



자료구조와 실습

4장 리스트 연습문제



학번 : 2016110056

학과 : 불교학부

이름 : 박승원

날짜 : 2016년 10월 3일



1. 리스트에 대한 설명 중 틀린 것은?

- (a) 구조체도 리스트의 요소가 될 수 있다.
- (b) 리스트의 요소간에는 순서가 있다.
- (c) 리스트는 여러 가지 방법으로 구현될 수 있다.
- (d) 리스트는 집합과 동일하다.

2. 다음은 순차적 표현과 연결된 표현을 비교한 것이다. 설명이 틀린 것을 모두 표시하시오.

- (a) 연결된 표현은 포인터를 가지고 있어 상대적으로 크기가 작아진다.
- (b) 연결된 표현은 삽입이 용이하다.
- (c) 순차적 표현은 연결된 표현보다 접근 시간이 많이 걸린다.
- (d) 연결된 표현으로 작성된 리스트를 2개로 분리하기가 쉽다.

3. 다음은 연결 리스트에서 있을 수 있는 여러 가지 경우를 설명한다. 잘못된 항목은?

- (a) 정적인 데이터보다는 변화가 심한 데이터에서 효과적인 방법이다.
- (b) 모든 노드는 데이터와 링크(포인터)를 가지고 있어야 한다.
- (c) 연결 리스트에서 사용한 기억 장소는 다시 사용할 수 있다.
- (d) 데이터들이 메모리상에 흩어져서 존재할 수 있다.

4. 삽입과 삭제 작업이 자주 발생할 때 실행 시간이 가장 많이 소요되는 자료구조는?

- (a) 배열로 구현된 리스트
- (b) 단순 연결 리스트
- (c) 원형 연결 리스트
- (d) 이중 연결 리스트

5. 다음 중 NULL 포인터(NULL pointer)가 존재하지 않는 구조는 어느 것인가?

- (a) 단순 연결 리스트
- (b) 원형 연결 리스트
- (c) 이중 연결 리스트
- (d) 헤더 노드를 가지는 단순 연결 리스트

6. 원형 연결 리스트에 대한 설명 중 틀린 것은?

- (a) 모든 노드들이 연결되어 있다.
- (b) 마지막에 삽입하기가 간단하다.
- (c) 헤더 노드를 가질 수 있다.
- (d) 최종 노드 포인터가 NULL이다.

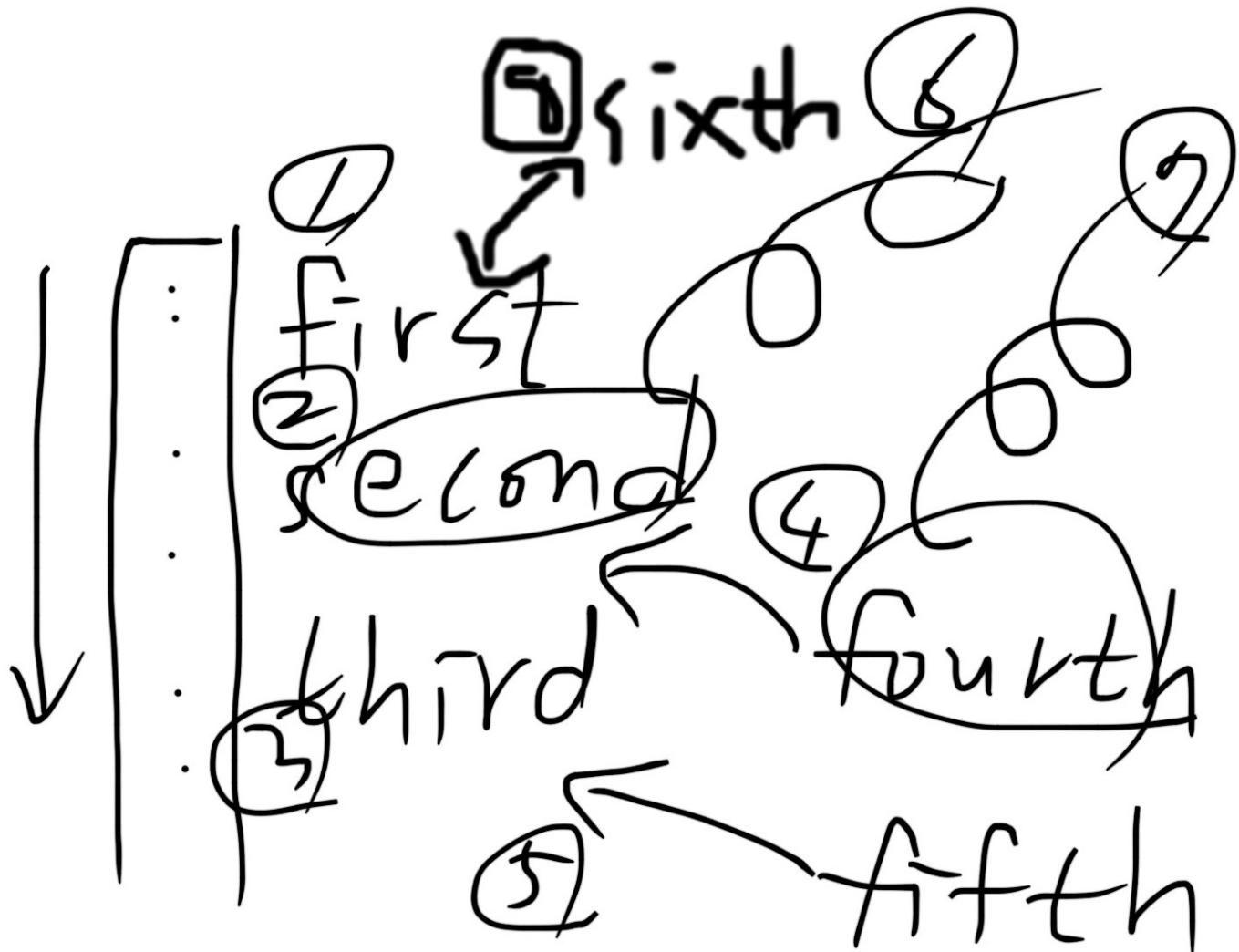
7. 리스트의 n번째 요소를 가장 빠르게 찾을 수 있는 구현 방법은 무엇인가?

- (a) 배열
- (b) 단순 연결 리스트

- (c) 원형 연결 리스트
 (d) 이중 연결 리스트
8. 단순 연결 리스트의 노드 포인터 p가 마지막 노드를 가리킨다고 할 때, 다음 수식 중 참인 것은?
- (a) $\text{last} == \text{NULL}$
 - (b) $\text{last} \rightarrow \text{data} == \text{NULL}$
 - (c) $\boxed{\text{last} \rightarrow \text{link} == \text{NULL}}$
 - (d) $\text{last} \rightarrow \text{link} \rightarrow \text{link} == \text{NULL}$
9. 단순 연결 리스트의 노드들을 노드 포인터 p로 탐색하고자 한다. p가 현재 가리키는 노드에서 다음 노드로 가려면 어떻게 하여야 하는가?
- (a) $\text{p}++;$
 - (b) $\text{p}--;$
 - (c) $\boxed{\text{p}=\text{p}\rightarrow \text{link};}$
 - (d) $\text{p}=\text{p}\rightarrow \text{data};$
10. 단순 연결 리스트의 관련 함수 f가 헤드 포인터 head를 변경시켜야 한다면 함수 매개 변수로 무엇을 받아야 하는가?
- (a) head
 - (b) $\boxed{\&\text{head}}$
 - (c) $\ast\text{head}$
 - (d) $\text{head}\rightarrow \text{link};$
11. A라는 공백 상태의 리스트가 있다고 가정하자. 이 리스트에 대하여 다음과 같은 연산들이 적용된 후의 리스트의 내용을 그려라.
- ```

add_first(A, "first");
add(A, 1, "second");
add_last(A, "third");
add(A, 2, "fourth");
add(A, 4, "fifth");
delete(A, 2);
delete(A, 2);
replace(A, 1, "sixth");

```



12. 배열을 이용하여 구현한 리스트의 경우, 리스트의 연산 중 일부 연산만 구현되어 있다. 본문의 코드를 참조하여 리스트 ADT의 나머지 연산들도 구현하여 보라.

```

#include <stdio.h>
#include <stdlib.h>

#define MAX_LIST_SIZE 100

typedef int element;
typedef struct {
 element list[MAX_LIST_SIZE];
 int length;
} ArrayListType;

void clear(ArrayListType* l) {
 l->length = 0;
}

void replace(ArrayListType* l, int pos, element item) {
 l->list[pos] = item;
}

```

```

element get_entry(ArrayListType* l, int pos) {
 return l->list[pos];
}

int get_length(ArrayListType* l) {
 return l->length;
}

```

13. 단순 연결 리스트에서 삭제 함수 delete 함수는 실제로는 헤드 포인터와 선행 노드 포인터의 2개의 매개변수만 있으면 작성이 가능하다. 이를 두 매개 변수만을 사용하여 다시 작성하라.

앞으로 계속 사용하게 될 헤더 파일 list.h

```

#include<stdio.h>
#include<time.h>
#include<stdlib.h>
typedef int element;
typedef struct List {
 element data;
 struct List* p;
} List;

List* setup(int n) {
 if(n == 0) return NULL;
 List* p = (List*)malloc(sizeof(List));
 p->data = rand() % 5;
 p->p = setup(n-1);
 return p;
}

void show(List* p) {
 if(p != NULL) {
 printf("%d ", p->data);
 show(p->p);
 }
}

void free_mem(List* p) {
 if(p == NULL) return;
 free_mem(p->p);
 free(p);
}

```

14. 단순연결 리스트에 정수가 저장되어 있다. 단순 연결 리스트의 모든 데이터 값은 더한 합을 출력하는 프로그램을 작성하시오.
15. 단순 연결 리스트에서 특정한 데이터 값을 갖는 노드의 개수를 계산하는 함수를 작성하여라.

16. 단순 연결 리스트에서 탐색 함수를 참고하여 특정한 데이터값을 갖는 노드를 삭제하는 함수를 작성하라.
17. 단순 연결 리스트의 헤드 포인터가 주어져 있을 때, 첫 번째 노드에서부터 하나씩 건너서 있는 노드를 삭제하는 함수를 작성하라. 즉, 홀수번째 있는 노드들이 전부 삭제된다.

문제 13 ~ 17

```
#include "list.h"

//13.

void remove_node(List *head, List *to_delete) {
 head->p = to_delete->p;
 free(to_delete);
}

//14.

int add(List* p)
{
 return p != NULL ? p->data + add(p->p) : 0;
}

//15.

int count(List* p, int n) {
 return p == NULL ? 0 : (p->data == n) + count(p->p, n);
}

//16~17.

List* del_if(List* p, int(*condition)(int)) {
 if(p != NULL) {
 if((*condition)(p->data)) {
 List* tmp = p->p;
 free(p);
 return del_if(tmp, condition);
 } else {
 p->p = del_if(p->p, condition);
 return p;
 }
 }
}

//16.

int del_2(int n) {
 return n == 2;
}

//17.

int del_everyother(int n) {
 static int k = 0;
 return k++ % 2 == 0;
}
```

```

int main()
{
 srand(time(NULL));
 List* p = setup(20);
 show(p);
 printf("\n2값은 %d개\n", count(p, 2));
 printf("합은 %d\n", add(p));
 p = del_if(p, del_2);
 show(p);
 printf("\n지운 후 2값은 %d개\n", count(p, 2));
 p = del_if(p, del_everyother);
 printf("\n둘 중 하나 지움. : "); show(p);
 remove_node(p, p->p);
 printf("\n2번째 지운 후 :"); show(p);
 free_mem(p);

}

```

```

3 0 4 2 4 0 4 4 3 3 4 2 2 2 1 3 4 0 2 2
2값은 6개
합은 49
3 0 4 4 0 4 4 3 3 4 1 3 4 0
지운 후 2값은 0개

둘 중 하나 지움. : 0 4 4 3 4 3 0
2번째 지운 후 : 0 4 3 4 3 0 ----- 문제 list번 실행을 종료합니다. -----

```

18. 두개의 단순 연결 리스트 A,B가 주어져 있을 경우, alternate 함수를 작성하라. alternate 함수는 A와 B로부터 노드를 번갈아 가져와서 새로운 리스트 C를 만드는 연산이다. 만약 입력 리스트 중에서 하나가 끝나게 되면 나머지 노드들을 전부 C로 옮긴다. 함수를 구현하여 올바르게 동작하는지 테스트하라. 작성된 함수의 시간 복잡도를 구하라.

```

#include "list.h"

List* alternate(List* p1, List* p2) {
 List *p, *head;
 p = head = (List*)malloc(sizeof(List));
 while (p1 != NULL || p2 != NULL) {
 if (p1 != NULL) {
 p->p = (List*)malloc(sizeof(List));
 p = p->p;
 p->data = p1->data;
 p1 = p1->p;
 }
 if (p2 != NULL) {
 p->p = (List*)malloc(sizeof(List));
 p = p->p;
 p->data = p2->data;
 p2 = p2->p;
 }
 }
}

```

```

List* tmp = head->p;
free(head);
return tmp;
}

int main() {
 List *p = setup(10);
 printf("\np : "); show(p);
 List *q = setup(15);
 printf("\nq : "); show(q);
 List* r = alternate(p, q);
 printf("\nnp,q 번갈아서 합침. : "); show(r);
 printf("\n");
 free_mem(p);
 free_mem(q);
 free_mem(r);
}

```

8n+3

```

zezeon@ubuntuZ:~/Programming/exercise$ make tex
gcc 18.c -o 18.x -g -fmax-errors=1
----- 문제 18번 실행을 시작합니다. -----
./18.x

p : 3 1 2 0 3 0 1 2 4 1
q : 2 2 0 4 3 1 0 1 2 1 1 3 2 4 2
p,q 번갈아서 합침. : 3 2 1 2 2 0 0 4 3 3 0 1 1 0 2 1 4 2 1 1 1 3 2 4 2
----- 문제 18번 실행을 종료합니다. -----

```

19. 원형 연결 리스트에서 헤드 노드를 사용하면 탐색 연산을 조금 빠르게 할수 있다. 즉 원형 연결 리스트에 아무런 역할도 하지 않는 헤드 노드를 하나 추가한다. 탐색 연산을 할 때 찾고자 하는 데이터를 헤드 노드에 미리 저장한다. 헤드 노드 다음 노드부터 탐색을 시작하여 만약 탐색이 헤드 노드에서 끝나게 되면 탐색은 실패한 것이다.

- (a) 다음과 같은 탐색연산을 원형 연결 리스트에 대하여 구현하고 테스트하라.

```

ListNode* search(ListNode *head, int data)
{
 ListNode *current = head->link;
 head->data = data;
 while (current->data != x) {
 current = current->link;
 }
 return ((current == head) ? NULL : current);
}

```

- (b) 이 원형 연결 리스트에서의 탐색 search와 단순 연결 리스트에서의 탐색 연산인 search의 실행 성능을 다음과 같이 비교하라. 단순 연결 리스트와 원형 연결 리스트의 크기가 각각 100,1000,10000,100000인 경우의 최악과 평균 실행 시간을 구하여 비교하라. 실행 시간이 차이가 나는 이유를 설명하시오.

```

#include<stdio.h>
#include<time.h>

```

```

#include "list.h"

List* search(List* head, element data)
{
 List *current = head->p;
 head->data = data;
 for(; current->data != data; current = current->p);
 return current == head ? NULL : current;
}

List* search2(List* l, element data) {
 for(; l; l = l->p) if(l->data == data) return l;
 return NULL;
}

int main() {
 for(int k=100; k!=1000000; k *=10) {
 double sum1=0, sum2=0;
 double max1=0, max2=0;
 for(int i=0; i<100; i++) {
 List* p = setup(k);
 List* head = (List*)malloc(sizeof(List));
 head->p = p;
 head->data = 6;

 clock_t start, end;
 start = clock();
 List* r = search2(p, 6);
 end = clock();
 double t = (double)(end - start);
 sum1 += t;
 max1 = max1 < t ? t : max1;
 if(r) printf("%d", r->data);
 List* tmp;
 for(; p; p = p->p) tmp = p;
 tmp->p = head;

 start = clock();
 r = search(head, 6);
 end = clock();
 t = (double)(end - start);
 sum2 += t;
 max2 = max2 < t ? t : max2;

 if(r) printf("%d", r->data);
 tmp->p = NULL;
 free(head);
 }
 }
}

```

```

 printf("%d개의 요소에서 헤드없는 탐색의 수행시간은 %lf\n", k, sum1/100);
 printf("%d개의 요소에서 헤드없는 탐색의 최악 수행시간은 %lf\n", k, max1);

 printf("%d개의 요소에서 헤드를 이용한 탐색의 수행시간은 %lf\n", k, sum2/100);
 printf("%d개의 요소에서 헤드를 이용한 탐색의 최악수행시간은 %lf\n", k, max2);
 }
}

```

----- 문제 19번 실행을 시작합니다 . -----  
./19.x  
100개의 요소에서 헤드없는 탐색의 수행시간은 3.090000  
100개의 요소에서 헤드없는 탐색의 최악 수행시간은 5.000000  
100개의 요소에서 헤드를 이용한 탐색의 수행시간은 2.760000  
100개의 요소에서 헤드를 이용한 탐색의 최악수행시간은 4.000000  
1000개의 요소에서 헤드없는 탐색의 수행시간은 20.770000  
1000개의 요소에서 헤드없는 탐색의 최악 수행시간은 107.000000  
1000개의 요소에서 헤드를 이용한 탐색의 수행시간은 17.210000  
1000개의 요소에서 헤드를 이용한 탐색의 최악수행시간은 42.000000  
10000개의 요소에서 헤드없는 탐색의 수행시간은 48.040000  
10000개의 요소에서 헤드없는 탐색의 최악 수행시간은 246.000000  
10000개의 요소에서 헤드를 이용한 탐색의 수행시간은 43.580000  
10000개의 요소에서 헤드를 이용한 탐색의 최악수행시간은 309.000000  
100000개의 요소에서 헤드없는 탐색의 수행시간은 457.750000  
100000개의 요소에서 헤드없는 탐색의 최악 수행시간은 561.000000  
100000개의 요소에서 헤드를 이용한 탐색의 수행시간은 424.010000  
100000개의 요소에서 헤드를 이용한 탐색의 최악수행시간은 464.000000  
----- 문제 19번 실행을 종료합니다 . -----

그다지 실행시간에 차이는 없었으나, 헤드를 이용한 탐색의 경우, 시간복잡도가  $n^2$ 이 더 적다. 즉 헤드가 없을 경우에는 항상 NULL인지 아닌지를 검사하는 과정이 한번 더 소요된다.

20. 두 개의 단순 연결 리스트를 병합하는 함수를 조금 변경하여 보자. 두개의 연결 리스트  $a = (a_1, a_2, \dots, a_n)$ ,  $b = (b_1, b_2, \dots, b_n)$ 가 데이터 값의 오름차순으로 노드들이 정렬되어 있는 경우, 이러한 정렬 상태를 유지하면서 합병을 하여 새로운 연결 리스트를 만드는 알고리즘 merge를 작성하라. a와 b에 있는 노드들은 전부 새로운 연결 리스트로 옮겨진다. 작성된 알고리즘의 시간복잡도도 구하라.

```

#include "list.h"

List* add(List* L, element n) {
 if (L == NULL || L->data > n) {
 List* new = (List*)malloc(sizeof(List));
 new->data = n;
 new->p = L;
 return new;
 } else {
 L->p = add(L->p, n);
 return L;
 }
}

List* merge(List* p1, List* p2) {
 List *p, *head;
 p = head = (List*)malloc(sizeof(List));
 while (p1 != NULL || p2 != NULL) {

```

```

p->p = (List*)malloc(sizeof(List));
p = p->p;
if(p2 == NULL) {
 p->data = p1->data;
 p1 = p1->p;
} else if(p1 == NULL) {
 p->data = p2->data;
 p2 = p2->p;
} else {
 if(p1->data > p2->data) {
 p->data = p2->data;
 p2 = p2->p;
 } else {
 p->data = p1->data;
 p1 = p1->p;
 }
}
p->p = NULL;
}

List* tmp = head->p;
free(head);
return tmp;
}

int main() {
 List *p = NULL;
 p = add(p, 1);
 p = add(p, 2);
 p = add(p, 5);
 p = add(p, 6);
 p = add(p, 8);
 List* q = NULL;
 q = add(q, 3);
 q = add(q, 4);
 q = add(q, 7);
 q = add(q, 9);
 q = add(q, 11);
 printf("\np : "); show(p);
 printf("\nq : "); show(q);
 List* r = merge(p, q);
 printf("\nnp,q 빙갈아서 합침. : "); show(r);
 printf("\n");
 free_mem(p);
 free_mem(q);
 free_mem(r);
}

```

```

p : 1 2 5 6 8
q : 3 4 7 9 11
p,q 번갈아서 합침. : 1 2 3 4 5 6 7 8 9 11
----- 문제 20번 실행을 종료합니다. -----

```

7n+3

21. 보통 연결 리스트에서는 선행 노드를 알아야만이 노드를 삭제할 수 있다. 그러나, 다음과 같이 하면 선행 노드를 모르고도 노드를 삭제할 수 있다. 먼저 원형 연결 리스트라고 가정하자. 어떤 노드를 가리키는 포인터 x가 주어진 경우, 그 노드의 후속 노드를 쉽게 찾을 수 있다. 후속 노드를 y라고 한다면 x에 y의 데이터 필드값을 복사한다. 그리고 y를 삭제한다. 그러면 실제로 x가 삭제된 것처럼 된다. 이런 식으로 노드를 삭제하는 함수 `remove_node2`를 작성하고 구현하여 테스트하라.

```

#include "list.h"

void remove_node2(List* x) {
 x->data = x->p->data;
 x->p = x->p->p;
 free(x->p);
}

int main() {
 List* p = setup(5);
 printf("p :"); show(p);
 remove_node2(p->p);
 printf("\np :"); show(p);
}

```

```

gcc 21.c -o 21.x -g -fmax-errors=1
-----none 문제 21번 실행을 시작합니다. -----
./21.x
p :3 1 2 0 3
p :3 2 0 3 ----- 문제 21번 실행을 종료합니다. -----

```

22. 단순 연결 리스트 C를 두개의 단순 연결 리스트 A와 B로 분리하는 함수 `split`를 작성하여 보자. C의 홀수번째 노드들은 모두 A로 이동되고 C의 짝수번째 노드들은 모두 B로 이동된다. 이 함수가 C를 변경하여서는 안된다. 작성된 알고리즘의 시간 복잡도를 구하고 구현하여 수행하여 보자.

```

#include "list.h"

List* split(List* from) {
 if(from == NULL) return NULL;
 List* p = (List*)malloc(sizeof(List));
 p->data = from->data;
 p->p = from->p == NULL ? NULL : split(from->p->p);
 return p;
}

int main() {
 List* p = setup(11);

```

```

List* q = split(p);
List* r = split(p->p);
printf("p : "); show(p);
printf("\nq : "); show(q);
printf("\nr : "); show(r);
}

```

23. 두 개의 다항식이 다음과 같이 주어졌다. 이들을 연결 리스트를 이용하여 나타내고 본문의 프로그램을 이용하여 두 다항식의 합을 구해보시오.

$$A(x) = 2x^6 + 7x^3 - 2x^2 - 9, B(x) = -2x^6 - 4x^4 + 6x^2 + 6x + 1$$

24. 다항식을 연결 리스트로 표현할 수 있음을 보였다. 다항식이 연결 리스트로 표현되어 있고, p를 다항식을 가리키는 포인터라고 할 때, 어떤 실수 x에 대하여 이 다항식의 값을 계산하는 함수 poly\_eval을 작성하라. 즉, 다항식이  $x^3 + 2x + 6$ 이고,  $x=2$ 라면  $2^3 + 2 * 2 + 6$ 를 계산하는 함수를 작성하여보라.
25. 다항식이 연결 리스트로 표현되어 있는 경우, 두 개의 다항식을 받아서 다항식의 뺄셈을 수행하는 함수 poly\_sub를 작성하라.

### 문제 23 ~ 25

```

#include<stdio.h>
#include<stdlib.h>
#include<math.h>
typedef struct List {
 int exp, coef;
 struct List* node;
} List;

List* add(List* l, int coef, int exp) {
 if(l == NULL || exp > l->exp) {
 List* new = (List*)malloc(sizeof(List));
 new->coef = coef;
 new->exp = exp;
 new->node = l;
 return new;
 } else if(exp == l->exp) l->coef += coef;
 else l->node = add(l->node, coef, exp);
 return l;
}

void display(List* l) {
 while(l != NULL) {
 if(l->coef) printf("%c%dx^%d", l->coef<0 ? ' ' : '+', l->coef, l->exp);
 l = l->node;
 }
}

```

```

 }

 }

List* poly_add(List* A, List* B) {
 List* C = NULL;
 while(A != NULL) {
 C = add(C, A->coef, A->exp);
 A = A->node;
 }
 while(B != NULL) {
 C = add(C, B->coef, B->exp);
 B = B->node;
 }
 return C;
}

List* poly_sub(List* A, List* B) {
 List* C = NULL;
 while(A != NULL) {
 C = add(C, A->coef, A->exp);
 A = A->node;
 }
 while(B != NULL) {
 C = add(C, -B->coef, B->exp);
 B = B->node;
 }
 return C;
}

float poly_eval(List* A, float n) {
 if(A == NULL) return 0;
 return pow(n, A->exp) * A->coef + poly_eval(A->node, n);
}

int main() {
 List* A = NULL;
 A = add(A, 2, 6);
 A = add(A, 7, 3);
 A = add(A, -2, 2);
 A = add(A, -9, 0);
 printf("\nA : ");
 display(A);

 printf("\nB : ");
 List* B = NULL;
 B = add(B, -2, 6);
 B = add(B, -4, 4);
 B = add(B, 6, 2);
}

```

```

B = add(B, 6, 1);
B = add(B, 1, 0);
display(B);

printf("\nA+B : ");
List* C = poly_add(A, B);
display(C);

printf("\n3을 대입한 값은 %f", poly_eval(C, 3));
List* D = poly_sub(A, B);
printf("\nA-B : ");
display(D);
}

```

```

gcc 23.c -o 23.x -g -fmax-errors=1 -lm
zezeon@ubuntuZ:~/Programming/exercise$ make 23.png
----- 문제 23번 실행을 시작합니다. -----
./23.x
A : +2x^6+7x^3 -2x^2-9x^0
B : -2x^6 -4x^4+6x^2+6x^1+x^0
A+B : -4x^4+7x^3+4x^2+6x^1 -8x^0
3을 대입한 값은 -89.000000
A-B : +4x^6+4x^4+7x^3 -8x^2 -6x^1 -10x^0
----- 문제 23번 실행을 종료합니다. -----

```

26. 다음의 연산에 대해 이중 연결 리스트가 단일 연결 리스트보다 좋은 점이 있는가를 설명하라.

- (a) 각 노드를 처리하기 위해 리스트를 순회한다.
- (b) loc 위치에 있는 노드를 삭제한다.
- (c) loc의 위치에 있는 노드 바로 앞에 새 노드를 삽입한다.
- (d) loc의 위치에 있는 노드 바로 뒤에 새 노드를 삽입한다.

27. 배열을 이용하여 숫자들을 입력 받아 항상 정렬된 상태로 유지하는 리스트 SortedList를 구현하여 보라. 다음의 연산들을 구현하면 된다.

#### ADT 4.2 SortedList

- 객체 : n개의 element형으로 구성된 순서있는 모임.
- 연산 :
  - `add(list, item) ::=` 정렬된 리스트에 요소를 추가한다.
  - `delete(list, item) ::=` 정렬된 리스트에서 item을 제거한다.
  - `clear(list) ::=` 리스트의 모든 요소를 제거한다.
  - `is_in_list(list, item) ::=` item이 리스트 안에 있는지를 검사한다.
  - `get_length(list) ::=` 리스트의 길이를 구한다.
  - `is_empty(list) ::=` 리스트가 비었는지를 검사한다.
  - `is_full(list) ::=` 리스트가 꽉찼는지를 검사한다.
  - `display(list) ::=` 리스트의 모든 요소를 표시한다.

```

#include <stdio.h>
#define MAX 100
typedef int element;

typedef struct SortedList {
 element data[MAX];
 int length;
} SortedList;

void add(SortedList* L, element n) {
 if(L->length == 0) {
 L->length = 1;
 L->data[0] = n;
 } else {
 int i;
 for(i=0; i<L->length; i++) if(L->data[i] > n) break;
 for(int j=L->length; j>i; j--) L->data[j] = L->data[j-1];
 L->data[i] = n;
 L->length++;
 }
}

void display(SortedList* L) {
 for(int i=0; i<L->length; i++) printf("%d ", L->data[i]);
}

int is_in(SortedList* L, element n) {
 for(int i=0; i<L->length; i++) {
 if(n == L->data[i]) return 1;
 else if(n < L->data[i]) return 0;
 }
 return 0;
}

void clear(SortedList* L) {
 L->length = 0;
}

int get_length(SortedList* L) {
 return L->length;
}

int is_empty(SortedList* L) {
 return L->length == 0;
}

int is_full(SortedList* L) {
 return L->length == MAX;
}

```

```

}

int main()
{
 SortedList list;
 list.length = 0;
 if(is_empty(&list)) printf("리스트는 비어있습니다.\n");
 add(&list, 8);
 add(&list, 3);
 add(&list, 1);
 add(&list, 5);
 add(&list, 6);
 display(&list);
 if(is_in(&list, 3)) printf("\n3은 리스트에 있습니다.\n");
 printf(" 리스트의 사이즈는 %d\n", get_length(&list));
 clear(&list);
 printf("지운 후 리스트의 데이터 : ");
 display(&list);

}

```

```

3은 리스트에 있습니다 .
리스트의 사이즈는 5
지운 후 리스트의 데이터 : zezeon@ubuntuZ:~/Programming/exercise$ make tex
gcc 27.c -o 27.x -g -fmax-errors=1
----- 문제 27번 실행을 시작합니다 . -----
./27.x
리스트는 비어있습니다 .
1 3 5 6 8
3은 리스트에 있습니다 .
리스트의 사이즈는 5
지운 후 리스트의 데이터 : ----- 문제 27번 실행을 종료합니다 . -----

```

28. 단순 연결 리스트를 이용하여 숫자들을 항상 정렬된 상태로 유지하는 리스트 SortedList를 구현하여 보자. 앞의 문제의 연산들을 구현하면 된다.

```

#include <stdio.h>
typedef int element;

typedef struct SortedList {
 element data;
 struct SortedList* node;
} SortedList;

SortedList* add(SortedList* L, element n) {
 if(L == NULL || L->data > n) {
 SortedList* new = (SortedList*)malloc(sizeof(SortedList));
 new->data = n;
 new->node = L;
 return new;
 } else {

```

```

 L->node = add(L->node, n);
 return L;
 }

}

void display(SortedList* L) {
 if(L == NULL) return;
 printf("%d ", L->data);
 display(L->node);
}

int is_in(SortedList* L, element n) {
 if(L == NULL) return 0;
 if(L->data == n) return 1;
 else return is_in(L->node, n);
}

SortedList* clear(SortedList* L) {
 if(L != NULL) {
 clear(L->node);
 free(L);
 }
 return NULL;
}

int get_length(SortedList* L) {
 if(L == NULL) return 0;
 return get_length(L->node) + 1;
}

int main()
{
 SortedList* list = NULL;
 list = add(list, 3);
 list = add(list, 1);
 list = add(list, 7);
 list = add(list, 2);
 display(list);
 if(is_in(list, 7)) printf("\n7은 리스트에 있습니다.\n");
 printf("길이는 %d\n", get_length(list));
 list = clear(list);
 display(list);
}

```

```

evince Exercise04.pdf
zezeon@ubuntuZ:~/Programming/exercise$ rm 28.png
zezeon@ubuntuZ:~/Programming/exercise$ make tex
----- 문제 28번 실행을 시작합니다. -----
./28.x
1 2 3 7
7은 리스트에 있습니다.
길이는 4
----- 문제 28번 실행을 종료합니다. -----

```

29. 이중 연결 리스트를 이용하여 숫자들을 항상 정렬된 상태로 유지하는 리스트 SortedList를 구현하여 보자. 앞의 문제의 연산들을 구현하면 된다.

```

#include <stdio.h>
typedef int element;

typedef struct SortedList {
 element data;
 struct SortedList *head, *tail;
} SortedList;

SortedList* add(SortedList* L, element n) {
 if(L == NULL || L->data > n) {
 SortedList* new = (SortedList*)malloc(sizeof(SortedList));
 new->data = n;
 new->node = L;
 return new;
 } else {
 L->node = add(L->node, n);
 return L;
 }
}

void display(SortedList* L) {
 if(L == NULL) return;
 printf("%d ", L->data);
 display(L->node);
}

int is_in(SortedList* L, element n) {
 if(L == NULL) return 0;
 if(L->data == n) return 1;
 else return is_in(L->node, n);
}

SortedList* clear(SortedList* L) {
 if(L != NULL) {
 clear(L->node);
 free(L);
 }
 return NULL;
}

```

```

}

int get_length(SortedList* L) {
 if(L == NULL) return 0;
 return get_length(L->node) + 1;
}

int main()
{
 SortedList* list = NULL;
 list = add(list, 3);
 list = add(list, 1);
 list = add(list, 7);
 list = add(list, 2);
 display(list);
 if(is_in(list, 7)) printf("\n7은 리스트에 있습니다.\n");
 printf("길이는 %d\n", get_length(list));
 list = clear(list);
 display(list);
}

```

30. 본문에서 언급한 대로 연결 리스트에서 헤더 노드(header node)의 개념을 사용하는 것은 상당한 장점이 있다. 먼저 어떤 장점이 있는지를 말하고 단순 연결 리스트의 함수들을 다음과 같은 헤더 노드의 포인터를 받아서 연산을 수행하도록 변경하여 보라.

```

#include "list.h"
// 연결 리스트 헤더
typedef struct ListHeader {
 int length;
 List* head;
 List* tail;
} ListHeader;

// 헤더노드를 이용한 삽입함수
void insert_node(ListHeader *header, List *new_node)
{
 List *p, *prev;
 for(p = header->head; p != new_node; p = p->p) prev = p;
 prev->p = (List*)malloc(sizeof(List));
 prev->p->p = new_node;
}

// 헤더 노드를 이용한 삭제함수
void remove_node(ListHeader *header, List *removed)
{
 List *p, *prev;
 for(p = header->head; p != removed; p = p->p) prev = p;
 prev->p = p->p;
 free(p);
}

```

```

}

// 헤더노드를 이용한 공백 검사 함수
int is_empty(ListHeader *header)
{
 return length == 0;
}

// 헤더노드를 이용한 검색 함수
List* search(ListHeader *header, element data)
{
 for(List* p = header->head; p != header->tail; p = p->p)
 if(p->data == element) return p;
 return NULL;
}

```

단순연결 리스트에서 리턴값을 잘 이용하고, 재귀함수를 잘 활용하면, 매우 간단하고 깔끔하게 리스트를 작성할 수 있다. 헤더노드 없이도 충분히 가능한데, 헤더노드를 사용하면 프로그램이 약간 지저분해지는 느낌이 들지만, 프로그래밍적 편의성 면과 실행 성능 측면에서는 더 낫다. 단순연결리스트의 장점은 매우 직관적이고 간명한 프로그램을 짤 수 있다는 것이다.

31. 행렬은 숫자나 문자를 정사각형 또는 직사각형으로 배열하여 그 양끝을 괄호로 묶은 것으로 많은 문제를 수학적으로 해결하는 도구이다. 희소 행렬(sparse matrix)은 많은 항들이 0인 행렬이다. 연결리스트를 이용하여 희소행렬을 표현하는 방법을 생각하여 보고 구현하라.

```

struct SparseMatrix { //첫번째 노드에서는 행렬의 너비 높이 0이 아닌 값의 갯수이고 ,
 int x, y, v; //두번째 노드부터는 x, y좌표와 그 값이다 .
 struct SparseMatrix* node;
} SparseMatrix;

```

### 리스트를 활용한 Line Editor

```

#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#define MAX 200

typedef struct Line {
 char line[MAX];
 struct Line* node;
} Line;

Line* insert_line(Line* l, int pos, char* contents) {
 if(l == NULL) {
 Line* new = (Line*)malloc(sizeof(Line));
 strcpy(new->line, contents);
 new->node = NULL;
 return new;
 } else if(pos == 0) {
 Line* new = (Line*)malloc(sizeof(Line));

```

```

 strcpy(new->line, contents);
 new->node = 1;
 return new;
 } else {
 l->node = insert_line(l->node, pos-1, contents);
 return l;
 }
}

Line* delete_line(Line* l, int pos) {
 if(l == NULL) return NULL;
 else if(pos == 0) {
 Line* tmp = l->node;
 free(l);
 return tmp;
 } else {
 l->node = delete_line(l->node, pos - 1);
 return l;
 }
}

void display(Line* l, int pos) {
 for(int i=1; l && i<=pos; l=l->node) printf("%d : %s", i++, l->line);
}

void save(Line* l) {
 FILE* f=fopen("sav.txt", "w");
 for(; l; l=l->node) fputs(l->line, f);
 fclose(f);
}

Line* command(Line* l) {
 int c, line;
 char buffer[MAX];
 printf("1.Delete[line], 2.View[line], 3.Insert[line] 4.Save 5.Quit\n");
 printf("입력할 명령은 ?>>");
 fflush(stdin);
 scanf("%d", &c); scanf("%d", &line);fflush(stdin);
 switch(c) {
 case 1: return delete_line(l, line-1);
 case 2: display(l, line); return l;
 case 3: fgets(buffer, MAX, stdin);
 return insert_line(l, line-1, buffer);
 case 4: save(l); return l;
 case 5: return NULL;
 }
}

```

```

int main(int c, char** v) {
 if(c < 2) return 0;
 FILE* f = fopen(v[1], "r");
 char buffer[MAX];
 Line* l = NULL;
 while(fgets(buffer, MAX, f)) l = insert_line(l, 1000, buffer);
 display(l, 100);
 fclose(f);
 while(l==command(l));
}

```

```

zezeon@ubuntuZ:~/Programming/exercise$./lineeditor.x sav.txt
1 : #include <stdio.h>
2 : #include <stdlib.h>
3 : #define MAX_LIST_SIZE 100
4 :
5 : typedef int element;
6 : typedef struct {
7 : element list[MAX_LIST_SIZE];
8 : int length;
9 : } ArrayListType;
10 :
11 : int main() {}
1.Delete[line], 2.View[line], 3.Insert[line] 4.Save 5.Quit
입력 할 명령은?>>3 10
1.Delete[line], 2.View[line], 3.Insert[line] 4.Save 5.Quit
입력 할 명령은?>>fdsf fsdfs
1.Delete[line], 2.View[line], 3.Insert[line] 4.Save 5.Quit
입력 할 명령은?>>2 33
1 : #include <stdio.h>
2 : #include <stdlib.h>
3 : #define MAX_LIST_SIZE 100
4 :
5 : typedef int element;
6 : typedef struct {
7 : element list[MAX_LIST_SIZE];
8 : int length;
9 : } ArrayListType;
10 : fsdf fsdfs
11 :
12 :
13 : int main() {}
1.Delete[line], 2.View[line], 3.Insert[line] 4.Save 5.Quit
입력 할 명령은?>>■

```

## 소감

리턴값을 잘 활용하면 리스트를 깔끔하게 만들 수 있다. 리스트는 재귀함수를 많이 활용할 수 있는 부분이었다. 포인터를 다루는 것은 아직도 많이 실수를 유발한다. 리스트를 좀 더 활용해 보기 위해 라인에디터를 직접 만들어 보았다. 라인을 삽입시에 개행문자가 하나 끼어드는 버그가 있다. 왜 그런지 잘 모르겠다. scanf를 fgets로 바꾸어 보기로 했는데 해결이 안된다.