자료구조 (Data Structure)
8주차: 그래프

지난 시간 요약

- 그래프: 노드와 연결선을 모은 것
- 깊이 우선 방문: 한 이웃의 이웃들을 먼저 방문
  - 그룹핑
  - 단절점 찾기

오늘의 목차

- 그래프 출력하기
- 깊이 우선 탐색 구현하기
- 그룹핑 하기
- 단절점 찾기

그래프 출력 - printf

```
#include <stdio.h>
int main()
{
        printf("Hello, graph!");
        return 0;
}
```

그래프 출력 - 인접리스트 구조체

```
struct edge;

struct elist {
        struct edge *edge;
        struct elist *next;
}
```

그래프 출력 - 인접리스트 구조체

- 링크드 리스트 형태
  - 데이터: 연결선의 주소
  - 링크: 다음 인접리스트 노드의 주소

그래프 출력 - 그래프 노드 구조체

```
struct node {
        char data;
        int index;
        int status;             /* 0: 방문전, 1: 도중, 2: 후 */
        struct elist *adj;
}
```

그래프 출력 - 그래프 노드 구조체

- 그래프 노드
  - 데이터: 노드를 표시할 문자
  - 인덱스: 그래프의 노드 리스트에서 몇번째
  - 상태: 깊이 우선 탐색 진행 상황
  - 이웃들: 연결선 정보

그래프 출력 - 그래프 노드 구조체 확인

```
void print_node(struct node *v)
{
        char str[4] = {' ', v->data, ' ', '\0'};
        if (v->status != 0)
                str[0] = '(';
        if (v->status == 2)
                str[2] = ')';

        printf("%s", str);
}
```

그래프 출력 - 그래프 노드 구조체 확인

```
int main()
{
        struct node v = {'A'};
        print_node(&v);
        v.status = 1;
        print_node(&v);
        v.status = 2;
        print_node(&v);
        return 0;
}
```

그래프 출력 - 그래프 노드 구조체 확인

- 구조체 선언
  - 초기값을 { } 안에 순서대로 적어준다.
  - { } 안에 없는 값은 0이 된다.

- 멤버
  - 구조체에서 .으로 멤버에 접근한다.
  - 구조체 주소에서 ->로 멤버에 접근한다.

그래프 출력 - 연결선 구조체

```
struct edge {
        struct node *node1;
        struct node *node2;
        char shape;                 /* '-', '/', '|' */
        int dfs_type;               /* 0: 방문전 */
                                    /* 1: 트리 선 */
                                    /* 2: 역방향 선 */
}
```

그래프 출력 - 연결선 구조체

- 연결선
  - 노드1: 연결선의 한쪽 노드
  - 노드2: 연결선의 다른쪽 노드
  - 모양: 화면에 표시할 모양. 선의 방향.
  - dfs 타입: 깊이우선탐색에서의 타입.

그래프 출력 - 연결선 구조체 확인

```
void print_edge(struct edge *e)
{
        char str[4] = "   ";
        if (e != NULL) {
                if (e->dfs_type == 0) {
                        str[1] = e->shape;
                } else if (e->dfs_type == 2) {
                        str[1] = ':';
                }
```

그래프 출력 - 연결선 구조체 확인

```
            else if (e->dfs_type == 1) {
                    if (e->shape == '-')
                            str[1] = '=';
                    else
                            str[1] = str[2] = e->shape;
            }
    }
    printf("%s", str);
}
```

그래프 출력 - 연결선 구조체 확인

```
int main()
{
        struct edge e = {NULL, NULL, '-', 0};
        print_edge(&e);
        e.dfs_type = 1;
        print_edge(&e);
        e.dfs_type = 2;
        print_edge(&e);
        return 0;
}
```

그래프 출력 - 연결선 구조체 확인

- 연결선을 출력할 때,
  - 트리선은 두줄로 출력한다.
  - 역방향 선은 .으로 출력한다.

그래프 출력 - 그래프 구조체

```
struct graph {
        struct node *vertexes;
        int n_vertexes;
        struct node *edges;
        int n_edges;
        struct edge ***adj;
}
```

그래프 출력 - 그래프 구조체

- 그래프
  - V: 노드 배열의 주소
  - n_V: 노드 개수
  - E: 연결선 배열의 주소
  - E_V: 연결선 개수
  - adj: 노드 인덱스 두개에 대한 연결선 이차원 배열

그래프 출력 - 그래프 구조체 확인

```
void print_graph(struct graph *g)
{
        /* 점 출력 */
        for (int i = 0; i < g->n_vertexes; i++) {
                print_node(g->vertexes + i);
                printf(", ");
        }
        printf("\n");
```

그래프 출력 - 그래프 구조체 확인

```
    /* 연결선 출력 */
    for (int i = 0; i < g->n_edges; i++) {
            printf("[");
            print_node(g->edges[i].node1);
            print_edge(g->edges + i);
            print_node(g->edges[i].node2);
            printf("], ");
    }
    printf("\n");
}
```

그래프 출력 - 그래프 구조체 확인

```
int main()
{
        struct graph g;
        struct node nodes[2] = {
                {.data = 'A'},
                {.data = 'B'},
        };
        g.vertexes = nodes;
        g.n_vertexes = 2;
```

그래프 출력 - 그래프 구조체 확인

```
    struct edge edges[1] = {
            {.node1 = g.vertexes + 0,
             .node2 = g.vertexes + 1,
             .shape = '-'}
    };
    g.edges = edges;
    g.n_edges = 1;
    print_graph(&g);
    return 0;
}
```

그래프 출력 - 그래프 구조체 확인

- 구조체 선언
  - 초기값을 { } 안에 적을 때, .멤버 = 형태로 특정 멤버의 초기값을 정한다.

- 배열
  - 배열은 계산할 때 0번째 멤버의 주소가 된다
  - i 번째 멤버의 주소는 배열 + i로 계산한다

그래프 출력 - 인접 행렬

```
void make_adj_matrix(struct graph *g)
{
        for (int i = 0; i < g->n_edges; i++) {
                struct edge *e = g->edges + i;
                int n1 = e->node1->index;
                int n2 = e->node2->index;
                g->adj[n1][n2] = g->adj[n2][n1] = e;
        }
}
```

그래프 출력 - 인접 행렬

```
void print_adj_matrix(struct graph *g)
{
        /* 첫번째 행: 노드 출력 */
        printf("  ");
        for (int i = 0; i < g->n_vertexes; i++) {
                printf("%c ", g->vertexes[i].data)
        }
        printf("\n");
        for (int i = 0; i < g->n_vertexes; i++) {
                /* 첫번째 열: 노드 출력 */
```

그래프 출력 - 인접 행렬

```
            printf("%c ", g->vertexes[i].data)
            for (int j = 0; j < g->n_vertexes; j++) {
                if (g->adj[i][j] == NULL)
                    printf("  ");
                else
                    printf("%c ", g->adj[i][j]->shape);
            }
            printf("\n");
        }
}
```

그래프 출력 - 인접 행렬

```
int main() {
        struct node n[5] = {
                {'A', 0},
                {'B', 1},
                {'C', 2},
                {'D', 3},
                {'E', 4}
        };
        g.vertexes = n;
        g.n_vertexes = 5;
```

그래프 출력 - 인접 행렬

```
struct edge edges2[] = {
        {n + 0, n + 1, '-'},
        {n + 1, n + 2, '-'},
        {n + 0, n + 3, '|'},
        {n + 1, n + 3, '/'},
        {n + 1, n + 4, '|'},
        {n + 2, n + 4, '/'}
};
g.edges = edges2;
g.n_edges = 6;
```

그래프 출력 - 인접 행렬

```
struct edge *adj_matrix[6][6] = {0};
struct edge **adj_rows[6] = {
        adj_matrix[0],
        adj_matrix[1],
        adj_matrix[2],
        adj_matrix[3],
        adj_matrix[4],
        adj_matrix[5]
};
g.adj = adj_rows;
```

그래프 출력 - 인접 행렬

```
    make_adj_matrix(&g);
    print_adj_matrix(&g);
    return 0;
}
```

그래프 출력 - 인접 행렬

- 배열 선언
  - 배열의 개수를 정하지 않고 선언하면,
    초기값의 개수에 따라 크기가 정해진다.

그래프 출력 - 인접 리스트

```
void make_adj_list(struct graph *g)
{
        for (int i = 0; i < g->n_vertexes; i++) {
                g->vertexes[i].adj = NULL;

                for (int j = g->n_vertexes - 1; j >= 0; j--) {
                        if (g->adj[i][j] == NULL)
                                continue;
```

그래프 출력 - 인접 리스트

```
                struct elist *l
                        = malloc(sizeof(struct elist));
                l->edge = g->adj[i][j];
                l->next = g->vertexes[i].adj;
                g->vertexes[i].adj = l;
            }
        }
}
```

그래프 출력 - 인접 리스트

```
void free_adj_list(struct graph *g)
{
        for (int i = 0; i < g->n_vertexes; i++) {
                while (g->vertexes[i].adj != NULL) {
                        struct elist *l = g->vertexes[i].adj;
                        g->vertexes[i].adj = l->next;
                        free(l);
                }
        }
}
```

그래프 출력 - 인접 리스트

```
void print_adj_list(struct graph *g)
{
        for (int i = 0; i < g->n_vertexes; i++) {
                struct node *n = g->vertexes + i;
                printf("%c: ", n->data);

                struct elist *e = n->adj;
                while (e != NULL) {
```

그래프 출력 - 인접 리스트

```
                struct node *near = e->edge->node1;
                if (near == n) {
                        near = e->edge->node2;
                }
                printf("%c ", near->data);
                e = e->next;
        }
        printf("\n");
    }
}
```

그래프 출력 - 인접 리스트

```
int main() {
        /* 이전 내용에 이어서 */
        make_adj_list(&g);
        print_adj_list(&g);

        free_adj_list(&g);
        return 0;
}
```

그래프 출력 - 인접 리스트

- 링크드 리스트에 추가하는 순서
  - head에 추가를 하기 때문에, 마지막부터 역순으로 추가한다.

그래프 출력 - 3열 그래프

```
void print_graph_3col(struct graph *g)
{
        int n_vertexes = g->n_vertexes;

        /* 첫번째 행 */
        for (int i = 0; i < 3 && i < n_vertexes; i++) {
                print_node(g->vertexes + i);
                print_edge(g->adj[i][i + 1]);
        }
        printf("\n");
```

그래프 출력 - 3열 그래프

```
for (int row = 1; row <= n_vertexes / 3; row++) {
        int i;
        for (i = row * 3; i < row * 3 + 2
                        && i < n_vertexes; i++) {
                print_edge(g->adj[i][i - 3]);
                print_edge(g->adj[i][i - 2]);
        }
        if (i < n_vertexes)
                print_edge(g->adj[i][i - 3]);
        printf("\n");
```

그래프 출력 - 3열 그래프

```
            for (i = row * 3; i <= row * 3 + 2
                              && i < n_vertexes; i++) {
                    print_node(g->vertexes + i);
                    print_edge(g->adj[i][i + 1]);
            }
            printf("\n");
        }
}
```

그래프 출력 - 3열 그래프

```
int main() {
        /* 이전 내용에 이어서 */
        make_adj_list(&g);
        print_adj_list(&g);

        print_graph_3col(&g);
        free_adj_list(&g);
        return 0;
}
```

그래프 출력 - 3열 그래프

- 3열 그래프에서 출력되는 연결선
  - 같은 행의 인접 노드간 연결선
  - 같은 열의 인접 노드간 연결선
  - 오른쪽 위 노드와 연결선

깊이 우선 탐색 구현

```
void dfs_prep(struct graph *g)
{
        for (int i = 0; i < g->n_vertexes; i++) {
                g->vertexes[i].status = 0;
        }

        for (int j = 0; j < g->n_edges; j++) {
                g->edges[j].dfs_type = 0;
        }
}
```

깊이 우선 탐색 구현

```
void dfs(struct graph *g, struct node *n, struct node *p)
{
        if (n == NULL || n->status != 0) return;

        n->status = 1;
        print_graph_3col(g);

        struct elist *e = n->adj;
        for ( ; e != NULL; e = e->next) {
```

깊이 우선 탐색 구현

```
struct node *near = e->edge->node1;
if (near == n)
        near = e->edge->node2;

if (near == p || e->edge->dfs_type != 0)
        continue;

if (near->status != 0 && near->status != 1) {
        printf("Error\n");
}
```

깊이 우선 탐색 구현

```
        else if (near->status == 0) {
                e->edge->dfs_type = 1;
                print_graph_3col(g)
                dfs(g, near, n);
        }
         else if (near->status == 1) {
                e->edge->dfs_type = 2;
                print_graph_3col(g);
        }
}
```

깊이 우선 탐색 구현

```
        n_status = 2;
        print_graph_3col(g);
}
```

깊이 우선 탐색 구현

```
int main() {
        /* 이전 내용에 이어서 */
        make_adj_list(&g);

        dfs_prep(&g);
        dfs(&g, g.vertexes, NULL);

        free_adj_list(&g);
        return 0;
}
```

깊이 우선 탐색 구현

- 깊이 우선 탐색
  - 한번 체크한 연결선은 다시 체크하지 않는다.
  - 방문중인 노드와 연결선은 역방향 선이다
  - 방문전인 노드와 연결선은 트리선이다

그룹핑 하기

```
struct node {
        char data;
        int index;
        int status;             /* 0: 방문전, 1: 도중, 2: 후 */
        struct elist *adj;
        int group;
}
```

그룹핑 하기

```
void print_node(struct node *v)
{
        char str[5] = {' ', v->data, ' ', ' ', '\0'};
        if (v->status != 0)
                str[0] = '(';
        if (v->status == 2)
                str[3] = ')';
        if (v->group > 0) str[2] = '0' + v ->group;
        printf("%s", str);
}
```

그룹핑 하기

```
void print_edge(struct edge *e)
{
        char str[5] = "    ";
        if (e != NULL) {
                if (e->dfs_type == 0) {
                        str[1] = e->shape;
                } else if (e->dfs_type == 2) {
                        str[1] = ':';
                }
```

그룹핑 하기

```
int group = 0;
void dfs(struct graph *g, struct node *n, struct node *p)
{
        if (n == NULL || n->status != 0) return;
        n->status = 1;
        n->group = group;
        print_graph_3col(g);

        struct elist *e = n->adj;
        for ( ; e != NULL; e = e->next) {
```

그룹핑 하기

```
int main() {
        /* 이전 내용에 이어서 */

        dfs_prep(&g);
        for (int i = 0; i < g.n_vertexes; i++) {
                if (g.vertexes[i].group == 0) {
                        group += 1;
                        dfs(&g, g.vertexes, NULL);
                }
        }
```

그룹핑 하기

- 노드 속성에 그룹을 추가한다.
- 노드 출력시 그룹도 출력한다.
- 연결선 출력 너비를 노드에 맞춰 조정한다.
- dfs에서 그룹을 설정한다.
- dfs를 처음 하기 전에 전역변수 그룹을 설정한다.

단절점 찾기

```
struct node {
        char data;
        int index;
        int status;             /* 0: 방문전, 1: 도중, 2: 후 */
        struct elist *adj;
        int dfs_time;
        int min_back;
        int is_ap;
}
```

단절점 찾기

```
void print_node(struct node *v)
{
        char str[5] = {' ', v->data, ' ', ' ', '\0'};
        if (v->status != 0)
                str[0] = '(';
        if (v->status == 2)
                str[3] = ')';
        if (v->is_ap != 0) str[2] = 'a';
        printf("%s", str);
}
```

단절점 찾기

```
void print_edge(struct edge *e)
{
        char str[5] = "    ";
        if (e != NULL) {
                if (e->dfs_type == 0) {
                        str[1] = e->shape;
                } else if (e->dfs_type == 2) {
                        str[1] = ':';
                }
```

단절점 찾기

```
int time = 0;
void dfs(struct graph *g, struct node *n, struct node *p)
{
        if (n == NULL || n->status != 0) return;
        int n_children = 0;
        time += 1;
        n->status = 1;
        n->min_back = n->dfs_time = time;
        print_graph_3col(g);
        struct elist *e = n->adj;
```

단절점 찾기

```
else if (near->status == 0) {
        n_children += 1;
        e->edge->dfs_type = 1;
        print_graph_3col(g)
        dfs(g, near, n);
        if (near->min_back < n->min_back)
            n->min_back = near->min_back;
        if (n->min_back >= n->dfs_time)
            n->is_ap = 1;
}
```

단절점 찾기

```
        else if (near->status == 1) {
                e->edge->dfs_type = 2;
                if (near->dfs_time < n->min_back)
                    n->min_back = near->dfs_time;
                print_graph_3col(g);
            }
    }
```

단절점 찾기

```
        n_status = 2;

        if (p == NULL && n_children > 1)
                n->is_ap = 1;

        print_graph_3col(g);
}
```

단절점 찾기

```
int main() {
        /* 이전 내용에 이어서 */

        dfs_prep(&g);
        for (int i = 0; i < g.n_vertexes; i++) {
                if (g.vertexes[i].group == 0) {
                        dfs(&g, g.vertexes, NULL);
                }
        }
```

요약

- 그래프 출력하기
- 깊이 우선 탐색 구현하기
- 그룹핑 하기
- 단절점 찾기