heap

-> priority queue를 이루는 자료구조

부모노드가 들어온 자식보다 작아야 함

루트노드와 끝 노드를 계속 비교하면서 작은 값을 우선 순위로 출력한다

단순한 자료구조로 push와 pop을 할 때 자동으로 오름차순으로 정렬해 주도록 하는 자료구조다.

min heap 구현

#include <iostream>  
using namespace std;  
​  
int heap[30];  
int index = 0;  
​  
void push(int value){  
   heap[++index] = value; //1번 인덱스부터 삽입  
   //부모와 넘어온 값  
   int p,now;  
   now = index;  
     
   while(1){  
       //부모 인덱스 확인  
       p= now/2;  
       //1번 인덱스만 채워짐   
       //비교할 대상이 없으므로 브레이크  
       if(now==1) break;  
       //부모랑 들어온 값을 비교 후 스왑  
       if(heap[p] > heap[now]) swap(heap[p],heap[now]);  
       else break;  
       //부모를 기점으로 부모의 부모와 비교  
       now=p;  
  }  
}  
​  
int top(){  
   return heap[1];  
}  
​  
void pop(){  
   //맨 끝을 맨 위로 올리고   
   //맨 끝 값을 지움  
   heap[1] = heap[index];  
   heap[index] = 0;  
   index--;  
     
   int p, son, son2;  
     
   p = 1;  
     
   while(1){  
       son = p\*2;  
       son2 = p\*2 + 1;  
       // 오른쪽에 자식이 있고 왼쪽 자식들과 비교  
       //작은 자식을 son으로 변경  
       if(son2 <= index && heap[son] > heap[son2]){  
           son = son2;  
      }  
       //배열에 마지막만 남거나 부모가 자식보다 작으면 break  
       if(son > index || heap[p] < heap[son]) break;  
       else swap(heap[p],heap[son]);  
         
       p=son;  
  }  
}  
​  
int main(){  
     
   push(9);  
   push(3);  
   push(5);  
   push(2);  
   push(8);  
   push(4);  
   push(1);  
     
     
     
   return 0;  
}  
​

Priority queue

-> queue에 있는 stl로 push만 해줘도 기본 max heap이 되어있기 때문에

우선순위가 높은 순서대로 출력된다.

min heap으로 사용하기 위해서는

priority queue<int,vector<int>,greater<tint>> q; 로 설정하면 된다.

struct compare {  
bool operator()(node &a, node &b) {  
if (a.n < b.n) return false;  
if (a.n > b.n) return true;  
return a.alp > b.alp;  
}  
};

우선 순위 queue에서 compare를 사용하려면 구조체에 넣고 operator 함수를 사용해야 한다.

또한, pq의 경우 기본적으로 max heap이 바탕이기 때문에

일반적인 compare 함수와는 다르게 오름차순으로 정렬하기를 원한다면

return 값을 true가 아닌 false로 해줘야한다.

#include <queue>  
#include <iostream>  
#include <vector>  
using namespace std;  
​  
​  
priority\_queue<int,vector<int>,greater<int>> pq;  
vector<int> vect;  
int main() {  
​  
int n;  
cin >> n;  
​  
pq.push(1);  
int now;  
int before = 0;  
​  
for (int i = 0; i < n; i++) {  
//중복 제거  
while (1) {  
now = pq.top();  
pq.pop();  
​  
if (now != before) break;  
}  
before = now;  
vect.push\_back(now);  
​  
pq.push(now \* 2);  
pq.push(now \* 3);  
pq.push(now \* 5);  
}  
​  
cout << vect[n - 1] << endl;  
​  
return 0;  
}

\*2, \*3, \*5한 값을 넣고 n번째 수 출력(중복 제거)

다익스트라 알고리즘

한 정점에서 다른 정점까지의 비용을 경유지를 사용하거나 사용하지 않는 방법을 따져서

최소 비용을 찾는 알고리즘

#include <iostream>  
using namespace std;  
char name[6] = "ABCDE";  
const int m = 999;  
​  
int map[5][5] = {  
m,3,m,9,5,  
m,m,7,m,1,  
m,m,m,1,m,  
m,m,m,m,m,  
m,m,1,m,m,  
};  
​  
int result[5] = {  
m,3,m,9,5,  
};  
​  
int used[5] = { 1 };  
​  
int check() {  
int Min = 21e8;  
int Minindex = 0;  
​  
for (int x = 0; x < 5; x++) {  
if (used[x]==0 && result[x] < Min) {  
Min = result[x];  
Minindex = x;  
}  
}  
return Minindex;  
}  
​  
void daijk() {  
​  
for (int y = 0; y < 4; y++) {  
int via = check(); //경유지 체크 함수  
used[via] = 1;  
​  
for (int x = 0; x < 5; x++) {  
int baro = result[x];  
int kyeong = result[via] + map[via][x];  
if (baro > kyeong) result[x] = kyeong;  
}  
}  
}  
​  
int main() {  
​  
daijk();  
​  
char ends;  
cin >> ends;  
​  
for (int x = 0; x < 5; x++) {  
if (name[x] == ends) {  
cout << result[x];  
break;  
}  
}  
​  
return 0;  
}

O(n^2)의 시간 복잡도

priority\_queue를 활용하면 O(nlogn)의 속도로 daijkstra 알고리즘이 가능하다

#include <iostream>  
#include<queue>  
#include<vector>  
using namespace std;  
char name[6] = "ABCDE";  
​  
struct node {  
int n; // 도착지 index  
int price;   
};  
​  
bool operator<(node back, node front) {  
return front.price < back.price;  
}  
​  
vector<vector<node>> vect(5);  
​  
priority\_queue<node> q;  
const int m = 999;  
​  
int result[5] = {  
m,m,m,m,m,  
};  
​  
int used[5] = { 1 };  
​  
​  
​  
int main() {  
​  
vect[0].push\_back({ 1,3 });  
vect[0].push\_back({ 3,9 });  
vect[0].push\_back({ 4,5 });  
vect[1].push\_back({ 2,7 });  
vect[1].push\_back({ 4,1 });  
vect[2].push\_back({ 3,1 });  
vect[4].push\_back({ 2,1 });  
​  
q.push({ 0,0 });  
​  
while (!q.empty()) {  
node via = q.top();  
q.pop();  
​  
if (used[via.n] == 1) continue;  
used[via.n] = 1;  
​  
//result vs 경유지 거쳐서 가는 비용 비교 후 result 갱신  
​  
for (int x = 0; x < vect[via.n].size(); x++) {  
node target = vect[via.n][x];  
​  
if (result[target.n] > via.price + target.price) {  
result[target.n] = via.price + target.price;  
q.push({target.n,result[target.n]});  
}  
}  
}  
​  
return 0;  
}

다익스트라 알고리즘은 노드 사이에 연결된 비용을 설정하고 목적 노드까지 경유해서 가는 것이 싼지 바로 가는 것이 싼지를 결정하는 알고리즘이다.

Priority queue를 사용하지 않을 경우는 used 배열과 result 배열을 만들어 준 후 시작 노드의 비용은 최대 비용으로 설정, 시작 노드와 연결된 노드의 비용을 result 배열에 넣어주고 바로 연결되지 않은 배열 역시 최대 비용으로 설정한다.

경유지 체크 함수를 이용해서 경유지를 이용해서 가는 경로와 바로 가는 경로 사이의 비용을 비교해서 값이 싼 비용을 result에 넣어주는 것을 반복하면서 설정한다.

Priority queue를 이용하는 경우는 priority queue자체가 max heap이기 때문에 compare 함수를 이용해 min heap으로 설정해 준 뒤, 시작 노드와 연결된 노드를 집어 넣으면 자동으로 비용이 싼 노드가 나오기 때문에 그 노드를 경유 노드로 설정, 위에서와 같이 바로 가는 경로와 경유지를 들르는 경로의 비용을 비교해서 result 배열에 설정해준다.