Data Structure Assignment

2016320205 이지혜

2015410072 김재현

1. 학생들의 성적을 내림차순으로 정렬

#include <stdio.h> 표준 입출력을 위한 헤더

#include <stdlib.h> 메모리를 할당하는 malloc과 배열 요소 개수를 세는 _countof 함수를 사용하기위한 헤더

#include <time.h> 시간을 측정하기 위한 헤더

#define BUCKETS 10 **Radix sort를 위한 큐 배열 크기**

#define BIGITS 3 **Radix sort를 위한 숫자 자리 수 정의**

typedef struct {...} Student;

typedef로 char 배열로 이름, int 자료형으로 학번과 점수를 저장하는 구조체를 Student라고 명명하였다.

typedef struct {...} StudentNode;

typedef로 Student배열을 저장하는 구조체를 StudentNode라고 명명하였다.

typedef struct {...} Queue;

typedef로 Student node와 다음 Queue를 가리키는 Queue 포인터 link를 담고 있는 구조체를 Queue라고 명명하였다.

typedef struct {...} QueueType;

typedef로 front와 rear를 가리키는 Queue형 포인터를 담고 있는 구조체를 QueueType이라고 명명하였다.

void init_student (Student s[]);

입력 받은 Student 배열을 초기화한다. 학번과 이름은 Student_info.txt라는 텍스트 파일로부터 입력 받으며, 파일을 읽어오지 못하면 '망함'을 출력하고 return한다. 파일이 무사히 열리면 '열림'을 출력하고 학번과 공백을 포함한 이름을 읽어온다.

void init_score (StudentNode *s);

StudentNode형 *s를 인자로 받아와 StudentNode의 Student형 배열의 score를 0~100사이의 랜덤한 값으로 초기화한다.

void insert_node (StudentNode *s, Student node[]);

StudentNode형 *s와 Student 형 node배열을 인자로 받아와 s의 node를 인자 node배열의 데이터로 초기화한다.

void display (StudentNode *s);

StudentNode형 *s를 인자로 받아와 s의 데이터들을 출력한다.

void swap (StudentNode *s, int i, int j);

StudentNode형 *s과 int형 i, j를 인자로 받아와 s의 i번째 node와 j번째 node의 데이터를 바꾼다.

void selection_sort (StudentNode *s);

StudentNode형 *s를 인자로 받아와 s의 node들을 score순으로 내림차순 selection sort한다.

void insertion_sort (StudentNode *s);

StudentNode형 *s를 인자로 받아와 s의 node들을 score순으로 내림차순 insertion sort한다.

void bubble_sort (StudentNode *s);

StudentNode형 *s를 인자로 받아와 s의 node들을 score순으로 내림차순 bubble sort한다.

void shell_sort (StudentNode *s);

StudentNode형 *s를 인자로 받아와 s의 node들을 score순으로 내림차순 shell sort한다.

void dec_insertion_sort (StudentNode *s, int start, int gap);

StudentNode형 *s과 int형 start, gap를 인자로 받아와 score순으로 start부터 gap만큼의 차이가 나는 위치의 node들을 내림차순 insertion sort한다.

void merge (StudentNode *s, int left, int mid, int right);

StudentNode형 *s와 int형 left, mid, right를 인자로 받아와 Student형 sorted 배열에 s의 node 데이터를 대입한다. 이때, s의 node를 left번째부터 mid, mid + 1번째부터 right 로 나누고, 전자의 score가 크면 sorted에 대입한다. 그렇지 않으면, 후자를 sorted에 대입한다. Index를 1씩 증가시켜가며 어느 한 인덱스가 mid또는 right에 도달하면 sorted에 대입을 멈추고, 나눠진 요소들 중 남은 것을 sorted에 대입한다.

void merge_sort (StudentNode *s, int left, int right);

StudentNode형 *s를 인자로 받아와 s의 node들을 score순으로 내림차순 merge sort한다.

int partition (StudentNode *s, int left, int right);

StudentNode형 *s와 int형 left, right를 받아와 가운데 node의 score를 기준으로 큰 것은 왼쪽에, 작은 것은 오른쪽에 위치하도록 swap한다.

void quick_sort (Student s[], int left, int right);

StudentNode형 *s를 인자로 받아와 s의 node들을 score순으로 내림차순 quick sort한다.

void init_queue (QueueType *q);

QueueType형 *q를 인자로 받아와 q의 front, rear를 NULL로 초기화한다.

int is_empty (QueueType *q);

QueueType형 *q를 인자로 받아와 q가 비어 있는지, 즉 front가 NULL인지 여부를 리턴한다.

void enqueue (QueueType *q, Student s);

QueueType형 *q와 Student형 s를 받아와 q에 s를 enqueue한다.

Student dequeue (QueueType *q);

QueueType형 *q를 인자로 받아와 dequeue하고 dequeue한 Student형 node를 리턴 한다.

void radix_sort (StudentNode *s, QueueType q[]);

StudentNode형 *s를 인자로 받아와 s의 node들을 score순으로 내림차순 radix sort한다.

int main (void) {...}

정보를 받아올 Student형 student를 선언한 뒤, init_student() 함수로 초기화 하였다. StudentNode형 포인터 s에 메모리를 할당해준 뒤, insert_node() 함수로 student의 데이터로 s의 node를 초기화하였다. 또한, init_score() 함수로 s의 점수를 초기화하고, 이를 출력하였다. Radix sort에 사용할 QueueType형 q를 init_queue() 함수로 초기화하였다. Selection sort, insertion sort, bubble sort, shell sort, merge sort, quick sort, radix sort를 각각 testcount 만큼 실행한 뒤, 각각의 실행시간을 출력하였다.

```
Selection sort  * 1000 = 24ms   average : 0ms
Insertion sort  * 1000 = 38ms   average : 0ms
Bubble sort     * 1000 = 159ms  average : 0ms
Shell sort      * 1000 = 25ms   average : 0ms
Merge sort      * 1000 = 29ms   average : 0ms
Quick sort      * 1000 = 28ms   average : 0ms
Radix sort      * 1000 = 138ms  average : 0ms
```

각 정렬의 결과는 다음과 같으며, 순서는 Selection > Shell > Quick > Merge > Insertion > Radix

> Bubble 순이다. 우선, Selection sort나 Shell sort가 Quick sort보다 빠른 이유는 첫 번째로, 정렬하는 Input의 개수가 대단히 적기 때문이고, 두 번째로, Quick sort와 Shell sort는 재귀문으로 작성되었지만, Selection sort는 반복문으로 작성되었기 때문이다. (재귀문 호출 횟수도 Shell sort보다 Quick sort가 더 많다.) 또한 알고리즘을 개선했다는 점에서 Shell sort는 Insesrtion sort보다 빠르다는 점을 알 수 있다. 그리고 주목할 만한 점은 Radix sort인데, 큐를 링크드 큐로 구현했기 때문에, 시간이 오래 걸리는 점을 알 수 있다. 또, 반복문으로 구현하였음에도 다른 모든 정렬보다 느린 Bubble sort는 솔직히 지구가 멸망할 때까지 사용해선 안 된다.

마지막으로, 어떤 정렬을 쓰더라도, 평균 소요 시간은 ms 미만이므로, Input이 적을 때는 어떤 정렬을 써도 무방한 것을 알 수 있다. 하지만 그래도 Bubble sort는 쓰면 안 된다.

2. 학생들의 성적을 BST로 표현

#include <stdio.h> 표준 입출력을 위한 헤더

#include <stdlib.h> 메모리를 할당하는 malloc과 배열 요소 개수를 세는 _countof 함수를 사용하기위한 헤더

typedef struct {...} Student;

typedef로 char 배열로 이름, int 자료형으로 학번과 점수를 저장하는 구조체를 Student라고 명명하였다.

typedef struct {...} StudentNode;

typedef로 Student배열을 저장하는 구조체를 StudentNode라고 명명하였다.

typedef struct {...} TreeNode;

typedef로 Student형 key, TreeNode형 포인터 left, right를 저장하는 구조체를 TreeNode라고 명명하였다.

void init_student (Student s[]);

입력 받은 Student 배열을 초기화한다. 학번과 이름은 Student_info.txt라는 텍스트 파일로부터 입력 받으며, 파일을 읽어오지 못하면 'failed to open file.'을 출력하고 return한다. 파일이 무사히 열리면 'Success to open file!'을 출력하고 학번과 공백을 포함한 이름을 읽어온다. 그런 다음, 점수는 순서대로 0~99점을 부여한다.

void display (StudentNode *s);

StudentNode형 *s를 인자로 받아와 s의 데이터들을 출력한다.

void connect_node (StudentNode *s, Student node[]);

StudentNode형 *s와 Student형 node 배열을 받아와 s의 node를 node배열의 데이터로 초기화

한다.

void init_tree (TreeNode **t);

TreeNode형 **t를 인자로 받아와 *t를 NULL로 초기화한다.

void mix_node (StudentNode *s);

StudentNode형 *s를 인자로 받아와 s의 node들을 랜덤하게 섞는다.

void insert_node (TreeNode **root, StudentNode *s);

TreeNode형 **root와 StudentNode형 *s를 인자로 받아와 s를 root를 루트로 하는 트리에 BST 규칙에 따라 삽입한다.

void preorder (TreeNode *root);

TreeNode형 *root를 인자로 받아와 root를 루트로 하는 트리를 전위 순회 하여 출력한다.

Student search_node (TreeNode *root, int score);

TreeNode형 *root와 int형 score를 인자로 받아와 root를 루트로 하는 트리에서 score와 같은 점수를 가진 Student형 key를 리턴한다. 탐색하면서 지나간 노드를 출력하며, 찾지 못했다면 메시지를 출력한다.

void delete_node(TreeNode **root, int score);

TreeNode형 **root와 int형 score를 인자로 받아와 root를 루트로 하는 트리에서 score와 같은 점수를 가진 Student형 key를 삭제한다. 삭제하려는 key가 없다면 '삭제하려는 key가 트리에 없습니다.'를 출력하고 리턴한다. 삭제하려는 노드가 단말 노드라면, '삭제하려는 노드는 자식이 없습니다.'를 출력하며, 삭제 노드의 부모 노드를 단말 노드로 만드는 방법을 통해 삭제 노드를 트리에서 삭제하고 삭제 노드에 할당된 메모리를 해제한다. 또, 삭제하려는 노드가 한 개의 자식을 가지면, 삭제 노드의 부모와 자식을 연결하는 방법을 통해 삭제 노드를 트리에서 삭제하고 삭제 노드에 할당된 메모리를 해제한다. 마지막으로, 삭제하려는 노드가 두 개의 자식을 가지면, 후속자를 찾아 삭제 노드의 위치와 변경하고, 후속자의 자리를 메우는 방법을 통해 삭제 노드를 트리에서 삭제하고 삭제 노드에 할당된 메모리를 해제한다.

int main (void) {...}

Student형 배열 student을 선언하고, StudentNode형 *s에 메모리를 할당한다. 또, TreeNode형 *root에 메모리를 할당한다.

init_student() 함수로 student에 데이터를 대입하고, connect_node() 함수로 student의 데이터들을 s에 저장한다. mix_node() 함수로 s의 node를 랜덤하게 정렬하고, 이를 display() 함수로 출력한다.

insert_node() 함수로 root에 s의 node들을 삽입하고, preorder() 함수로 앞선 트리에 담긴 노드들을 전위 순회하여 출력한다.

search_node() 함수를 통해 score 값이 각각 96, 98, 95, 100인 node 들을 찾는다. 각각은 단말 노드, 1개의 자식 노드를 가지는 노드, 2개의 자식 노드가 있는 노드, 트리에 없는 노드이며, 각 노드를 삭제한 후 preorder() 함수를 통해 트리를 전위 순회하여 출력한다.

3. 그래프의 최단거리를 Dijkstra, Floyd로 구하기

#include <stdio.h> 표준 입출력을 위한 헤더

#define INF 100000 노드의 거리 초기화를 위한 충분히 큰 수

#define MAX_NODE 9 그래프의 노드 개수

typedef struct {...} Graph;

typedef로 노드 개수와 노드 사이의 가중치를 담은 2차원 배열을 담은 구조체를 Graph라고 명명하였다.

typedef struct {...} Heap;

typedef로 시작 노드와 도착 노드, 두 노드 사이의 거리를 담은 구조체를 Heap이라고 명명하였다.

typedef struct {...} HeapType;

typedef로 힙 정렬된 Heap을 담을 Heap형 배열과 Heap의 내용물의 개수를 나타내는 int형 size를 담은 구조체를 HeapType이라고 명명하였다.

enum Vertex {...};

enum을 통해 각 vertex들을 0~9로 설정하였다.

static char enum_string (enum Vertex v) {...};

enum Vertex형 v를 인자로 받아와 v에 해당하는 vertex의 이름을 return한다.

void init_graph (Graph *g);

Graph형 *g를 인자로 받아와 g의 가중치들을 INF로, 행과 열이 같은 배열은 0으로 초기화 한 뒤 insert_edge() 함수를 통해 vertex 사이의 가중치를 입력한다.

void insert_edge (Graph *g, int start_edge, int end_edge, int dis);

Graph형 *g와 int형 start_edge, end_edge, dis를 받아와 start_edge와 end_edge 사이를 dis로 입력받은 가중치로 g에 초기화한다.

void init_heap (HeapType *h);

HeapType *h를 인자로 받아와 h의 size를 0으로 초기화 한다.

void insert_min_heap (HeapType *h, int start_edge, int end_edge, int dis);

HeapType형 *h와 int형 start_edge, end_edge, dis를 인자로 받아와 dis를 기준으로 최소 힙 정렬한 index를 찾은 후, 해당 index에 인자로 받아온 값을 초기화한다.

Heap delete_min_heap (HeapType *h);

HeapType형 *h를 인자로 받아와 h의 루트 노드를 삭제하고 리턴한다. 삭제한 뒤, 다시 h의 node들을 최소 힙 정렬한다.

void find_path (Graph *g, HeapType *h, int start_edge);

Graph형 *g와 HeapType *h, int형 start_edge를 인자로 받아와 힙을 통한 우선순위 큐와 Dijkstra 알고리즘을 통해 최단거리를 구한다. Found를 통해 탐색한 노드를 표시한다. 현재 가지고 있는 경로보다 더 짧은 경로가 있다면 더 짧은 경로로 대체한다. 다음 경로를 찾기 위해서 현재 탐색하고 있는 노드와 이어진 노드를 힙에 삽입하며, 이때 현재 탐색 노드와 이어진 노드의 경로는 시작 노드에서 현재 탐색 노드까지의 거리와 탐색 노드에서 이어진 노드까지의 거리를 더한 값이다.

void floyd (Graph *g);

Graph형 *g를 인자로 받아와 Floyd 알고리즘을 통해 최단거리를 구한다. 모든 간선들을 순회하면서 현재 가지고 있는 경로보다 짧은 경로가 있다면 업데이트를 하는 방법으로 최단거리를 구한다.

void display (Graph *g, int start_edge, int end_edge);

Graph형 *g와 int형 start_edge, end_edge를 인자로 받아와 start edge에서 end edge까지의 최단거리를 출력한다.

void display_all (Graph *g);

Graph형 *g를 인자로 받아와 g에 존재하는 모든 node간의 최단거리를 출력한다.

int main (void) {...}

Graph형 g1, g2를 선언하고 HeapType형 h를 선언하였다. init_graph() 함수를 통해 g1, g2를 초기화하고, init_heap() 함수를 통해 h를 초기화하였다. find_path() 함수를 통해 Dijkstra 알고리즘으로 최단거리를 찾고, floyd() 함수를 통해 Floyd 알고리즘으로 최단거리를 찾았다. 그리고 display(), display_all() 함수를 통하여 최단거리를 출력한다.