< 자료구조>

결과 보고서

COME331006

Prof. 김종화 교수님

전자공학부 A

2017115976 이하영

(1)Heap ADT작성 및 priority heap응용

1)간단한 설명

Queue에서는 어떤 data가 들어오면 순서대로 처리된다. 즉, 줄서기와 같이 data가 들어온 순서대로 정렬되고 빠져나갈 때도 먼저 들어온 data순으로 처리된다. 하지만 Queue에서는 우리가 원하는 data를 다른 특정 data보다 우선적으로 정렬하게 할 순 없다. 이 때 heap구조를 이용하여 우리가 원하는 data에 우선순위를 부여한다면, 그 data의 위치를 이미 정렬된 다른 data들 보다 우선적으로(순서대로가 아닌) 배치할 수 있다. 지금부터 볼 priority heap구조는 고객정보를 다루는 알고리즘이다. Time sale 기간에 고객들이 줄을 섰다고 가정하자. 일반적인 Queue 구조에서는 줄을 선 고객부터 차례대로 입장할 것이다. 하지만 만약 VIP인 고객이라면 줄을 늦게 서더라도 우선순위를 부여해야할 것이다. 따라서 heap구조를 이용해 고객정보에 우선순위크기를 부여하여 줄을 선 순서와 상관없이 우선순위가 큰 사람부터 입장할 수 있도록 만들것이다. 만약 우선순위가 같은 사람이 줄을 섰다면, 줄을 선 순서대로 입장하게 된다.

2)코드 분석

<heap.h>

① HEAP 구조체 선언

```
typedef struct {

    void** heapAry;
    int last;
    int size;
    int(*compare)(void* argu1, void* argu2);
    int maxSize;
}HEAP;
```

→heap 구조체 멤버로 heapAry, last, size, compare함수, maxSize를 선언한다.

② HEAP* heapCreate(int maxSize, int(*compare)(void* argu1, void* argu2))

```
HEAP* heapCreate(int maxSize, int(*compare)(void* argu1, void* argu2)) {
    HEAP* heap;
    heap=(HEAP*)malloc(sizeof(HEAP));
    if (!heap)
        return NULL;

    heap->last = -1;
    heap->compare = compare;

    heap->maxSize = (int)pow(2, ceil(log2(maxSize))) - 1;
    heap->heapAry = (void**)calloc(heap->maxSize, sizeof(void*));
    return heap;

}
```

→동적 메모리 영역에 HEAP구조체 포인터를 할당한다.

→heap 구조체를 초기화하고 heapAry에는 calloc을 사용하여 할당과 동시에 초기화를 한다. heap->maxSize는 할당공간의 개수를 의미하고 sizeof(void*)는 할당공간의 크기를 의미한다.

③ void reheapUp(HEAP* heap, int childLoc)

```
void reheapUp(HEAP* heap, int childLoc) {
    int parent;
    void** heapAry;
    void* hold;

    if (childLoc) {
        heapAry = heap->heapAry;
        parent = (childLoc - 1) / 2;//여기서 childLoc은 insert된 즉 위로 올라갈 node, parent는 childLoc위에 있는 childLoc과 자리바꿀 아이.
        if (heap->compare(heapAry[childLoc], heapAry[parent]) > 0) {//child가 더크면
            hold = heapAry[parent];//현재 parent는 hold하고
            heapAry[parent] = heapAry[childLoc];//child를 parent자리에 대입? 여튼 둘이 자리바꾼다.
            reheapUp(heap, parent);//child에서 parent가 된 애랑 다시 그 위의 parent랑 비교
        }
    }
    return;
}
```

→reheapUp함수의 매개변수로 선언된 childLoc이 만약 제 값이 있다면 if문을 실행하게 된다. 이때 이중포인터인 heapAry는 구조체포인터 HEAP멤버의 heapAry와 같다.

→parent는 childnode의 윗단계인 node를 의미한다. Parent에 (childLoc-1)/2한 주소값을 준다.

→heapAry[childLoc]:childLoc이 배열의 위치를 의미하므로 heapAry[childLoc]은 그 위치의 data값이다.

마찬가지로 heapAry[parent] int형 parent의 배열 위치에 해당하는 data값을 의미한다.

→compare함수를 통해 두 data를 비교했을 때 childLoc에 저장된 data값이 더 크다면(즉, childnode값이 더 크면) 현재 parentnode 값은 저장하고 parentnode 자리에 childenode값을 넣어주어 두 data의 위치를 바꾼다.

→이렇게 함으로써 첫번째 reheapUp이 이루어지고 재귀함수 구조를 통해서 처음 childLoc에 있던 data값은 다른 data들과의 크기비교를 통해 heap 구조에서 자기자리에 맞는 위치에 배열될 것이다.

④ bool heapInsert(HEAP* heap, void* dataPtr)

```
bool heapInsert(HEAP* heap, void* dataPtr) {
    if (heap->size == 0) {
        heap->size = 1;
        heap->last = 0;
        heap->heapAry[heap->last] = dataPtr;
        return true;
    }

    if (heap->last == heap->maxSize - 1)
        return false;
    ++(heap->last);
    ++(heap->size);
    heap->heapAry[heap->last] = dataPtr;
    reheapUp(heap, heap->last);
    return true;
}
```

→heap에 새로운 data를 넣어보자. 만약 heap 배열 크기(size)가 0이면 size는 1로 바꿔주고 last 값은 0으로 저장한다.

→heap->heapAry[heap->last]=dataptr : heapAry배열의 마지막부분에 insert할 dataPtr을 넣는다.

→reheapUp구조를 통해서 방금 삽입한 dataPtr과 그 위의 parentnode 값을 비교하여 dataPtr이 더 크면 parentnode와 자리를 바꾼다. 재귀구조를 통해서 reheapUp이 계속 실행되므로 마지막엔 insert한 dataPtr이 heap구조에 맞는 자기 자리를 찾을 것이다.

⑤ void reheapDown(HEAP* heap, int root)

```
void reheapDown(HEAP* heap, int root) {
    void* hold;
    void* leftData;
    void* rightData;
    int largeLoc;
    int last;

    last = heap->last;
    if ((root * 2 + 1) <= last) {
        leftData = heap->heapAry[root * 2 + 1];//왼자식의 값을 leftData에 삽입

        if ((root * 2 + 2) <= last)
            rightData = heap->heapAry[root * 2 + 2];//오자식의 값을 rightData에 삽입

        else
            rightData = NULL;

        if ((!rightData) || heap->compare(leftData, rightData) > 0) {//예를들어 만약 왼쪽자식 데이터가 더 크면
            largeLoc = root * 2 + 1;//largeLoc은 왼자식
        }
        else {
            largeLoc = root * 2 + 2;
        }
```

→last는 int형 변수로 배열의 마지막 위치를 나타낸다.

→leftData = heap->heapAry[root * 2 + 1] : 만약 root*2+1의 위치가 배열의 마지막 값보다 같거나 작으면 leftData는 배열의 root*2+1위치에 있는 값으로 저장된다.

→rightData = heap->heapAry[root * 2 + 2] : 만약 root*2+2의 위치가 배열의 마지막 값보다 같거나 작으면 rightData는 배열의 root*2+2의 위치에 있는 값으로 저장된다.

→if ((!rightData) || heap->compare(leftData, rightData) > 0) : compare함수는 통해 아까전에 저장했던 leftData값과 rightData값을 비교했을 때 leftData가 더크면 1을 반환하고 rightData가 더 크면 -1을 반환한다. 지금 if 문에서는 만약 leftData가 더 크다면을 가정한다.

→만약 leftData값이 더 크면 largeLoc에는 root*2+1의 배열 위치값을 저장하고 rightData값이 더 크면 largeLoc에는 root*2+2의 배열 위치값을 저장한다.

```
if (heap->compare(heap->heapAry[root], heap->heapAry[largeLoc]) < 0) {//largeLoc 여기서는 왼자식 즉 왼자식이 루트값보다 더 크면
    hold = heap->heapAry[root];//root값은 hold하고
    heap->heapAry[root] = heap->heapAry[largeLoc];//원래 root자리에는 자식 넣어주고
    heap->heapAry[largeLoc] = hold;//여기서 hold가 아까 잡았던 큰 root값이고 그 root(hold)를 largeLoc자리 즉 자식자리로 Down시킨다
    reheapDown(heap, largeLoc);//다시 그 밑의 자식들과 비교하면서 ..
}
```

→ 첫번째 사진의 과정을 통해서 root의 왼쪽자식과 오른쪽자식 중 누가 더 큰지 비교하고, 큰 자식의 배열위치를 largeLoc에 저장했었다. 이제 root값과 위에서 선택한 자식들을 비교해보자.

→만약 largeLoc에 있는 data값(root의 자식)이 root보다 크면 그 heap은 오류가 발생한다. 따라서 largeLoc값이 root보다 크면 두개 data의 위치를 서로 바꿔준다.

→ root값은 hold변수에 저장하고 heap->heapAry[root] = heap->heapAry[largeLoc]를 통해서 root자리에는 자식들 중 더 큰 node값을 저장한다.(즉, root(parent)값과 largeLoc위치에 있는 data값(children)을 서로 바꾼다!!)

→heap->heapAry[largeLoc] = hold : 여기서 hold는 root값을 의미한다. 따라서 largeLoc자리(children)에 hold값(root)을 넣어준다.

→ 재귀구조를 통해서 다시 root값을 밑의 자식들

과 비교해주면서 reheapDown을 실행한다.

⑥ bool heapDelete(HEAP* heap, void** dataOutPtr)

```
bool heapDelete(HEAP* heap, void** dataOutPtr) {
    if (heap->size == 0)
        return false;
    *dataOutPtr = heap->heapAry[0];
    heap->heapAry[0] = heap->heapAry[heap->last];
    (heap->last)--;
    (heap->size)--;
    reheapDown(heap, 0);
    return true;
}
```

→보통 heap에서의 삭제는 root노드를 삭제한다. Root를 삭제하고 나면 두 subtree가 연결이 안된 상태이므로 배열의 가장 마지막을 root로 이동시키고 아까 전의 reheapDown 구조를 통해서 다시 완성된 heap구조를 만들어준다.

<HeapTest.c>

① CUSTOMER구조체 선언과 사용될 함수들

```
typedef struct {
    char name[10];//고객 이름
    int priority;
    int point;
}CUSTOMER;

int compareCus(void* stu1, void* stu2);
void processPQ(HEAP* heap);
char menu(void);
CUSTOMER* getCus(void);
```

→ CUSTOMER구조체 멤버로 고객 이름(name)과 부여될 우선순위(priority, 값이 클수록 우선순위가 높다.), 그리고 우선순위로부터 계산될 point를 선언하였다.(우선순위가 높은 사람일수록 point가 높다.)

② main함수

```
int main(void) {

    HEAP* prQueue;

    printf("Begine Priority Queue Demonstration₩n");

    prQueue = heapCreate(maxQueue, compareCus);
    processPQ(prQueue);
    printf("End Priority Queue Demonstration₩n");
    return 0;
}
```

→ HEAP구조체 포인터인 prQueue를 선언하고 동적메모리 영역에서 할당받는다.

실제 이 프로그램은 processPQ함수를 통해 실행된다!

③ int compareCus(void* cus1, void* cus2)

```
int compareCus(void* cus1, void* cus2) {
    CUSTOMER c1;
    CUSTOMER c2;


    c1 = *(CUSTOMER*)cus1;
    c2 = *(CUSTOMER*)cus2;


    if (c1.point < c2.point)
        return -1;
    else if (c1.point == c2.point)
        return 0;
    return +1;
}
```

→compare함수를 통해 두 매개변수 값을 비교해보자.

→ CUSTOMER 구조체인 c1과 c2를 선언한다.

→매개변수로 받은 void포인터형 cus1과 cus2를 CUSTOMER구조체 포인터형으로 형변환을 하고 그게 가리키는 값이 구조체 c1,c2가 되도록 c1,c2를 초기화한다.

→만약 c1구조체의 point멤버 값이 c2구조체의 point멤버 값보다 작으면 -1, 같으면 0, 크면 1을 반환하도록 함수를 설정한다.

④ void processPQ(HEAP* prQueue)

-option이 'e'인 경우

```
void processPQ(HEAP* prQueue) {
    CUSTOMER* customer;
    bool result;
    char option;
    int numCustomer = 0;

    do {
        option = menu();
        switch(option) {
            case 'e':
                customer = getCus();
                numCustomer++;
                customer->point = customer->priority * 1000 + (1000 - numCustomer);

                printf("고객이름    Point₩n");
                printf("%s     %d₩n", customer->name, customer->point);
                result = heapInsert(prQueue, customer);

                if (!result) {
                    printf("Error inserting into heap₩n");
                    exit(101);
                }
                break;
```

→ 이제 priority heap구조의 작동중심이라 할 수 있는 processPQ함수를 분석해보자!

→ 조금 있다가 볼 menu()함수를 실행하고 난 return값으로 option변수를 초기화한다. 만약 option이 'e'였다면 getCus()함수를 실행한다. getCus()함수는 새로 삽입할 고객의 이름과 우선순위를 입력받고 customer 구조체에 그 값을 저장하여 그 구조체를 반환하는 함수이다.

→ getCus()함수로부터 새로운 정보가 입력된 customer구조체를 생성하였다. 따라서 numCusomer의 값을 1증가시켜주고 입력받은 priority값으로부터 point값을 계산하여 저장한다.(customer->point = customer->priority * 1000 + (1000 - numCustomer)) 옆의 계산식을 보면 priority가 클수록 point가 크다는 것을 알 수 있다. 또한 만약 priority값이 같을 경우 numCustomer에 따라 point값이 달라진다. 즉 우선순위가 같을 경우 먼저 줄을 선 사람이 더 큰 point값을 가진다.

→ 정보가 잘 저장되었는지 확인하기 위해서 저장된 정보를 출력하는 printf()함수를 써보았다.

→ 고객이름(name), 우선순위(priority), point가 저장된 customer구조체를 heap에 삽입해보자.

heapInsert(HEAP* heap, void* dataPtr)함수를 통해 prQueue HEAP구조체 포인터에 customer 구조체 포인터가 저장된다. 즉 heap->heapAry[heap->last]=dataPtr 문장을 통해 크기가 20인

heapAry배열의 마지막에 cutomer구조체가 삽입되고, reheapUp을 통하여 point가 큰 고객부터 heap구조로 배열이 정렬된다.

→즉 어떤 고객이 먼저 왔더라도(먼저 삽입됐더라도) point가 큰 고객이 먼저 입장하게 된다. 왜냐하면 reheapUp을 실행할 때 compareCus가 실행되는데, 이때 point값으로 배열들을 비교하기 때문이다.

→만약 heapInsert가 제대로 안이루어져 false를 반환했다면, 에러가 났다는 메시지와 함께 eixt(101)로 코드를 종료한다.

-option이 'd'인 경우

```
        case 'd':
            result = heapDelete(prQueue, (void**)&customer);
            if (!result)
                printf("Error:customer not found₩n");
            else {
                printf("Customer %4s deleted₩n", customer->name);
                numCustomer--;
            }
    }
} while (option != 'q');


system("PAUSE");
return;
```

→option값이 'd'이면 heapDelete함수가 실행된다. point크기순으로 정렬된 heapAry에서 root값이(point 값이 가장큰) 삭제된다.

→delete가 제대로 이루어지지 않으면, 에러메시지를 출력한다.

→만약 delete가 제대로 실행됐다면, 삭제된 customer 이름을 출력하면서 삭제가 잘 되었는지 확인한다. 그런 후 numCustomer수를 1감소시킨다.

→option이 'q'이면 do-while반복문을 종료한다.

⑤ char menu()

```
char menu() {
    char option;
    bool valid;

    printf("₩n============Menu============₩n");
    printf("e: Enter Customer Point₩n");
    printf("d: Delete Customer Point₩n");
    printf("q: Quit.₩n");
    printf("============================₩n");
    printf("Please enter your choice:  ");


    do {

        scanf("%c", &option);
        option = tolower(option);
        ClearLineFromReadBuffer();
        switch (option) {
        case 'e':
        case 'd':
        case 'q': valid = true;
            break;
        default: printf("Invalid choice. Re-Enter:  ");
            valid = false;
            break;
        }
```

→menu()함수는 return 값으로 opiton을 반환한다. option값이 'e', 'd', 'q'이면 valid 값으로 true를 반환하고 이외의 다른 것 문자이면 다시 입력하라는 메시지와 함께 valid는 false로 반환한다. Do-while 반복문은 while(!valid) 이므로 valid가 true이면 반복문이 종료된다.

⑥ CUSTOMER* getCus()

```
CUSTOMER* getCus() {
    CUSTOMER* customer;
    customer = (CUSTOMER*)malloc(sizeof(CUSTOMER));
    if (!customer) {
        printf("Memory overflow in getCustomer₩n");
        exit(200);
    }

    printf("Enter customer name:     ");
    scanf("%s", &customer->name);
    printf("Enter customer priority:  ");
    scanf("%d", &customer->priority);
    return customer;
}
```

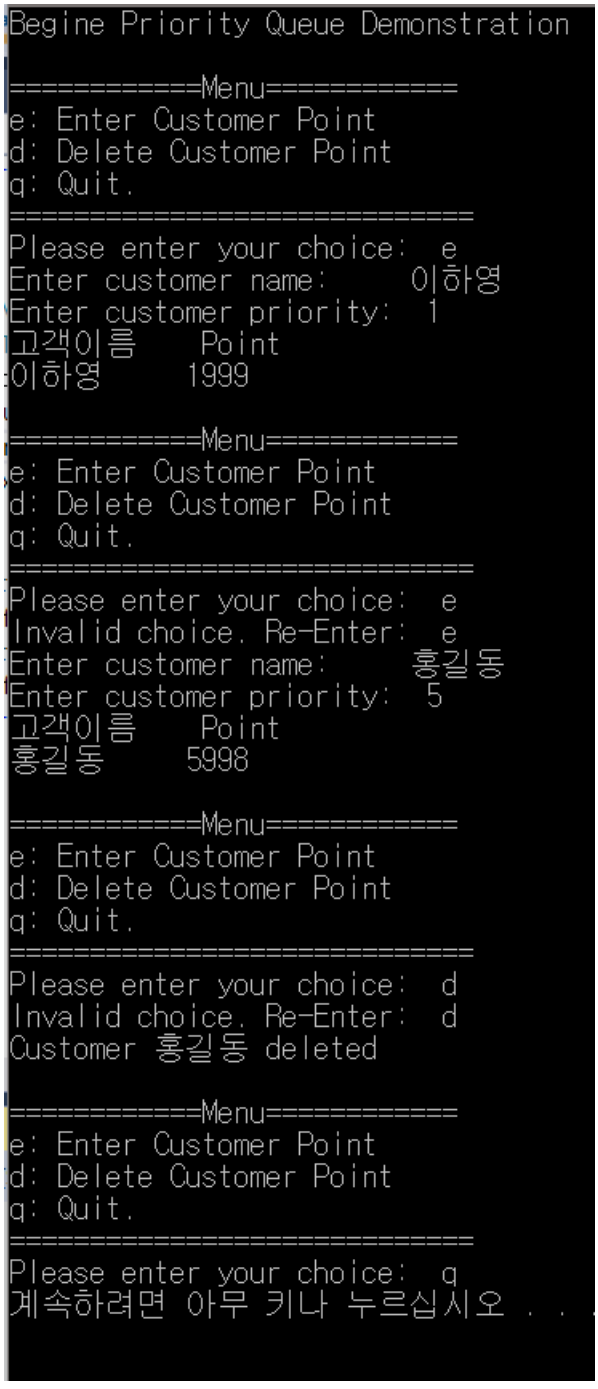
→getCus()는 processPQ에서 option값이 'e' 일 시 customer구조체에 고객정보를 입력받기 위한 함수이다.

→CUSTOMER 구조체 포인터를 동적할당 받고 만약 동적할당이 이루어지지 않았으면 overflow가 일어났다는 메시지를 출력한다. 그리고 eixt(200)으로 알고리즘을 종료한다.

→동적할당이 제대로 이루어졌으면 scanf함수를 통해 고객 이름과 우선순위 정보를 입력받고 customer 포인터 구조체를 반환한다.

3)실행 결과

<우선순위가 다른 경우>



→choice로 'e'를 입력하였다. 즉 option으로 'e'를 입력하였다. processPQ 함수대로라면 'e'가 입력 되었을 시 getCus()를 통해 customer의 고객정보를 입력 받는다. 옆에서 확인해 보면, 처음 고객 정보로 ("이하영", priority=1)을 입력하였다.

→Priority를 입력하면 processPQ함수에서 point가 계산된다. 고객이름과 고객point값을 알기 위해 printf함수를 통해 실행창에 출력하였다.

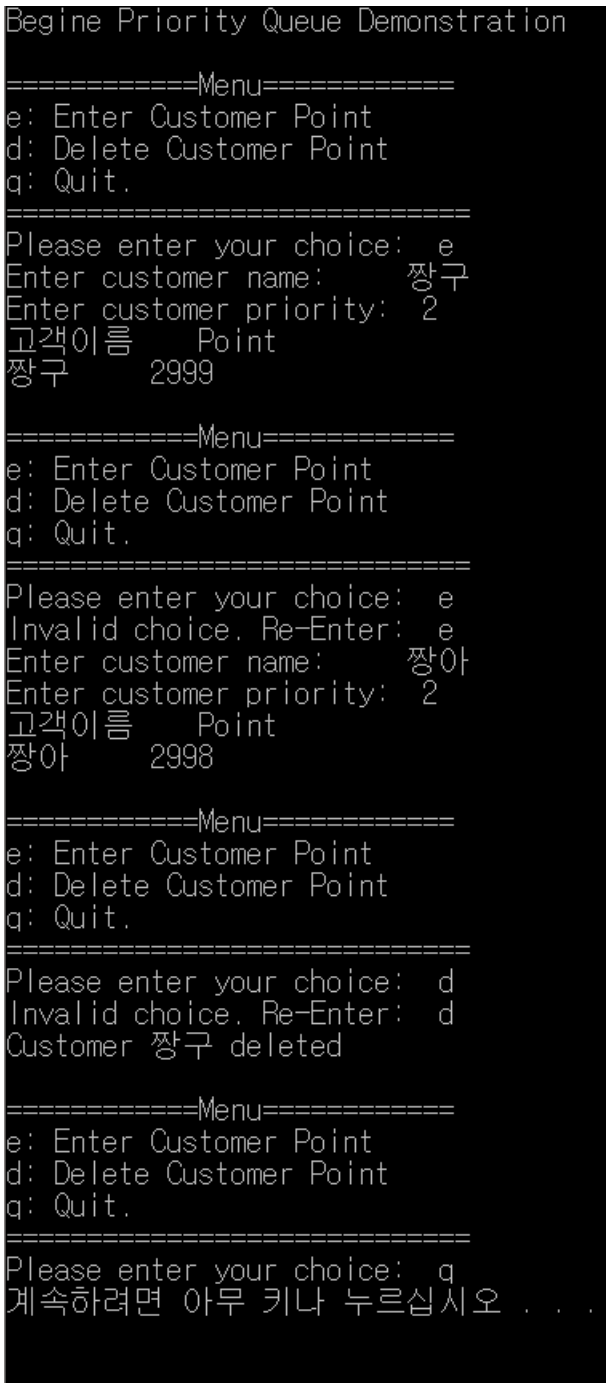
→두번째 do-while실행에서도 option값으로 'e'를 입력하고 "홍길동" 고객의 정보를 저장하였다.

→새로운 구조체가 삽입되었으므로 heapInsert, reheapUp, compareCus 함수를 통해 point값이 큰 고객이 heap구조의 배열에 우선순위로 정렬된다.

→즉 priority가 더 큰 "홍길동"이 "이하영"을 제치고 우선순위로 배열에서 정렬된다. 비록 "이하영" 고객이 먼저 왔지만(먼저 입력 받음) priority가 커서 point가 큰 "홍길동"고객이 먼저 줄을 서게 된다.

→실제 delete를 통해 data를 지워보면 root값(가장 큰 값) 이 삭제되는데 "홍길동"고객이 삭제된 것을 알 수있다.

<우선순위가 같은 경우>



→"짱구" 고객이 먼저 줄을 서고(먼저 입력되어서 배열에 저장된 경우) priority 값으로 2를 가진다면 계산된 point값은 '2999' 일 것이다.

→"짱아" 고객이 그 다음 순서로 들어오고 똑같이 priority로 2를 갖는다면 numCustomer이 "짱구"보다 하나 더 줄었으므로 point는 '2998'의 값을 갖는다.

→이렇게 priority가 같은 경우는 Queue와 같이 먼저 들어온 순서에 따라 값이 배열에 저장된다. 왜냐하면 reheapUp이 고객정보 중 point 값에 따라 일어나는데(앞에서도 설명했지만 reheapUp의 compareCus에서 point값으로 parent와 child 노드가 비교된다.) 먼저 들어온 고객이 point가 더 크기때문이다.

→확인해보기 위해서 heapDelete를 실행해 본다. 그 결과 "짱구" 고객이 delete된 것을 확인할 수 있다. 즉 point가 큰 "짱구"가 root에 저장돼 있었다는 것을 알 수 있다.

<느낀점>

Queue,List 와는 다른 Tree구조를 수업시간에 배웠는데 Tree구조에서 더 나아가 Heap구조를 이렇게 과제를 통해서 응용해보니 앞에서 배운 자료구조들과 차이를 확실히 알 수 있었습니다. 이번 과제는 Queue와는 다른 속성을 지닌 Heap구조로부터 우선순위로 data를 정렬하는 법을 배웠습니다. 처음 코드를 해석할 때 어려움도 있었지만, 3번에 걸친 ADT과제를 통해서 면역력..?이 생겼습니다. 이번학기 자료구조 수업을 들으면서 제일 성장한 부분은 포인터 변수에 대한 두려움이 조금 사라졌다는 것입니다. 1학년 때 C프로그래밍을 들으면서 배웠던 건 정말 작은 부분인 것을 깨달았습니다. 지금 배운 것도 빙산의 일각이겠지만 1학년때 보다는 성장한 것 같아서 기쁩니다!

자료구조 수업을 통해서 Stack, Queue, list, Tree, Heap구조의 응용과 이론 뿐만 아니라 C프로그래밍도 함께 배울 수 있어서 도움이 됐던 것 같습니다.