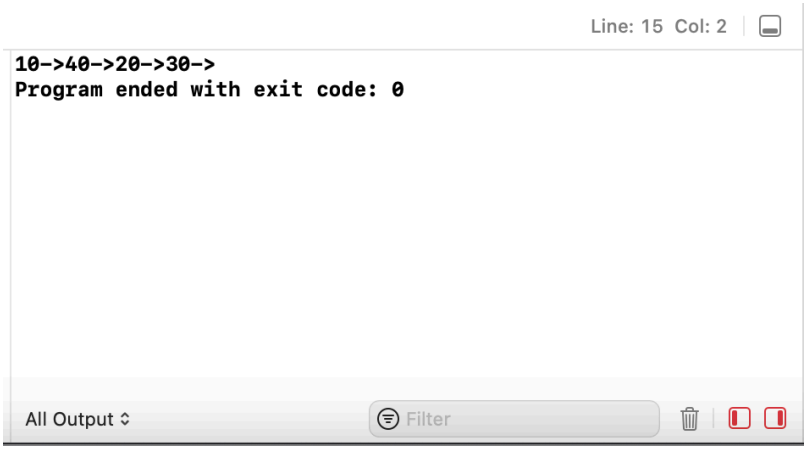
원형 연결 리스트는 마지막 노드의 링크가 첫 번째 노드를 가리키는 리스트이다.

원형 연결 리스트 프로그램

```
#include

#include


typedefintelement;

typedefstructListNode

{

    element data;

    struct ListNode *link;

} ListNode;


void error(char *message)

{

    fprintf(stderr, "%s₩n", message);

    exit(1);

}


ListNode* insert_first(ListNode *head, element data)

{

    ListNode *node = (ListNode *) malloc(sizeof(ListNode));

    node -> data = data;

    

    if(head == NULL)

    {

        head = node;
```

```
        node -> link = head;
    }

    else
    {
        node -> link = head -> link;
        head -> link = node;
    }
    return head;
}

ListNode* insert_last(ListNode *head, element data)
{
    ListNode *node = (ListNode *)malloc(sizeof(ListNode));
    node -> data = data;
    if(head == NULL)
    {
        head = node;
        node -> link = head;
    }

    else
    {
        node -> link = head -> link;
        head -> link = node;
        head = node;
    }

    return head;
}

voidprint_list(ListNode *head)
{
    ListNode* p;

    if(head == NULL) return;
    p = head -> link;
```

```
    do
    {
        printf("%d->", p -> data);
        p = p -> link;
    } while (p != head);
    printf("%d-> ₩n", p -> data);
}

intmain(void)
{
    ListNode *head = NULL;

    head = insert_last(head, 20);
    head = insert_last(head, 30);
    head = insert_first(head, 40);
    head = insert_first(head, 10);

    print_list(head);

    return 0;
}
```



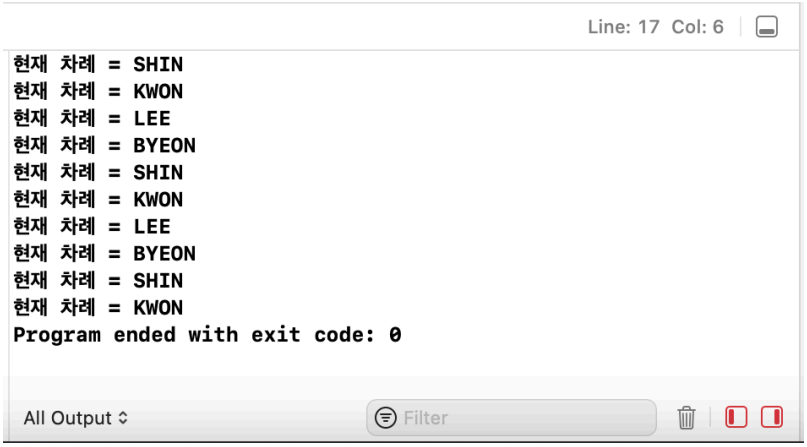
원형 연결 리스트로 구현한 게임 순서 안내 프로그램

```
#include
#include
#include

typedefcharelement[100];
typedefstructListNode
{
```

학습 내용

```
    element data;
    struct ListNode *link;
} ListNode;

ListNode* insert_first(ListNode* head, element data)
{
    ListNode *node = (ListNode *)malloc(sizeof(ListNode));
    strcpy(node->data, data);
    if(head == NULL)
    {
        head = node;
        node -> link = head;
    }

    else
    {
        node -> link = head -> link;
        head -> link = node;
    }

    return head;
}

intmain(void)
{
    ListNode *head = NULL;
    int i;

    head = insert_first(head, "SHIN");
    head = insert_first(head, "BYEON");
    head = insert_first(head, "LEE");
    head = insert_first(head, "KWON");

    ListNode* p = head;

    for(i=0;i<10;i++)
    {
```

```
        printf("현재 차례 = %s₩n", p -> data);
        p = p -> link;
    }

    return 0;
}
```



3) 이중 연결 리스트(Doubly Linked List)

이중 연결 리스트는 하나의 노드가 선행 노드와 후속 노드에 대한 두 개의 링크를 가지는 리스트이다.

장점

링크가 양방향이므로 검색이 가능해진다.

단점

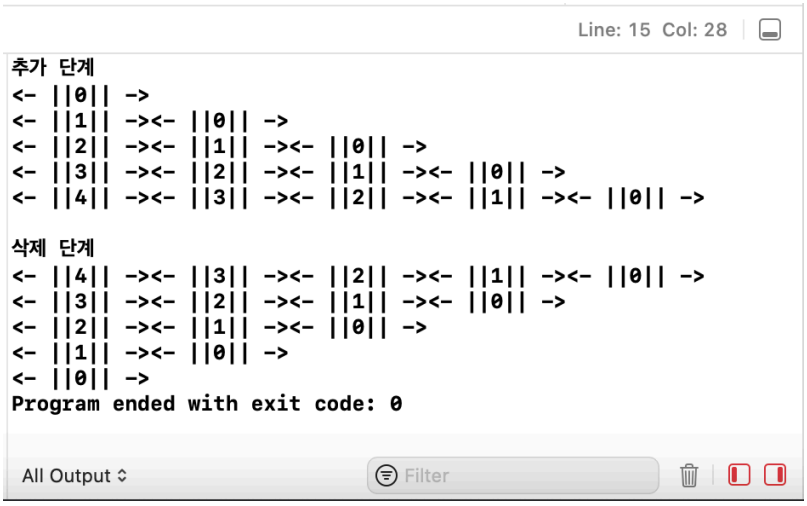
공간을 많이 차지하고 코드가 복잡해진다.

이중 연결 리스트 프로그램

```
#include
#include

typedefintelement;
typedefstructDListNode
{
    element data;
    struct DListNode* llink;
    struct DListNode* rlink;
} DListNode;

void init(DListNode* phead)
{
    phead -> llink = phead;
    phead -> rlink = phead;
```

```
}

voidprint_dlist(DListNode* phead)
{
    DListNode* p;
    for(p=phead->rlink;p!=phead;p=p->rlink)
        printf("", p->data);

    printf("₩n");
}

void dinsert(DListNode *before, element data)
{
    DListNode *newnode = (DListNode *)malloc(sizeof(DListNode));

    newnode -> data = data;
    newnode -> llink = before;
    newnode -> rlink = before -> rlink;
    before -> rlink -> llink = newnode;
    before -> rlink = newnode;
}

void ddelete(DListNode* head, DListNode* removed)
{
    if(removed == head) return;
    removed -> llink -> rlink = removed -> rlink;
    removed -> rlink -> llink = removed -> llink;
    free(removed);
}

intmain(void)
{
    int i;
    DListNode* head = (DListNode *)malloc(sizeof(DListNode));
    init(head);

    printf("추가 단계₩n");
    for(i=0;i<5;i++)
```

```
    {
        dinsert(head, i);
        print_dlist(head);
    }

    printf("₩n삭제 단계₩n");
    for(i=0;i<5;i++)
    {
        print_dlist(head);
        ddelete(head, head -> rlink);
    }

    free(head);

    return 0;
}
```



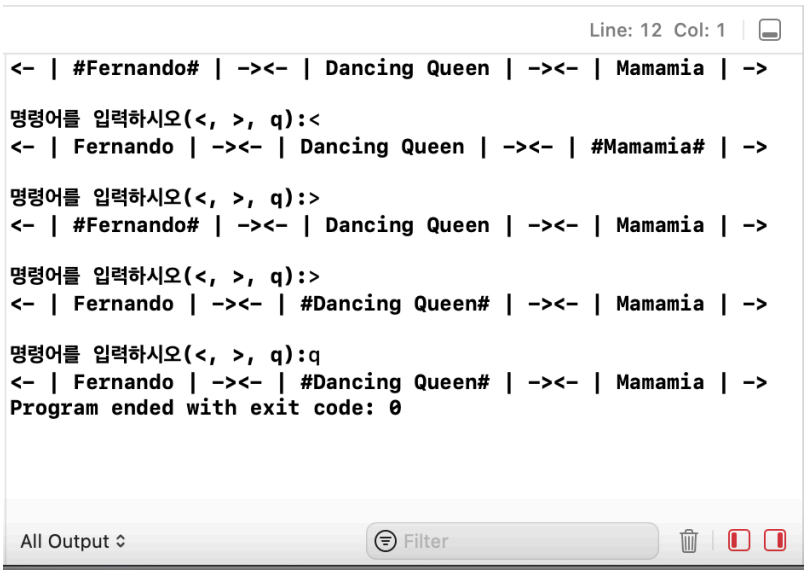
이중 연결 리스트로 구현한 mp3 플레이어 프로그램

```
#include
#include
#include

typedefcharelement[100];
typedefstructDListNode
{
    element data;
    struct DListNode* llink;
    struct DListNode* rlink;
```

```
} DListNode;

DListNode* current;

void init(DListNode* phead)
{
    phead -> llink = phead;
    phead -> rlink = phead;
}

voidprint_dlist(DListNode* phead)
{
    DListNode* p;
    for(p=phead->rlink;p!=phead;p=p->rlink)
    {
        if(p == current)
            printf("", p->data);
        else
            printf("", p->data);
    }

    printf("₩n");
}

void dinsert(DListNode *before, element data)
{
    DListNode *newnode = (DListNode *)malloc(sizeof(DListNode));

    strcpy(newnode -> data, data);
    newnode -> llink = before;
    newnode -> rlink = before -> rlink;
    before -> rlink -> llink = newnode;
    before -> rlink = newnode;
}

void ddelete(DListNode* head, DListNode* removed)
{
    if(removed == head) return;
```

```
    removed -> llink -> rlink = removed -> rlink;
    removed -> rlink -> llink = removed -> llink;
    free(removed);
}

intmain(void)
{
    char ch;
    DListNode* head = (DListNode *)malloc(sizeof(DListNode));
    init(head);

    dinsert(head, "Mamamia");
    dinsert(head, "Dancing Queen");
    dinsert(head, "Fernando");

    current = head -> rlink;
    print_dlist(head);

    do
    {
        printf("₩n명령어를 입력하시오(<, >, q):");
        ch = getchar();
        if(ch == '<')
        {
            current = current -> llink;
            if(current == head)
                current = current -> llink;
        }

        else if(ch == '>')
        {
            current = current -> rlink;
            if(current == head)
                current = current -> rlink;
        }
        print_dlist(head);
        getchar();
```

```
    } while (ch != 'q');

}
```



4) 연결 리스트로 구현한 스택

장점

연결 리스트를 이용하여 스택을 만들게 되면 크기가 제한되지 않는다.
동적 메모리 할당만 할 수 있으면 스택에 새로운 요소를 삽입할 수 있다.

단점

동적 메모리 할당이나 해제를 해야 하므로 삽입이나 삭제 시간이 좀 더 걸린다.

연결 리스트로 구현한 스택 프로그램

```
#include

#include


typedefintelement;

typedefstructStackNode

{

    element data;

    struct StackNode *link;

} StackNode;


typedefstruct

{

    StackNode *top;

} LinkedStackType;


voidinit(LinkedStackType *s)

{

    s -> top = NULL;
```

```
}

intis_empty(LinkedStackType *s)
{
    return (s -> top == NULL);
}

intis_full(LinkedStackType *s)
{
    return 0;
}

voidpush(LinkedStackType *s, element item)
{
    StackNode *tmp = (StackNode *)malloc(sizeof(StackNode));
    tmp -> data = item;
    tmp -> link = s -> top;
    s -> top = tmp;
}

voidprint_stack(LinkedStackType *s)
{
    for(StackNode *p=s->top;p!=NULL;p=p->link)
        printf("%d -> ", p->data);
    printf("NULL ₩n");
}

elementpop(LinkedStackType *s)
{
    if(is_empty(s))
    {
        fprintf(stderr, "스택이 비어있음 ₩n");
        exit(1);
    }

    else
    {
        StackNode *tmp = s -> top;
```

```
        int data = tmp -> data;
        s -> top = s -> top -> link;
        free(tmp);

        return data;
    }
}

elementpeek(LinkedStackType *s)
{
    if(is_empty(s))
    {
        fprintf(stderr, "스택이 비어있음₩n");
        exit(1);
    }

    else
    {
        return s -> top -> data;
    }
}

intmain(void)
{
    LinkedStackType s;
    init(&s);

    push(&s, 1);
    print_stack(&s);
    push(&s, 2);
    print_stack(&s);
    push(&s, 3);
    print_stack(&s);
    pop(&s);
    print_stack(&s);
    pop(&s);
    print_stack(&s);
```

```
    pop(&s);
    print_stack(&s);

    return 0;
}
```

```
98 lines
1 -> NULL
2 -> 1 -> NULL
3 -> 2 -> 1 -> NULL
2 -> 1 -> NULL
1 -> NULL
NULL
Program ended with exit code: 0
All Output
Filter
```

5) 연결 리스트로 구현한 큐

장점

크기가 제한되지 않는다.

단점

코드가 약간 더 복잡해지고, 링크 필드 때문에 메모리 공간을 더 많이 사용한다.

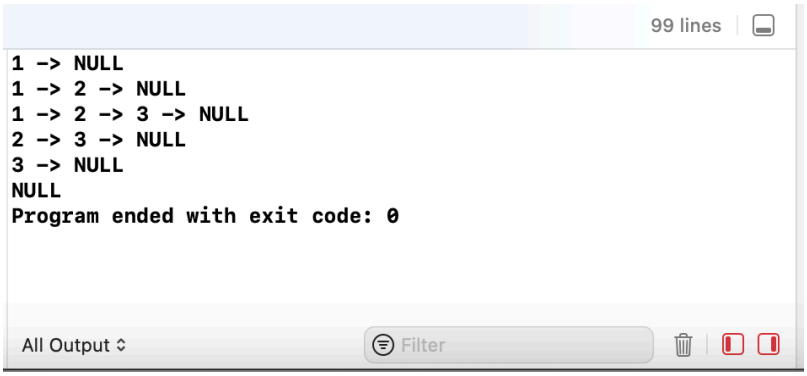
연결 리스트로 구현한 큐

```
#include
#include

typedefintelement;
typedefstructQueueNode
{
    element data;
    struct QueueNode *link;
} QueueNode;

typedefstruct
{
    QueueNode *front, *rear;
} LinkedQueueType;

voidinit(LinkedQueueType *q)
{
    q -> front = q -> rear = 0;
}
```

```
intis_empty(LinkedQueueType *q)
{
    return (q -> front == NULL);
}

intis_full(LinkedQueueType *q)
{
    return 0;
}

voidenqueue(LinkedQueueType *q, element data)
{
    QueueNode *tmp = (QueueNode *)malloc(sizeof(QueueNode));
    tmp -> data = data;
    tmp -> link = NULL;

    if(is_empty(q))
    {
        q -> front = tmp;
        q ->rear = tmp;
    }

    else
    {
        q -> rear -> link = tmp;
        q -> rear = tmp;
    }
}

elementdequeue(LinkedQueueType *q)
{
    QueueNode *tmp = q -> front;
    element data;

    if(is_empty(q))
    {
        fprintf(stderr, "스택이 비어있음₩n");
```

```
        exit(1);
    }

    else
    {
        data = tmp -> data;
        q -> front = q -> front -> link;
        if(q -> front == NULL)
            q -> rear = NULL;
        free(tmp);
        return data;
    }
}

voidprint_queue(LinkedQueueType *q)
{
    QueueNode *p;
    for(p=q->front;p!=NULL;p=p->link)
        printf("%d -> ", p->data);
    printf("NULL ₩n");
}

intmain(void)
{
    LinkedQueueType queue;
    init(&queue);

    enqueue(&queue, 1);
    print_queue(&queue);
    enqueue(&queue, 2);
    print_queue(&queue);
    enqueue(&queue, 3);
    print_queue(&queue);
    dequeue(&queue);
    print_queue(&queue);
    dequeue(&queue);
    print_queue(&queue);
```

```
    dequeue(&queue);
    print_queue(&queue);

    return 0;
}
```



『제 24기 청학동』Tutoring 4주 주간 학습 보고서

학습활동 참여 점검	
팀명	2+3=5
성찰활동 잘한 점	신수빈 : 학습한 내용을 바탕으로 응용할 수 있도록 노력했다.변장무 : 매주 청학동을 진행하면서 늦지 않고 항상 먼저 준비해서 지난 시간에 한 프로젝트와 공부한 내용을 복습하면서 대기하였다. 또한 코딩을 하면서 최대한 스스로 해결하려고 노력을 들였다.이동현 : 지난 주차 때 아쉬웠던 코드 리뷰 시간을 늘려 지난 문제점을 보완할 수 있었다.이준호 : 어려운 문제를 해결할 때 서로의 코드에서 오류가 나는 부분을 도와주었다.권재현 : 4주간 꾸준히 출석했다.
성찰활동 개선할 점	신수빈 : 과목 특성상 점점 내용이 심화되어 튜티들이 버거워하는데 어떻게 하면 쉽게 이해시키고 포기하지 않도록 격려할 수 있을까 고민해 봐야겠다.변장무 : 프로젝트에서 스스로 코딩을 하려고 노력을 함에 있어서 막히는 부분이 길어지면 집중력이 끊기는 부분이 중간중간 있었다. 집중력을 끊기게 하지 않도록 노력하여야 하며 막히는 부분이 길어지면 질문을 하고 다시 복습을 하는 방향으로 해야 할 것 같다.이동현 : 코드 리뷰가 늘어났기 때문에 전체적인 속도는 줄어든 것 같다. 따라서 코드 리뷰와 진행 속도 둘 다 챙길 수 있는 방법을 찾아야 할 것 같다.이준호 : 없다.권재현 : 문제가 너무 어려워서 후반에 집중을 잘 못했다
구성원 학습활동 사진	
다음 학습할 내용	스택과 큐