REPORT

Data Structures

Student ID : 1771008
Name : Minjeong Kim

Homework3_1

headnode (= 데이터값이 없는 노드)가 존재하고, circular doubly linked list로 구현하였다.

Q1. dinsert_node를 사용해서 double linked list의 끝에 새로운 node 삽입하기

변수(variable) 분석

doubly linked list의 node는 struct로 구현한다.

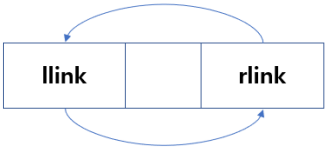
DlistNode	
type	**name**
element	data
DlistNode *	llink
DlistNode *	rlink

main함수에서 DlistNode를 할당하면 다음과 같은 노드가 생성된다.

함수(function) 분석

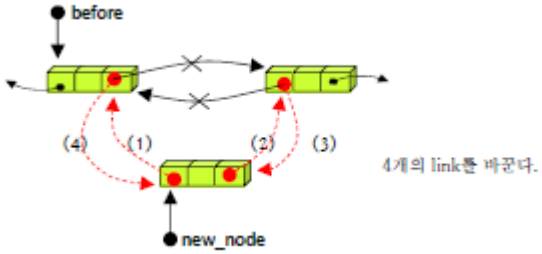
1. init()

main함수에 먼저 head_node를 생성하고, head_node를 init() : 초기화 한다.
초기화 결과 head_node는 data값이 없고, 자기가신을 가리키기 때문에
circular double linked list의 꼴이 된다.
다음과 같은 형태의 head_node가 생성되고, 이후 dinsert_node()함수를 사용해서
이 head_node 뒤에 새로운 node를 붙이기 때문에, circular double linked list의 형태이다.



2. dinsert_node(DlistNode *before, DlistNod *new_node)

ppt 41p 대로 아래그림을 구현한다.



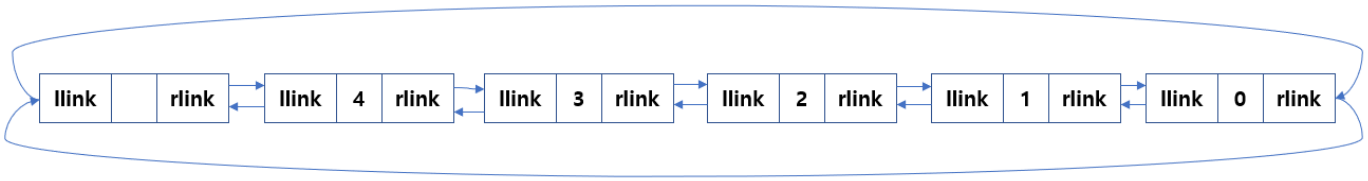
3. display(DlistNode *phead)

Node의 이전, 현재, 이후의 주소값을 확인하고, node의 data 값을 알아내기 위해 printf()를 사용해서 다음 형태로 출력한다. 이때, circle doubly linked list라는 특성 때문에 p가 rlink 값으로 반복될 때마다 바뀌게 되는데,
p의 값이 phead와 같아지면, 모든 node를 확인 한 것이기 때문에 조건절을 p!=phead로 주었다.
<---|llink address| data: node_data, address: node_address|rrlink address|--->

Answer Q1

head_node는 데이터 값이 존재하지 않는 node이기 때문에, 우리는 doubly linked list의 마지막 node는 head_node의 이전 node (llink)와 같다고 볼 수 있다.

headnode는 데이터값이 존재하지 않기 때문에, doubly linked list의 마지막 node는 headnode의 이전 node와 같다. 따라서 즉, 코드에 따르면 현재 list는



다음 꼴을 보이고 있는데, head_node는 데이터 값이 없는 노드이기 때문에, 데이터 값이 존재하는 node를 마지막에 insert한다는 뜻은 head_node의 llink node 즉 데이터값이 0번이 노드의 오른쪽에 삽입한다는 의미이므로,

함수 dinsert_node(DlistNode *before, DlistNode *new_node)에 before값으로 head_node.llink을 전달하면 된다.
즉, dinsert_node(head_node.llink, new_node) 로 before의 인자값을 설정하면 마지막에 insert가 된다.

결론적으로 dinsert_node를 사용해서 doubly linked list에 새로운 node를 insert하는 것은 **가능하다.**

addition

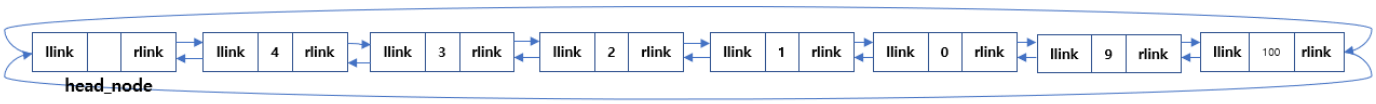
dinsert_node로 마지막에 새로운 node를 insert 할 수 있지만, before값에 head_node의 llink값을 전달하지않고,

dinsert_last_node함수로 head_node 값을 전달해주는 방법을 구현해 보았다.

```
void dinsert_last_node(DlistNode *phead, DlistNode *new_node) {
    new_node->rlink = phead;
    new_node->llink = phead->llink;
    phead->llink->rlink = new_node;
    phead->llink = new_node;
}
```

식은 다음과 같으며 인자로 받은 phead값을 이용해서 last node에 insert한다는 점에서 차이가 있다.

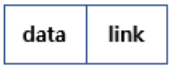
결국 본인의 Homework03_1.c로 형성된 circle doubly linked list의 형태는 아래와 같다.



Homework3_2

변수(variable) 분석

ListNode	
type	**name**
element	data
ListNode *	link



main 함수에서 ListNode를 할당하면 다음과 같은 데이터가 형성된다.

함수(function) 분석

1.init(ListNode **phead)

head pointer의 값을 NULL로 초기화한다.
이때, headpointer의 주소값을 받는 이유는,
headpointer는 node의 첫번째 값의 주소값을 저장하고 있는데,
subfunction에서 headpointer의 값이 바뀌었을 때, 반영을 하기위해headpointer의 주소값을 알아야한다.
즉 *phead는 첫번째 node의 주소를 담고 있고 (=headpointer)
**phead는 첫번째 node의 주소의 주소값을 담고 있다. (=headpointer의 주소값)

2. insert_last(ListNode **phead, element data)

data의 값으로 새로운 Node를 가장 마지막에 넣는다. 이때, 마지막 node에 접근하는 방법은, phead의 값을 받아 link가 NULL이 아닐때까지 반복하여 반복이 끝나면 마지막 node의 값에 도달하게 한다.

main함수에서 a[], b[] 배열안에 data값이 들어있는데,
이때 for문을 사용해서 배열의 각요소에 접근하고 매 반복마다 insert_last함수를 이용해서
a배열은 a의 headnode인 A에 linked list를 구현하고, b배열은 b의 headnode인 B에 linked list를 구현한다.

3. make_C(ListNode *A, ListNode *B, ListNode **C)

A linked list와 B linked list를 합친 값을 C linked list에 할당한다.
이때 while문을 사용하여 Node A와 Node B값이 null값일때까지 반복한다
A의 data값, B의 data값을 비교하여 더 작은 data값을 element input_C에 저장하고
B의 값을 B->link로 할당하여 다음 Node로 이동한다.
이후, insert_last를 이용해서 C linked list에 input_C의 값을 넣는다.

즉, data값이 작은 Node는 C에 값을 넣은 후 다음 Node로 이동하도록 구현한다.

차례차례 A,B모두 다음 Node로 이동하다보면, 한 Node의 값만 NULL이 되는 경우가 생긴다.
이런 경우의 예외처리를 위해 if문에 A== NULL || B==NULL을 설정하고,
A가 NULL인경우, B의 데이터를 넣고, B가 NULL인경우, A의 데이터를 넣는다.

A와 B모두 NULL값이 되면, A, B linked list안에 있는 data를 모두 확인 한 것이기 때문에, while을 종료한다.

시간복잡도 계산

make_C에 인자로 들어있는 A, B, C중에서
ListNode *A의 list size를 n이라고 가정하고, B의 list size를 m이라고 가정하자,
그러면 while문의 반복횟수는 n+m이되며,
매반복마다 element정의 1번, 중첩 if else문 2번, if-else문 안의 할당 2번,
insert_last()함수실행 x번으로 총,
(n+m)*(1+2+2+i) = (5+i)*(n+m)가 된다.

여기서 insert_last()함수를 보자면,
insert_last()함수의 phead의 list size x라고 하면
new_node 할당 1번, new_node 멤버할당 1번, if문 1번, MAX일경우는 else문이므로
else문안의 listNode 할당 1번, while loop x번 그리고 last->link 할당 1번이므로
insert_last()함수는 1+1+1+1+x+1 = x+5 번이다.

이때 다시 make_C함수로 돌아가면
(n+m)*(5+i)이고 i = x+5이므로
(n+m)*(10+x)이다.
이때 x값은 while 루프가 m+n번 진행됨에 따라, 1 2 3 ...m+n으로 증가하므로
(n+m)*(1+n+m)/2 이므로
(n+m)(10+(n+m)*(1+n+m)/2)
이때 m을 상수라고 가정하고, n에 대한 빅오 값은
(n+m)(10+(n+m)*(1+n+m)/2) <= c*n^3 이므로,
O(n^3)을 갖는다.

```
void make_C(ListNode *A, ListNode *B, ListNode **C) {
    while (!(A == NULL) || !(B == NULL)) {
        element input_C;                    n+m번 선언
        if (A == NULL || B == NULL) {       n+m번 비교
            if (A == NULL) {                n+m번 비교
                input_C = B->data;          n+m번 할당
                B = B->link;                n+m번 할당
            }
            else {
                input_C = A->data;
                A = A->link;
            }
        }
        else {
            if (A->data < B->data) {
                input_C = A->data;
                A = A->link;
            }
            else {
                input_C = B->data;
                B = B->link;
            }
        }
        insert_last(C, input_C);            i*(n+m)번 실행
    }
}
```

```
void insert_last(ListNode **phead, element data) {
    ListNode *new_node = (ListNode*)malloc(sizeof(ListNode));1번 정의

    new_node->link = NULL;  1번 할당
    new_node->data = data;  1번 할당
    if (*phead == NULL) {   1번 비교
        *phead = new_node;
    }
    else {
        struct ListNode *last = *phead; 1번 할당
        while (last->link != NULL) {
            last = last->link;          x번 할당
        }
        last->link = new_node;          1번 할당
    }
}
```

(n+m)*(5+i)　　i = (5+x)

= (n+m)*(5+5+x) = (n+m)*(10+x)　　// x = 1+2+3+ ... +(n+m) = (n+m)*(1+n+m)/2

= (n+m)*(10+(n+m)*(1+n+m)/2) <= c*n^3　∴O(n^3)

Homework3_3

변수(varialbe) 분석

ListNode	
type	**name**
element	data
ListNode *	link

ListType	
type	**name**
ListNode *	head
ListNode *	tail
int	length

main 함수에서 ListNode와 ListType을 할당하면 다음과 같은 데이터가 형성된다.

함수(function) 분석

1. init(ListType *list)

ListType은 하나의 linked list의 head와 tail node, 그 linked list의 길이를 저장해 놓은 구조체이다.
이 구조체의 head와 tail 값을 이용해서 해당 linked list의 함수 및 값의 접근을 좀더 쉽게 해준다.

2. get_node_at(ListType *list, int pos)

해당 pos에 위치한 linked list의 node를 return한다. pos만큼 반복문을 실행하여, 다음 node에 접근하고, 반복문이 끝나면 해당 pos에 있는 node가 나오게 된다.

3. add(ListType *list, int position, element data)

insert_node를 이용해서 원하는 position에 node를 insert한다. 이때,
insert_node함수의 before값을 찾기위해 get_node_at으로 position-1 위치의 node를 찾는다.
position 값이 0인 경우: get_node_at에서 position = -1 가 불가능하므로 add_first()함수를 이용한다.
position 값이 list의 length값인 경우: 그냥 add()만 했을 때, tail값이 변하지 않아, add_last()함수를 이용한다.

4. add_first(ListType *list, element data)

linked list의 첫번째에 insert하기 때문에 list의 head값을 이용한다.
인자로 받은 data값을 가진 new_node를 할당한다.
ㄱ. list의 head가 NULL 일 때: head와 tail 모두 new_node로 설정한다.
ㄴ. list의 head가 NULL 이 아닐 때: 먼저 new_node의 link를 head와 연결하고, head값은 new_node로 설정한다.

5. add_last(ListType *list, element data)

linked list의 마지막에 insert하기 때문에 list의 tail값을 이용한다.
인자로 받은 data 값을 가진 new_node를 할당한다.
ㄱ. list의 tail이 NULL 일 때: head와 tail 모두 new_node로 설정한다.
ㄴ. list의 tail이 NULL 이 아닐 때: tail이었던 node의 link를 new_node와 연결하고, tail값은 new_node로 설정한다.

6. delete(ListType *list, int pos)

remove_node를 이용해서 원하는 position에 있는 node를 없앤다 (= 메모리를 해제한다). 이때
remove_node함수의 before값을 찾기위해 get_node_at으로 position-1 위치의 node를 찾는다.
position 값이 0인 경우: get_node_at에서 postion -1 이 불가능하므로 delete_first()함수를 이용한다.
position 값이 list의 length-1 값인 경우(= 마지막 노드인 경우): 그냥 delete만 했을 때, tail 값이 변하지 않아, delete_last()함수를 이용한다.

7. is_in_list(ListType *list, element item)

해당 linked list의 data에 item이 있는 지 확인하는 함수이다.

list의 head포인터에 접근해서 p에 저장한다. p는 반복문 동안 다음 node로 계속 갱신되며, item값과 p의 data값이 일치할 경우 true를 return한다.

8. get_entry(ListType *list, int pos)

해당 position에 있는 data값을 return한다.

여기서 get_node_at과 헷갈릴 수 잇는데,

get_node_at의 return 타입은 ListNode이고, get_entry의 return타입은 element라는 점에서 차이가 있다.

위에 있는 함수10개를 이용해서 List ADT를 구현할 수 있다.