자료구조 (Data Structure)
9주차: 그래프 최단경로

지난 시간 내용

- 최단경로: 시작 노드에서 가장 낮은 비용이 드는 경로
- 너비우선방문: 가장 가까운 노드에 우선순위 부여
  - 일반 큐: 연결선의 비용이 모두 1일 때
  - 양방향 큐: 연결선의 비용이 0 또는 1일 때
  - 우선순위 큐: 연결선의 비용이 0 이상일 때

오늘의 목차

- 그래프 출력하기
- 너비 우선 탐색 (BFS) 구현하기
- 양방향 큐 - 최단경로 탐색 구현하기
- 우선순위 큐 - 최단경로 탐색 구현하기

그래프 출력 - printf

```
#include <stdio.h>
int main()
{
        printf("Hello, shortest path!\n");
        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: 처리전, 1: 처리후 */
        int distance;
        struct node* parent;
        struct elist *adj;
};
```

그래프 출력 - 그래프 노드 구조체

- 그래프 노드
  - 데이터: 노드를 표시할 문자
  - 인덱스: 그래프의 노드 리스트에서 몇번째
  - 상태: 최단 경로 탐색 진행 상황
  - 거리: 현재까지 최단 거리
  - 이전노드: 최단거리 경로 상에서 이전 노드
  - 이웃들: 연결선 정보

그래프 출력 - 그래프 노드 구조체 확인

```
void print_node(struct node *v)
{
        char str[4] = "   ";
        str[0] = v->data;
        str[1] = '0' + v->distance;
        if (v->status != 0)
                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.distance = 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 length;                 /* 연결선의 길이 */
};
```

그래프 출력 - 연결선 구조체

- 연결선
  - 노드1: 연결선의 한쪽 노드
  - 노드2: 연결선의 다른쪽 노드
  - 모양: 화면에 표시할 모양. 선의 방향.
  - 길이: 거리 계산시 쓰이는 길이

그래프 출력 - 연결선 구조체 확인

```
void print_edge(struct edge *e)
{
        char str[4] = "   ";
        if (e != NULL) {
                str[0] = '0' + e->length;
                str[1] = e->shape;
        }
        printf("%s", str);
}
```

그래프 출력 - 연결선 구조체 확인

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

그래프 출력 - 연결선 구조체 확인

- 연결선을 출력할 때,
  - 길이도 출력한다.

그래프 출력 - 그래프 구조체

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

그래프 출력 - 그래프 구조체 확인

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

그래프 출력 - 그래프 구조체 확인

```
    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() {
        /* 이전 내용에 이어서 */
        struct edge *mat[6][6] = { 0 };
        struct edge **rows[6] = {mat[0], mat[1], mat[2],
                mat[3], mat[4], mat[5]};
        g.adj = rows;
        make_adj_matrix(&g);
        print_adj_matrix(&g);
        return 0;
}
```

그래프 출력 - 인접 행렬

- 인접 행렬
  - 인접 행렬은 두 노드에 대한 연결선 주소를 저장한 행렬이다

그래프 출력 - 인접 리스트

```
struct elist *new_elist(struct edge *e, struct elist *n)
{
        struct elist *el = malloc(sizeof(struct elist));
        if (el == NULL) return NULL;
        el->edge = e;
        el->next = n;
        return el;
}
```

그래프 출력 - 인접 리스트

```
void make_adj_list(struct graph *g)
{
        for (int i = g->n_edges - 1; i >= 0; i--) {
                struct edge *e = g->edges + i;
                struct node *u = e->node1;
                struct node *v = e->node2;
                struct elist *eu = new_elist(e, u->adj);
                if (eu != NULL)
                        u->adj = eu;
```

그래프 출력 - 인접 리스트

```
            struct elist *ev = new_elist(e, v->adj);
            if (ev != NULL)
                    v->adj = ev;
        }
}
```

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

- 인접 리스트
  - 인접 리스트는 노드 마다 가지고 있는 그 노드에 연결된 연결선의 리스트다

- 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 */

        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

너비 우선 탐색 구현

```
/* 큐 구조체와 함수 */
struct queue;
struct queue *new_queue(int size);
void enqueue(struct queue *q, struct node *n);
struct node *dequeue(struct queue *q);
int is_empty(struct queue *q);
void delete_queue(struct queue *q);
```

너비 우선 탐색 구현

```
void bfs(struct graph *g, struct node *s)
{
        for (int i = 0; i < g->n_vertexes; i++) {
                struct node *u = g->vertexes + i;
                u->distance = 9;
                u->parent = NULL;
                u->status = 0;
        }
        struct queue *q = new_queue(g->n_vertexes);
        print_graph_3col(g);
```

너비 우선 탐색 구현

```
s->distance = 0;
s->status = 1;
enqueue(q, s);
print_graph_3col(g);

while (!is_empty(q)) {
        struct node *u = dequeue(q);
        struct elist *near = u->adj;
```

너비 우선 탐색 구현

```
while (near != NULL) {

        struct node *v;

        if (near->edge->node1 == u)
                v = near->edge->node2;
        else
                v = near->edge->node1;
```

너비 우선 탐색 구현

```
            if (v->status == 0) {
                    v->parent = u;
                    v->distance = u->distance + 1;
                    v->status = 1;
                    enqueue(q, v);
                    print_graph_3col(g);
            }
            near = near->next;
}
```

너비 우선 탐색 구현

```
        }
        delete_queue(q);
}
```

너비 우선 탐색 구현

```
int main() {
        /* 이전 내용에 이어서 */
        bfs(&g, g.vertexes + 0);

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

길이 0/1 그래프 최단거리

```
/* 양방향 큐 구조체와 함수 */
struct deque;
struct deque *new_deque(int size);
void push_front(struct deque *q, struct node *n);
void push_rear(struct deque *q, struct node *n);
struct node *pop_front(struct deque *q);
struct node *pop_rear(struct deque *q);
int deque_is_empty(struct deque *q);
void delete_deque(struct deque *q);
```

길이 0/1 그래프 최단거리

```
void bfs_0_1(struct graph *g, struct node *s)
{
        for (int i = 0; i < g->n_vertexes; i++) {
                struct node *u = g->vertexes + i;
                u->distance = 9;
                u->parent = NULL;
                u->status = 0;
        }
        struct deque *q = new_deque(g->n_vertexes);
        print_graph_3col(g);
```

길이 0/1 그래프 최단거리

```
s->distance = 0;
push_front(q, s);
print_graph_3col(g);

while (!deque_is_empty(q)) {
        struct node *u = pop_front(q);
        if (u->status == 1) continue;
        u->status = 1;
        print_graph_3col(g);
```

길이 0/1 그래프 최단거리

```
struct elist *near = u->adj;
while (near != NULL) {
        struct edge *e = near->edge;
        int dist = u->distance + e->length;
        struct node *v;
        if (e->node1 == u)
                v = e->node2;
        else
                v = e->node1;
```

길이 0/1 그래프 최단거리

```
if (v->distance > dist) {
        v->parent = u;
        v->distance = dist;
        if (e->length == 0)
                push_front(q, v);
        else
                push_rear(q, v);
        print_graph_3col(g);
}
```

길이 0/1 그래프 최단거리

```
                near = near->next;
            }
        }
        delete_deque(q);
}
```

길이 0/1 그래프 최단거리

```
int main() {
        /* 이전 내용에 이어서 */
        g.edges[0].length = 0;
        g.edges[3].length = 0;

        bfs_0_1(&g, g.vertexes + 0);
        free_adj_list(&g);
        return 0;
}
```

길이 ≥ 0 그래프 최단거리

```
/* 양방향 큐 구조체와 함수 */
struct pqueue;
struct pqueue *new_pqueue(int size);
void push(struct pqueue *pq, struct node *n);
struct node *pop(struct pqueue *pq);
int pqueue_is_empty(struct pqueue *pq);
void delete_pqueue(struct pqueue *pq);
```

길이 ≥ 0 그래프 최단거리

```
void dijkstra(struct graph *g, struct node *s)
{
        for (int i = 0; i < g->n_vertexes; i++) {
                struct node *u = g->vertexes + i;
                u->distance = 9;
                u->parent = NULL;
                u->status = 0;
        }
        struct pqueue *pq = new_pqueue(g->n_edges);
        print_graph_3col(g);
```

길이 ≥ 0 그래프 최단거리

```
s->distance = 0;
push(q, s);
print_graph_3col(g);

while (!pqueue_is_empty(pq)) {
        struct node *u = pop(pq);
        if (u->status == 1) continue;
        u->status = 1;
        print_graph_3col(g);
```

길이 ≥ 0 그래프 최단거리

```
struct elist *near = u->adj;
while (near != NULL) {
        struct edge *e = near->edge;
        int dist = u->distance + e->length;
        struct node *v;
        if (e->node1 == u)
                v = e->node2;
        else
                v = e->node1;
```

길이 ≥ 0 그래프 최단거리

```
if (v->distance > dist) {
        v->parent = u;
        v->distance = dist;
        push(pq, v);
        print_graph_3col(g);
}
```

길이 ≥ 0 그래프 최단거리

```
                near = near->next;
            }
        }
        delete_pqueue(pq);
}
```

길이 ≥ 0 그래프 최단거리

```
int main() {
        /* 이전 내용에 이어서 */
        g.edges[4].length = 3;

        dijkstra(&g, g.vertexes + 0);
        free_adj_list(&g);
        return 0;
}
```

요약

- 그래프 출력하기
- 길이 1 그래프 - 너비 우선 탐색
- 길이 0 / 1 그래프 - 양방향 큐
- 길이 ≥ 0 그래프 - 우선순위 큐

너비 우선 탐색 - 큐

```
struct queue {
        struct node **data;
        int front;
        int rear;
};
```

너비 우선 탐색 - 큐

```
struct queue *new_queue(int size)
{
        struct queue *q = malloc(sizeof(struct queue));
        q->data = malloc(sizeof(struct node*) * size);
        q->front = 0;
        q->rear = 0;
        return q;
}
```

너비 우선 탐색 - 큐

```
void enqueue(struct queue *q, struct node *n)
{
        q->data[q->rear++] = n;
}
```

너비 우선 탐색 - 큐

```
struct node *dequeue(struct queue *q)
{
        return q->data[q->front++];
}
```

너비 우선 탐색 - 큐

```
int is_empty(struct queue *q)
{
        return q->front == q->rear;
}
```

너비 우선 탐색 - 큐

```
void delete_queue(struct queue *q)
{
        free(q->data);
        free(q);
}
```

길이 0/1 그래프 최단거리 - 양방향 큐

```
struct deqnode {
        struct node *data;
        struct deqnode *prev, *next;
};
```

길이 0/1 그래프 최단거리 - 양방향 큐

```
struct deque {
        struct deqnode *head;
};
```

길이 0/1 그래프 최단거리 - 양방향 큐

```
struct deqnode *new_deqnode(struct node *n)
{
        struct dqnode *dnode
                = malloc(sizeof(struct deqnode));
        dnode->data = n;
        dnode->prev = dnode;
        dnode->next = dnode;
        return dnode;
}
```

길이 0/1 그래프 최단거리 - 양방향 큐

```
struct deque *new_deque(int size)
{
        struct deque *q = malloc(sizeof(struct deque));

        if (q == NULL) return NULL;

        struct deqnode *dummy = new_deqnode(NULL);
```

길이 0/1 그래프 최단거리 - 양방향 큐

```
        if (dummy == NULL) {
                free(q);
                return NULL;
        }

        q->head = dummy;
        return q;
}
```

길이 0/1 그래프 최단거리 - 양방향 큐

```
int push_front(struct deque *q, struct node *n)
{
        struct deqnode *dnode = new_deqnode(n);
        if (dnode == NULL) return -1;
        dnode->prev = q->head;
        dnode->next = q->head->next;
        q->head->next->prev = dnode;
        q->head->next = dnode;
        return 0;
}
```

길이 0/1 그래프 최단거리 - 양방향 큐

```
int push_rear(struct deque *q, struct node *n)
{
        struct deqnode *dnode = new_deqnode(n);
        if (dnode == NULL) return -1;
        dnode->next = q->head;
        dnode->prev = q->head->prev;
        q->head->prev->next = dnode;
        q->head->prev = dnode;
        return 0;
}
```

길이 0/1 그래프 최단거리 - 양방향 큐

```
struct node *pop_front(struct deque *q)
{
        if (q->head->next == q->head) return NULL;
        struct deqnode *old_front = q->head->next;
        struct node *n = old_front->data;
        old_front->next->prev = q->head;
        q->head->next = old_front->next;
        free(old_front);
        return n;
}
```

길이 0/1 그래프 최단거리 - 양방향 큐

```
struct node *pop_rear(struct deque *q)
{
        if (q->head->prev == q->head) return NULL;
        struct deqnode *old_rear = q->head->prev;
        struct node *n = old_rear->data;
        old_rear->prev->next = q->head;
        q->head->prev = old_front->prev;
        free(old_rear);
        return n;
}
```

길이 0/1 그래프 최단거리 - 양방향 큐

```
int deque_is_empty(struct deque *q)
{
        return q->head->next == q->head;
}
```

길이 0/1 그래프 최단거리 - 양방향 큐

```
void delete_deque(struct deque *q) {
        struct deque_node *n = q->head->next;
        while (n != q->head) {
                struct deque_node *old = n;
                n = n->next;
                free(old);
        }
        free(q->head);
        free(q);
}
```

길이 ≥ 0 그래프 최단거리 - 우선순위 큐

```
struct pqueue {
        struct node **data;
        int *key;
        int size;
};
```

길이 ≥ 0 그래프 최단거리 - 우선순위 큐

```
struct pqueue *new_pqueue(int size)
{
        struct pqueue *pq
                = malloc(sizeof(struct pqueue));
        if (pq == NULL) return NULL;
        pq->data = malloc(sizeof(struct node *) * size);
        pq->key = malloc(sizeof(int) * size);
        pq->size = 0;
        return pq;
}
```

길이 ≥ 0 그래프 최단거리 - 우선순위 큐

```
void swap(struct pqueue *pq, int i, int j) {
        struct node **data = pq->data;
        int *key = pq->key;
        int tmp = key[i];
        struct node *tmp_node = data[i];
        key[i] = key[j];
        data[i] = data[j];
        key[j] = tmp;
        data[j] = tmp_node;
}
```

길이 ≥ 0 그래프 최단거리 - 우선순위 큐

```
void push(struct pqueue *pq, struct node *n)
{
        int i = pq->size;
        struct node **data = pq->data;
        int *key = pq->key;

        data[i] = n;
        key[i] = n->distance;
        pq->size = i + 1;
```

길이 ≥ 0 그래프 최단거리 - 우선순위 큐

```
        while (i > 0) {
                int p = (i - 1) / 2;
                if (key[i] >= key[p])
                        break;
                swap(pq, i, p);
                i = p;
        }
        printf("push: node %c[%d], size = %d\n",
                n->data, n->distance, pq->size);
}
```

길이 ≥ 0 그래프 최단거리 - 우선순위 큐

```
struct node *pop(struct pqueue *pq)
{
        struct node **data = pq->data;
        int *key = pq->key;
        int size = pq->size;
        struct node *result = data[0];
        int distance = key[0];
        swap(pq, 0, size - 1);
        pq->size = size = size - 1;
```

길이 ≥ 0 그래프 최단거리 - 우선순위 큐

```
int i = 0;
while (2 * i + 1 < size) {
        int c = 2 * i + 1;
        if (c + 1 < size && key[c + 1] < key[c])
                c = c + 1;
        if (key[i] <= key[c])
                break;
        swap (pq, i, c);
        i = c;
}
```

길이 ≥ 0 그래프 최단거리 - 우선순위 큐

```
        printf("pop: node %c[%d], size = %d\n",
                result->data, distance, pq->size);
        return result;
}
```

길이 ≥ 0 그래프 최단거리 - 우선순위 큐

```
int pqueue_is_empty(struct pqueue *pq)
{
        return pq->size == 0;
}
```

길이 ≥ 0 그래프 최단거리 - 우선순위 큐

```
void delete_pqueue(struct pqueue *pq)
{
        free(pq->data);
        free(pq->key);
        free(pq);
}
```