자료구조 (Data Structure)
2주차: 링크드 리스트

오늘 목표

- 단방향 링크드 리스트 구현
- 다항식 구현
- 양방향 링크드 리스트 구현

필수 헤더

```
#include <stdio.h>

#include <stdlib.h>
```

단방향 노드

- 데이터와 다음 데이터 정보의 묶음
- next: 다음 노드 정보
- val: 저장된 데이터

단방향 노드

```
struct node {
        struct node *next;
        int val;
};
```

CREATE(value, next)

```
struct node *node_create(int value, struct node *next)
{
    struct node *node = malloc(sizeof(struct node));
    if (!node) exit(1);
    node->val = value;
    node->next = next;
    return node;
}
```

단방향 노드 관련 기능

- INSERT_AFTER(prev, node):
  prev 노드 뒤에 node를 추가한다

- DELETE_AFTER(prev):
  prev 다음 노드를 삭제한다

INSERT_AFTER(prev, node)

- prev의 다음 노드를 node로 만들고, 기존에 있던 prev의 다음 노드는 node 다음으로 이동한다
- 조건: prev와 node 존재
- 가정: node의 next는 비어있다
- 동작: node의 next에 prev의 next 저장
  prev의 next에 node 저장

INSERT_AFTER(prev, node)

```
void insert_after(struct node *prev, struct node *node)
{
    if (!prev || !node) exit(1);

    node->next = prev->next;
    prev->next = node;
}
```

DELETE_AFTER(prev)

- prev의 다음 노드를 삭제한다. 삭제할 노드의 다음 노드는 prev에 연결한다
- 조건: prev 존재, prev의 다음 노드 존재
- 동작: target에 prev의 next 저장
  prev의 next에 target의 next 저장
  target 삭제

DELETE_AFTER(prev)

```
void delete_after(struct node *prev)
{
    if (!prev || !prev->next) exit(1);

    struct node *target = prev->next;

    prev->next = target->next;
    free(target);
}
```

단방향 링크드 리스트

- 첫번째 노드만 알면 모든 노드 방문 가능

```
struct list {
        struct node *first;
};
```

단방향 리스트 관련 기능

- INSERT_FIRST(l, node):
  리스트 l의 첫번째 노드 자리에 node를 추가한다

- DELETE_FIRST(l):
  리스트 l의 첫번째 노드를 삭제한다

- RETRIEVE(l, k):
  리스트 l에서 k번째 노드를 반환한다.
  없으면 NULL을 반환한다

INSERT_FIRST(l, node)

- 리스트 l의 첫번째 노드 자리에 node를 추가한다
- 조건: l과 node 존재
- 가정: node의 next는 비어있다
- 동작: node의 next에 l의 first 저장
  l의 first에 node 저장

INSERT_FIRST(l, node)

```
void insert_first(struct list *l, struct node *node)
{
    if (!l || !node) exit(1);

    node->next = l->first;
    l->first = node;
}
```

DELETE_FIRST(l)

- 리스트 l의 첫번째 노드를 삭제한다
- 조건: l 존재, l의 first 존재
- 동작: target에 l의 first 저장

  l의 first에 l의 first의 next 저장

  target 삭제

DELETE_FIRST(l)

```
void delete_first(struct list *l)
{
        if (!l || !l->first) exit(1);

        struct node *target = l->first;
        l->first = l->first->next;

        free(target);
}
```

RETRIEVE(l, k)

- 링크드 리스트 l에서 k번째 노드를 반환한다. 없거나 에러 발생시 NULL을 반환한다.

- 조건: l 존재, k > 0
- 동작: result에 l->first 저장

    반복: k - 1번

        만약: result가 NULL이면 반복 종료

        그외: result에 result의 next 저장

    반환: result

RETRIEVE(l, k)

```
struct node *retrieve(struct list *l, int k)
{
    if (!l || k < 1) return NULL;
    struct node *result = l->first;
    for (int i = 1; i < k; ++i) {
        if (!result) break;
        result = result->next;
    }
    return result;
}
```

예제: 다항식 저장하기

- $2x^2 + 3x$를 저장하려면,
- 1. 항 노드를 만든다 (계수 3, 지수 1, 다음 없음)
- 2. 항 노드를 만든다 (계수 2, 지수 2, 다음 3x)

항 노드

- coeff: 계수
- exp: 지수
- next: 다음 항

항 노드

```
struct TermNode {
        int coeff;          // 계수
        int exp;            // 지수
        struct TermNode *next;  // 다음 항
};
```

TERM_CREATE(coeff_val, exp_val, nxt)

- 동작: 새로운 TermNode 메모리 할당
  node의 coeff에 coeff_val 저장
  node의 exp에 exp_val 저장
  node의 next에 nxt 저장

- 반환: node

TERM_CREATE(coeff_val, exp_val, nxt)

```
struct TermNode *term_create(int coeff_val, int
exp_val, struct TermNode *nxt)
{
    struct TermNode *node
        = malloc(sizeof(struct TermNode));
    node->coeff = coeff_val;
    node->exp = exp_val;
    node->next = nxt;
    return node;
}
```

INSERT_TAIL(tail, coeff_val, exp_val)

- tail의 다음 노드를 새로운 항 노드로 만든다
- 조건: tail 존재
- 가정: tail의 next는 비어있다
- 동작: tail의 next에 TERM_CREATE(coeff_val, exp_val, NULL) 저장

INSERT_TAIL(tail, coeff_val, exp_val)

```
void insert_tail(struct TermNode *tail, int coeff_val, int exp_val) {
    tail->next = term_create(coeff_val, exp_val, NULL);
}
```

ADD(X, Y) 1/5

- X, Y는 (exp 내림차순, 동차 지수 없음) 리스트로 가정

- 동작: 새로운 리스트 dummy 생성 (coeff 0, exp 0, next NULL)

  tail에 dummy 저장

  반복: X와 Y가 모두 NULL이 아닐 때까지

ADD(X, Y) 1/5

```
struct TermNode *add(struct TermNode *X, struct
TermNode *Y)
{
    struct TermNode dummy;
    struct TermNode *tail = &dummy;
    dummy.coeff = 0;
    dummy.exp = 0;
    dummy.next = NULL;
    while (X != NULL && Y != NULL) {
```

ADD(X, Y) 2/5

- 만약: X.exp > Y.exp

  INSERT_TAIL(tail, X.coeff, X.exp)

  X에 X.next를 저장

  tail에 tail.next를 저장

  다음 반복으로

ADD(X, Y) 2/5

```
if (X->exp > Y->exp) {
        insert_tail(tail, X->coeff, X->exp);
        X = X->next;
        tail = tail->next;
        continue;
}
```

ADD(X, Y) 3/5

- 만약: X.exp < Y.exp

  INSERT_TAIL(tail, Y.coeff, Y.exp)

  Y에 Y.next를 저장

  tail에 tail.next를 저장

  다음 반복으로

ADD(X, Y) 3/5

```
if (X->exp < Y->exp) {
        insert_tail(tail, Y->coeff, Y->exp);
        Y = Y->next;
        tail = tail->next;
        continue;
}
```

ADD(X, Y) 4/5

- sum에 X.coeff + Y.coeff 저장
- 만약: sum != 0

  INSERT_TAIL(tail, sum, Y.exp)

  tail에 tail.next 저장
- X에 X.next 저장
- Y에 Y.next 저장

ADD(X, Y) 4/5

```
        int sum = X->coeff + Y->coeff;
        if (sum != 0) {
                insert_tail(tail, sum, Y->exp);
                tail = tail->next;
        }
        X = X->next;
        Y = Y->next;
}
```

ADD(X, Y) 5/5

- 만약: X != NULL

  tail.next에 X 저장

- 그외:

  tail.next에 Y 저장

- 반환: dummy.next

ADD(X, Y) 5/5

```
    if (X != NULL) {
        tail->next = X;
    } else {
        tail->next = Y;
    }
    return dummy.next;
}
```

양방향 노드

- 데이터와 다음 데이터 정보, 이전 데이터 정보의 묶음
- val: 저장된 데이터
- next: 다음 노드 정보
- prev: 이전 노드 정보

양방향 노드

```
struct dnode {
        struct dnode *next;
        struct dnode *prev;
        int val;
};
```

DNODE_CREATE(value, prev, next)

```
struct dnode *dnode_create(int value, struct dnode
*prev, struct dnode *next)
{
    struct dnode *node = malloc(sizeof(struct dnode));
    if (!node) exit(1);
    node->val = value;
    node->next = next;
    node->prev = prev;
    return node;
}
```

양방향 노드 관련 기능

- DNODE_BETWEEN(prev, next, node):
  prev와 next 사이에 node를 추가한다

- DNODE_DELETE(node):
  node를 삭제한다

DNODE_BETWEEN(prev, next, node)

- prev와 next 사이에 node를 추가한다
- 조건: node 존재
- 가정: node의 next와 prev는 비어있다
- 동작: node의 next에 next 저장
  node의 prev에 prev 저장
  next의 prev에 node 저장
  prev의 next에 node 저장

DNODE_BETWEEN(prev, next, node) 1/2

```
void dnode_between(struct dnode *prev, struct
dnode *next, struct dnode *node)
{
    if (!node) exit(1);

    node->next = next;
    node->prev = prev;
```

DNODE_BETWEEN(prev, next, node) 2/2

```
        if (next)
                next->prev = node;
        if (prev)
                prev->next = node;
}
```

DNODE_DELETE(node)

- node를 삭제한다

- 조건: node 존재
- 동작: 만약 node의 prev가 존재하면

    node의 prev의 next에 node의 next 저장

  만약 node의 next가 존재하면

    node의 next의 prev에 node의 prev 저장

  node 삭제

DNODE_DELETE(node)

```
void dnode_delete(struct dnode *node)
{
        if (!node) exit(1);

        if (node->prev)
                node->prev->next = node->next;
        if (node->next)
                node->next->prev = node->prev;
        free(node);
}
```

양방향 링크드 리스트

- 첫번째 노드로부터 순방향으로 모든 노드 방문 가능
- 마지막 노드로부터 역방향으로 모든 노드 방문 가능
- 노드 개수 저장

- first: 첫번째 노드 정보
- last: 마지막 노드 정보
- size: 노드 개수

양방향 링크드 리스트

```
struct dlist {
        struct dnode *first;
        struct dnode *last;
        int size;
};
```

DLIST_CREATE()

- 양방향 링크드 리스트를 생성한다

- 동작: 새로운 dlist 메모리 할당

  first와 last를 NULL로 초기화

  size를 0으로 초기화

  새로 생성한 리스트 반환

DLIST_CREATE()

```
struct dlist *dlist_create() {
    struct dlist *new_list = malloc(sizeof(struct dlist));
    new_list->first = NULL;
    new_list->last = NULL;
    new_list->size = 0;
    return new_list;
}
```

양방향 링크드 리스트 관련 기능

- DLIST_INSERT_FIRST(l, node):
  리스트 l의 첫번째 자리에 node를 추가한다

- DLIST_INSERT_BEFORE(l, next, node):
  리스트 l의 next 앞에 node를 추가한다

- DLIST_DELETE(l, node):
  리스트 l의 노드를 삭제한다

DLIST_INSERT_FIRST(l, node)

- 양방향 리스트 l의 첫번째 자리에 node를 추가한다
- 조건: l과 node 존재
- 가정: node의 next와 prev는 비어있다
- 동작: DNODE_BETWEEN(NULL, l의 first, node)

  l의 first에 node 저장

  만약: l의 last가 NULL이면

  l의 last에 node 저장

  l의 size 1 증가

DLIST_INSERT_FIRST(l, node)

```
void dlist_insert_first(struct dlist *l, struct dnode
*node)
{
        if (!l || !node) exit(1);
        dnode_between(NULL, l->first, node);
        l->first = node;
        if (!l->last)
                l->last = node;
        l->size++;
}
```

DLIST_INSERT_BEFORE(l, next, node)

- l의 next 앞에 node를 추가한다
- 조건: l, next, node 존재
- 가정: node의 next와 prev는 비어있다
- 동작: 만약: next가 l의 first이면

  DLIST_INSERT_FIRST(l, node)

  그외:

  DNODE_BETWEEN(next의 prev, next, node)

  l의 size 1 증가

DLIST_INSERT_BEFORE(l, next, node) 1/2

```
void dlist_insert_before(struct dlist *l, struct dnode
*next, struct dnode *node)
{
    if (!l || !next || !node) exit(1);

    if (next == l->first) {
        dlist_insert_first(l, node);
    }
```

DLIST_INSERT_BEFORE(l, next, node) 2/2

```
        else {
            dnode_between(next->prev, next, node);
            l->size++;
        }
    }
```

DLIST_DELETE(l, node)

- 리스트 l의 노드를 삭제한다
- 조건: l과 node 존재, l의 size > 0
- 동작: 만약: node가 l의 first이면

    l의 first에 node의 next 저장

  만약: node가 l의 last이면

    l의 last에 node의 prev 저장

  DNODE_DELETE(node)

  l의 size 1 감소

DLIST_DELETE(l, node)

```
void dlist_delete(struct dlist *l, struct dnode *node)
{
        if (!l || !node || l->size <= 0) exit(1);
        if (node == l->first)
                l->first = node->next;
        if (node == l->last)
                l->last = node->prev;
        dnode_delete(node);
        l->size--;
}
```

요약

- 단방향 링크드 리스트에서 데이터 추가/삭제
- 양방향 링크드 리스트에서 데이터 추가/삭제
- 다항식 예제
- 다음 시간: 스택

C 복습 - 구조체.c

```
#include <stdio.h>

struct Point {
	int x;
	int y;
};
```

C 복습 - 구조체.c

```
void struct_test()
{
	struct Point p;
	p.x = 1;
	p.y = 2;
	printf("(%d, %d)\n", p.x, p.y);
}
```

- struct_test()를 실행했을 때 화면에 출력되는 문자열은?

C 복습 - 구조체_포인터.c

```
#include <stdio.h>

struct Point {
	int x;
	int y;
};
```

C 복습 - 구조체_포인터.c

```
void struct_ptr_test()
{
	struct Point p;
	struct Point *p_ptr = &p;
	p_ptr->x = 1;
	p_ptr->y = 2;
	printf("(%d, %d)\n", p_ptr->x, p_ptr->y);
}
```

- struct_ptr_test()를 실행했을 때 화면에 출력되는 문자열은?

C 복습 - 구조체_포인터_파라미터.c

```
#include <stdio.h>

struct Point {
	int x;
	int y;
};
```

C 복습 - 구조체_포인터_파라미터.c

```
void struct_set_members(struct Point *p_ptr)
{
	p_ptr->x = 1;
	p_ptr->y = 2;
}
```

C 복습 - 구조체_포인터_파라미터.c

```
void struct_ptr_parameter_test()
{
	struct Point p;
	struct_set_members(&p);
	printf("(%d, %d)\n", p.x, p.y);
}
```

struct_ptr_parameter_test()를 실행했을 때 화면에 출력되는 문자열은?

C 복습 - 메모리_할당_해제.c

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

struct Point {
	int x;
	int y;
};
```

C 복습 - 메모리_할당_해제.c

```
struct Point *create_point()
{
    return malloc(sizeof(struct Point));
}

void free_point(struct Point *p_ptr)
{
    free(p_ptr);
}
```

C 복습 - 메모리_할당_해제.c

```
void struct_set_members(struct Point *p_ptr)
{
    p_ptr->x = 1;
    p_ptr->y = 2;
}
```

C 복습 - 메모리_할당_해제.c

```
void struct_ptr_malloc_test()
{
    struct Point* p_ptr = create_point();
    if (!p_ptr)
        return;
    struct_set_members(p_ptr);
    printf("(%d, %d)\n", p_ptr->x, p_ptr->y);
    free_point(p_ptr);
}
```

- struct_ptr_malloc_test() 실행 후 화면 출력은?

C 코딩 스타일 - if-문의 중괄호

- if와 else에서 모두 문장 하나만 실행할 때는 중괄호를 쓰지 않는다.

```
if (condition)
    do_this();
else
    do_that();
```

C 코딩 스타일 - if-문의 중괄호 (나쁜 예)

- if의 문장을 숨길 것이 아닌 한 한줄에 쓰지 않는다.

```
if (condition) do_this;
do_something_everytime;
```

C 코딩 스타일 - if-문의 중괄호 (나쁜 예)

- 괄호를 쓰지 않기 위해서 쉼표를 쓰지 않는다.

```
if (condition)
        do_this(), do_that();
```

C 코딩 스타일 - if-문의 중괄호

- if나 else 중 하나에만 문장이 두개 이상 있어도 모두 중괄호를 쓴다.

```
if (condition) {
        do_this();
        do_that();
} else {
        otherwise();
}
```

C 코딩 스타일 - 반복문의 중괄호

- 반복문은 단순 문장 한개만 있지 않은 한 항상 중괄호를 쓴다.

```
while (condition) {
    if (test)
        do_something();
}
```

C 코딩 스타일 - 띄어쓰기 - 키워드

- 다음 키워드 다음에는 띄어쓰기를 한칸 한다.

  if, switch, case, for, do, while

- sizeof 다음에는 띄어쓰기를 하지 않는다.

  좋은 예: s = sizeof(struct file);

- 괄호 안쪽 주변에는 띄어쓰기를 하지 않는다.

  나쁜 예: s = sizeof(struct file);

C 코딩 스타일 - 띄어쓰기 - 포인터

- 포인터 변수를 선언하거나, 포인터를 반환하는 함수를 선언할 때,
  - * 은 변수 이름이나 함수 이름에 붙인다.
  - * 을 타입 옆에 붙이지 않는다.

```
char *linux_banner;

unsigned long long memparse(char *ptr,
                            char **retptr);

char *match_strdup(substring_t *s);
```

C 코딩 스타일 - 띄어쓰기 - 이항연산자

- 다음 이항 연산자 / 삼항 연산자 주변에는 띄어쓰기를 한칸 한다.

  = + - * / % < > <= >= == != & | ? :

- 다음 구조체 멤버 연산자 주변에는 띄어쓰기를 하지 않는다.

  . ->

- area = rect.x * rect.y;

C 코딩 스타일 - 띄어쓰기 - 단항연산자

- 다음 단항 연산자 뒤에는 띄어쓰기를 하지 않는다.

  & * + - ~ ! sizeof

- 다음 postfix 증감 단항 연산자 주변에는 띄어쓰기를 하지 않는다.

  ++ --

- 다음 prefix 증감 단항 연산자 주변에는 띄어쓰기를 하지 않는다.

  ++ --

C 코딩 스타일 - 이름

- C 언어는 단순하고 미니멀한 것을 추구한다.
- 어떤 프로그래밍 언어에서는 ThisVariableIsATemporaryCounter로 쓰지만
- C 언어에서는 쓰기 훨씬 쉽고, 이해하기 많이 어렵지 않은 tmp로 쓴다.
- 한편, 전역 범위에 있는 이름은 자세해야 한다.

C 코딩 스타일 - 전역 변수 이름, 함수 이름

- 전역 변수는 다른 방법이 전혀 없어서 어쩔 수 없이 꼭 필요할 때만 쓴다.
- 전역 변수와 함수의 이름은 단어들을 _로 구분해서 자세하게 만든다.
- 좋은 예: count_active_users()
- 나쁜 예: cntusr()

C 코딩 스타일 - 지역 변수 이름

- 지역 변수 이름은 짧고, 정보를 담고 있어야 한다.
- 좋은 예: 루프 변수 i
- 나쁜 예: 루프 변수 loop_counter (명확한 상황에서)
- 좋은 예: 임시 변수 tmp

C 코딩 스타일 - 함수

- 함수는 짧고 한가지 일만 해야 한다.
- 함수에서 지역 변수를 10개 이하로 써야 한다.
- 함수 정의들 간에는 한줄을 띄워서 쓴다.
- 함수 선언에서 타입과 이름을 모두 쓴다.

C 코딩 스타일 - 함수 관련 goto

- goto는 많은 프로그래머들이 더이상 쓰지 않는다.

- 여러 군데에서 함수가 종료되는데 공통된 정리 코드가 있을 때 유용하다.

```
int fun(int a)
{
	int result = 0;
	char *buffer;
```

C 코딩 스타일 - 함수 관련 goto

```
buffer = malloc(SIZE);
if (!buffer)
        return -ENOMEM;

if (condition1) {
        while (loop1) {
                ...
        }
        result = 1;
        goto out_free_buffer;
}
...
```

C 코딩 스타일 - 함수 관련 goto

```
out_free_buffer:
        free(buffer);
        return result;
}
```

C 코딩 스타일 - 함수 관련 goto 에러

- one err bugs로 알려진 흔한 버그

```
err:
        free(foo->bar);
        free(foo);
        return ret;
```

- foo가 NULL인 상태에서 err로 이동했을 때 버그 발생

C 코딩 스타일 - 함수 관련 goto 에러 해결책

- err_free_bar:와 err_free_foo:로 라벨을 분리한다.

```
err_free_bar:
        free(foo->bar);
err_free_foo:
        free(foo);
        return ret;
```

- foo가 NULL이 아닐 때만 err_free_bar로 이동

C 코딩 스타일 - 주석

- 주석은 함수 앞에서 무엇을 왜 하는지 설명한다.
- 아주 복잡한 함수의 경우 본문 안에도 구획을 나눠서 주석을 달 수 있다.

```
/*
 * This is the preferred style for multi-line
 * comments in the Linux kernel source code.
 */
```

- 데이터에 대해 짧은 주석을 다는 것도 중요하다.

C 복습 - 이름의 범위

```
int x = 0;
void function(void)
{
    int a = x;        /* 전역 x(=0) */
    int x = 1;        /* 함수 지역 x */
    for (int x = 2; x < 10; x++) {
        int b = x;  /* for-블록의 x */
    }
    a = x;          /* 여기서의 x는 함수 지역 x(=1) */
} /* 함수가 끝나기 전 a에 저장된 값은? */
```

C 복습 - 배열.c

- #include <stdio.h>

```
void array_test()
{
	int arr[3];

	arr[0] = 1;
	arr[1] = 2;
	arr[2] = 3;
```

C 복습 - 배열.c

```
    for (int i = 0; i < 3; i++)
        printf("%d ", arr[i]);

    printf("\n");
}
```

- array_test()를 실행했을 때 화면에 출력되는 문자열은?

C 복습 - 배열_포인터.c

```
#include <stdio.h>


void array_ptr_test()
{
	int arr[3];
	int *arr_ptr = arr;


	arr_ptr[0] = 1;
	arr_ptr[1] = 2;
	arr_ptr[2] = 3;
```

C 복습 - 배열_포인터.c

```
    for (int i = 0; i < 3; i++)
        printf("%d ", arr_ptr[i]);

    printf("\n");
}
```

- array_ptr_test()를 실행했을 때 화면에 출력되는 문자열은?

C 복습 - 배열_포인터_파라미터.c

```
#include <stdio.h>


void array_set_elements(int *arr_ptr)
{
	arr_ptr[0] = 1;
	arr_ptr[1] = 2;
	arr_ptr[2] = 3;
}
```

C 복습 - 배열_포인터_파라미터.c

```
void array_fun_test()
{
	int arr[3];
	array_set_elements(arr);
	for (int i = 0; i < 3; i++)
		printf("%d ", arr[i]);

	printf("\n");
}
```

- array_fun_test()를 실행했을 때 화면 출력은?

C 키워드 _Bool

- 0 또는 1을 저장할 수 있는 타입
- 거짓 또는 참을 저장하기 위해 쓰임