ch4.

1. 리스트에 대한 설명중 틀린 것은?
(1) 구조체도 리스트의 요소가 될 수 있다.
(2) 리스트의 요소간에는 순서가 있다.
(3) 리스트는 여러가지 방법으로 구현될 수 있다.
(4) 리스트는 집합과 동일하다.

2. 다음은 순차적 표현과 연결된 표현을 비교한 것이다. 설명이 틀린 것은?
(1) 연결된 표현은 포인터를 가지고 있어 상대적으로 크기가 작아진다.
(2) 연결된 표현은 삽입이 용이하다.
(3) 순차적 표현은 연결된 표현보다 액세스 시간이 많이 걸린다.
(4) 연결된 표현으로 작성된 리스트를 2개로 분리하기가 쉽다.

3. 다음은 연결 리스트에서 있을 수 있는 여러 가지 경우를 설명했는데 잘못된 항목은?
(1) 정적인 데이터보다는 다양하고 변화있는 데이터에서 효과적인 방법이다.
(2) 모든 노드는 데이터와 링크(포인터)를 가지고 있어야 한다.
(3) 연결 리스트에서 사용한 기억장소는 다시 사용할 수 없다.
(4) 데이터들이 메모리상에 흩어져서 존재할 수 있다.

4. 삽입과 삭제작업이 자주 발생할 때 실행시간이 가장 많이 소요되는 자료구조는?
(1) 배열로 구현된 리스트
(2) 단순 연결 리스트
(3) 원형 연결 리스트
(4) 이중 연결 리스트

5. 다음 중 NULL 포인터(NULL pointer)가 존재하지 않는 구조는 어느 것인가?
(1) 단순 연결 리스트
(2) 원형 연결 리스트
(3) 이중 연결 리스트
(4) 헤더 노드를 가지는 단순 연결 리스트

6. 원형 연결 리스트에 대한 설명 중 틀린 것은 ?
(1) 모든 노드들이 연결되어 있다.
(2) 마지막에 삽입하기가 간단한다.
(3) 헤더 노드를 가질 수 있다.
(4) 최종 노드 포인터가 NULL이다.

7. 리스트의 n번째 요소를 가장 빠르게 찾을 수 있는 구현 방법은 무엇인가?
(1) 배열 (2) 단순 연결 리스트 (3) 원형 연결 리스트 (4) 이중 연결 리스트

8. 단순 연결 리스트의 노드 포인터 last가 마지막 노드를 가리킨다고 할 때 다음 수식 중, 참인 것은?
(1) `last == NULL`
(2) `last->data == NULL`
(3) `last->link == NULL`
(4) `last->link->link == NULL`

9. 단순 연결 리스트의 노드들을 노드 포인터 p로 탐색하고자 한다. p가 현재 가리키는 노드에서 다음 노드로 가려면 어떻게 하여야 하는가?
(1) `p++;` (2) `p--;` (3) `p=p->link;` (4) `p=p->data;`

10. 단순 연결 리스트의 관련 함수 f가 헤드 포인터 head를 변경시켜야 한다면 함수 파라미터로 무엇을 받아야 하는가?
(1) `head` (2) `&head` (3) `*head` (4) `head->link;`

11. A라는 공백상태의 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, 3, "sixth");
```

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

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

#define MAX_LIST_SIZE 100 // 배열의 최대크기
#define TRUE 1
#define FALSE 0

typedef int element;
typedef struct {
	int list[MAX_LIST_SIZE];	// 배열 정의
	int length;			// 현재 배열에 저장된 자료들의 개수
} ArrayListType;

// 오류 처리 함수
void error(char *message)
{
	fprintf(stderr,"%s₩n",message);
	exit(1);
}
// 리스트 초기화
void init(ArrayListType *L)
{
	L->length = 0;
}
// 리스트가 비어 있으면 1을 반환
// 그렇지 않으면 0을 반환
int is_empty(ArrayListType *L)
{
	return L->length == 0;
}
```

```
// 리스트가 가득 차 있으면 1을 반환
// 그렇지 않으면 1을 반환
int is_full(ArrayListType *L)
{
	return L->length == MAX_LIST_SIZE;
}
// 리스트 출력
void display(ArrayListType *L)
{
	int i;
	for(i=0;i<L->length;i++)
		printf("%d₩n", L->list[i]);
}
// position: 삽입하고자 하는 위치
// item: 삽입하고자 하는 자료
void add(ArrayListType *L, int position, element item)
{
	if( !is_full(L) && (position >= 0) &&
			(position <= L->length) ){
		int i;
		for(i=(L->length-1); i>=position;i--)
			L->list[i+1] = L->list[i];
		L->list[position] = item;
		L->length++;
	}
}
void add_first(ArrayListType *L, element item)
{
	add(L,0,item);
}
void add_last(ArrayListType *L, element item)
{
	add(L,L->length,item);
}
// position: 삭제하고자 하는 위치
// 반환값: 삭제되는 자료
element delete(ArrayListType *L, int position)
{
	int i;
	element item;

	if( position < 0 || position >= L->length )
		error("위치 오류");
```

```
        item = L->list[position];
        for(i=position; i<(L->length-1);i++)
                L->list[i] = L->list[i+1];
        L->length--;
        return item;
}
void clear(ArrayListType *L)
{
        L->length=0;
}
void replace(ArrayListType *L, int position, element item)
{
        L->list[position]=item;
}
int is_in_list(ArrayListType *L, element item)
{
        int i;
        for(i=0; i<(L->length);i++)
                if(L->list[i]==item) return TRUE;
        return FALSE;
}
element get_entry(ArrayListType *L, int position)
{
        if( position < 0 || position >= L->length )
                error("위치 오류");
        return L->list[position];
}
//
main()
{
        ArrayListType list1;

        // ListType를 정적으로 생성하고 ListType를 가리키는
        // 포인터를 함수의 파라미터로 전달한다.
        init(&list1);
        add_first(&list1, 10);
        add_first(&list1, 20);
        add_last(&list1, 30);
        display(&list1);
}
```

13. 단순 연결 리스트에서 삭제함수 `delete` 함수는 사실은 헤드포인터와 선행노

드 포인터의 2개의 파라미터만 있으면 작성이 가능하다. 이들 두 파라미터만을 사용하여 다시 작성하라.

```
// phead : 헤드 포인터에 대한 포인터
// p: 삭제될 노드의 선행 노드
void remove_node( ListNode **phead, ListNode *p )
{
        ListNode *removed;
        if( p == NULL ){
                removed = (*phead);
                *phead = (*phead)->link;
        }
        else {
                removed=p->link;
                p->link = removed->link;
        }
        free(removed);
}
```

14. 단순 연결 리스트에 정수가 저장되어 있다. 단순 연결 리스트의 모든 데이터 값을 더한 합을 출력하는 프로그램을 작성하시오.

```
//
int sum( ListNode *head )
{
        ListNode *p=head;
        int sum=0;
        while( p != NULL ){
                sum += p->data;
                p = p->link;
     }
        return sum;
}
```

15. 단순 연결 리스트에서 특정한 데이터값을 갖는 노드의 개수를 계산하는 함수를 작성하라.

```
//
int count( ListNode *head, int x )
{
        ListNode *p=head;
```

```
        int count=0;
        while( p != NULL ){
                if( p->data == x ) count++;
                p = p->link;
    }
        return count;
}
```

16. 단순 연결 리스트에서의 탐색함수를 참고하여 특정한 데이터값을 갖는 노드를 삭제하는 함수를 작성하라.

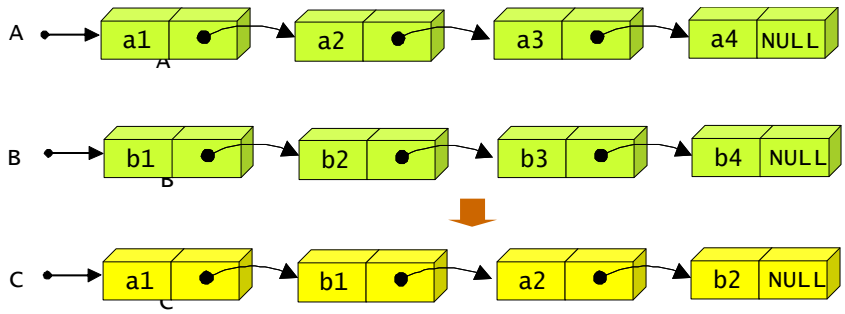
```
void search_remove(ListNode **phead, int x)
{
        ListNode *p, *prev=NULL;
        p = *phead;
        while( p != NULL ){
                if( p->data == x ) {
                        remove_node(phead, prev, p);
                        if( prev!= NULL ) p=prev->link;
                        else p=*phead;
                }
                else {
                        prev=p;
                        p = p->link;
                }
        }
}
```

17. 단순 연결 리스트의 헤드 포인터가 주어져 있을 때 첫 번째노드에서부터 하나씩 건너서 있는 노드를 전부 삭제하는 함수를 작성하라. 즉 홀수번째 있는 노드들이 전부 삭제된다.

```
// 홀수번째 노드 삭제
void remove_odd(ListNode **phead)
{
        ListNode *p, *prev=NULL;
        int count=0;
        p = *phead;
        while( p != NULL ){
                if( (count%2)!=0 ) {
                        remove_node(phead, prev, p);
```

```
                if( prev!= NULL ) p=prev->link;
                else p=*phead;
        }
        else {
                prev=p;
                p = p->link;
        }
        count++;
}
```

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




```
//
//
void insert_node_last(ListNode **phead, element item)
{
        ListNode *new_node = create_node(item, NULL);
        if( *phead == NULL ) *phead = new_node;
        else {
                ListNode *p = *phead;
                while(p->link != NULL) p = p->link;
                p->link = new_node;
        }
}

//
//
ListNode *alternate(ListNode *list1, ListNode *list2)
```

```
{
        ListNode *list3=NULL;
        ListNode *p1, *p2, *p3=NULL;
        if( list1 == NULL ) return list2;
        else if( list2 == NULL ) return list1;
        else {
                p1 = list1;
                p2 = list2;
                while(( p1 != NULL ) || ( p2 != NULL )){
                        if( p1 != NULL ) {
                                insert_node_last(&list3, p1->data);
                                p1=p1->link;
                        }
                        if( p2 != NULL ) {
                                insert_node_last(&list3, p2->data);
                                p2=p2->link;
                        }
                }
                return list3;
        }
}
```