Assignment01

목차

1. Priority Queue의 자료 구조 형태 선정 배경

우선순위 큐(Priority Queue)는 배열을 사용하여 여러 가지의 자료 구조를 이용하여 만들 수 있다. 그 중에서 우선순위 큐는 주로 힙(Heap)의 구조 형태를 취하고 있다. Max Heap 혹은 Min Heap를 사용하여 최대 혹은 최소값이 항상 루트에 위치하게 하고, 삽입과 삭제가 모두 배열의 끝에서 이루어지게 한다는 점에서 힙의 자료 구조는 우선순위 큐에서 매우 효율적인 방법 중의 하나로 널리 알려져 있다. 하지만 이 힙의 자료 구조는 최대 혹은 최소 중 하나라는 값을 삽입 혹은 삭제하는 데 매우 효율적이며, 그에 반대되는 케이스(Min Heap의 경우 최댓값, Max Heap의 경우 최솟값) 혹은 중간 값과 같은 값들을 찾아서 반환하거나 삭제해야 하는 경우 정렬되지 않은 배열에서 값을 찾아내는 것과 같은 효과를 주게 되어 효율적이지 않다.

반대되는 케이스의 경우(Min Heap의 경우 최댓값, Max Heap의 경우 최솟값)에는 리프 노드들만을 탐색하여 해당되는 값을 찾으면 되므로, 전체 큐의 절반 크기만을 선형적으로 탐색하여 삭제 혹은 탐색에 linear한 시간이 걸린다. 하지만 중간 값을 구하기 위해서는 정렬되어 있지 않은 상태이기 때문에 모든 배열을 돌며 가장 작은 것, 그 다음 작은 것, … , 중간 값의 순으로 찾아야 하기에 $O(n^2)$의 시간 복잡도가 소요된다.

이로 인해 힙의 구조는 이러한 최대, 최소, 중간 값을 모두 찾아야 하는 경우에서는 적절하지 않은 자료 구조임을 판단하고, 이러한 작업을 하기 위해 사용하는 자료 구조로 정렬된 배열을 사용하기로 결정하였다.

2. main()

main() 함수에서는 기본적인 변수들을 선언 및 초기화하는 부분과 입력을 받는 부분, 해당 명령어에 따른 작업을 하는 부분, 마지막으로 결과를 출력해주는 부분, 총 4개의 큰 부분으로 나눌 수 있다.

```
17      // variables
18      int loopNum; // variable to decide how many times to iterate
19      char operationType, target; // operation type and target
20      int elementNum; // element to insert
21      int queueSize = 0; // variable for expressing size of priority queue; it means numbers of elements in Queue
22      int resultNum = 0; // variable for expressing size of result array; it means numbers of elements in resultPrint
23      int* Queue; // priority queue array(Max heap); queue starts at index 1 due to heap structure
24      int* temp; // temporary pointer to store the result of find operation
25      int** resultPrint; // result array to print
```

먼저 변수 초기화의 부분이다.

- loopNum 변수는 입력을 받을 횟수를 저장하는 변수이다.
- operationType 변수는 insert, find, delete 중 어느 연산을 진행할지 정하는 변수이다.
- target 변수는 최대, 최소, 중간값 중 어떤 값에 해당 연산을 수행할지를 정하는 변수이다.
- elementNum은 현재 우선순위 큐 안에 저장되어 있는 element의 수를 알려주는 변수이다.

- resultNum은 find 연산으로 인해 최종적으로 출력할 결과물들을 담은 배열의 크기를 알려주는 변수이다.
- Queue는 우선순위 큐의 역할을 해 주는 변수로, 오름차순으로 정렬되어 있는 배열이다.
- temp는 최종적으로 출력할 결과 하나하나를 포인팅하는 포인터이다. 비어있는 큐에서 탐색을 시도하였을 경우, 이 포인터 값은 NULL이 되며 아닐 경우 해당 탐색의 결과 값을 포인팅한다.
- resultPrint는 출력 결과들을 담고 있는 포인터를 저장하는 배열이다.

```
27      // get iteration times
28      scanf("%d", &loopNum);
29
30      //memory allocation
31      Queue = (int*) malloc(sizeof(int) * (loopNum + 1));
32      if (Queue == NULL) {
33          printf("Memory allocation failed\n");
34          exit(EXIT_FAILURE);
35      }
36      resultPrint = (int**) malloc(sizeof(int*) * loopNum);
37      if (resultPrint == NULL) {
38          printf("Memory allocation failed\n");
39          exit(EXIT_FAILURE);
40      }
```

선언 이후에 반복 횟수를 입력받고, 이후 그 크기만큼의 배열을 만들기 위해 동적 할당을 한다

다음으로는 입력을 받고 해당 연산을 처리하는 부분이다.

```
42      // loop "loopNum" times
43      for (int i = 0; i < loopNum; i++) {
44          scanf(" %c", &operationType);
45          if (operationType == 'I') { // Insert
46              scanf(" %d", &elementNum);
47              insert(elementNum, Queue, &queueSize);
48          }
49          else if (operationType == 'D') { // Delete
50              scanf(" %c", &target);
51              if (target == 'M') { // Delete minimun
52                  delete_min(Queue, &queueSize);
53              }
54              else if (target == 'X') { // Delete maximum
55                  delete_max(Queue, &queueSize);
56              }
57              else if (target == 'E') { // Delete median
58                  delete_median(Queue, &queueSize);
59              }
60              else printf("Invalid target\n");
61          }
```

```
62          else if(operationType == 'F') { // Find
63              scanf(" %c", &target);
64              if (target == 'M') { // Find minimum
65                  if (queueSize == 0) resultPrint[resultNum++] = NULL; // if Queue is empty, nothing is found
66                  else {
67                      temp = (int*) malloc(sizeof(int));
68                      if (temp == NULL) {
69                          printf("Memory allocation failed\n");
70                          exit(EXIT_FAILURE);
71                      }
72                      *temp = find_min(Queue, queueSize);
73                      resultPrint[resultNum++] = temp;
74                  }
75              }
76              else if (target == 'X') { // Find maximum
77                  if (queueSize == 0) resultPrint[resultNum++] = NULL; // if Queue is empty, nothing is found
78                  else {
79                      temp = (int*) malloc(sizeof(int));
80                      if (temp == NULL) {
81                          printf("Memory allocation failed\n");
82                          exit(EXIT_FAILURE);
83                      }
84                      *temp = find_max(Queue, queueSize);
85                      resultPrint[resultNum++] = temp;
86                  }
87              }
88              else if (target == 'E') { // Find median
89                  if (queueSize == 0) resultPrint[resultNum++] = NULL; // if Queue is empty, nothing is found
90                  else {
91                      temp = (int*) malloc(sizeof(int));
92                      if (temp == NULL) {
93                          printf("Memory allocation failed\n");
94                          exit(EXIT_FAILURE);
95                      }
96                      *temp = find_median(Queue, queueSize);
97                      resultPrint[resultNum++] = temp;
98                  }
99              }
100             else printf("Invalid target\n");
101         }
102         else printf("Invalid operation type\n");
```

먼저 첫 입력을 받아 어떤 연산을 수행할지를 정한다. 이후 그 안에서 입력을 한 번 더 받아, 최대와 최소, 중간값 중 어떠한 값에 대해 연산을 수행할 것인지 정한 이후 해당 연산을 수행한다. 이 이외의 경우에는 유효하지 않은 연산 유형이거나 타겟임을 프린트하도록 하였다. 여기서 find의 경우 아예 비어있는 우선순위 큐에서 탐색을 시도하려 할 경우, find 함수들을 아예 실행하지 않고 바로 NULL포인터를 resultPrint 배열의 다음 인덱스로 연결하여 주는 방식으로 프로그램을 설계하였다.

```
112     // printing result
113     for (int i = 0; i < resultNum; i++) {
114         if (resultPrint[i] == NULL) printf("NULL\n"); // print NULL if nothing found
115         else {
116             printf("%d\n", *(resultPrint[i]));
117             free(resultPrint[i]); // memory deallocation
118         }
119     }
120
121     // memory deallocation
122     free(Queue);
123     free(resultPrint);
124
125     return 0;
126 }
```

다음으로는 결과를 표준 입출력으로 출력하는 부분이다. 배열을 돌아가면서 값을 확인하고, NULL포인터일 경우 NULL을 출력하고 아닐 경우 해당 포인터가 참조하고 있는 값, 즉 탐색의 결과를 출력한다. 이후 모든 메모리를 해제하며 프로그램을 종료한다.

3. insert()

```
129  // functions
130  void insert(int element, int Queue[], int* n) { // Time complexity of O(n); due to shifting operations
131      // variables initialization
132      int i = (*n)++;
133
134      // sorting priority Queue; assume Queue is already sorted before insert element
135      // then we have to find appropriate position and shift elements to left
136
137      for (; (i > 0) && (Queue[i - 1] > element); i--) { // if find appropriate position or i becomes 0, loop will stop
138          Queue[i] = Queue[i - 1]; // shift elements to right
139      }
140      Queue[i] = element;
141  }
```

insert 함수의 알고리즘은 다음과 같다. 이미 정렬이 완료된 우선순위 큐인 Queue와 그 우선순위 큐의 크기 n이 들어오면, 배열의 맨 뒤에서부터 element와 값을 비교해 가며 element가 들어갈 자리를 찾는다. 하나의 element가 추가로 들어오므로 n의 값을 하나 늘려준다. 이후 만약 현재 위치의 바로 앞 인덱스 값이 element보다 크다면 그 앞의 값을 현재 위치로 한 칸 shift한 후 현재 위치를 그 바로 앞의 자리로 수정하는 과정을 반복한다. 그 위치가 배열의 맨 앞이 되거나, 아니면 element가 현재 위치의 값의 바로 앞 인덱스 값보다 클 경우 반복을 멈추고 해당 위치에 element를 삽입하며 종료한다.

이 삽입 작업의 경우 Worst case는 모든 배열의 요소들을 탐색하고 shift한 다음 맨 앞에 해당 element를 삽입하는 과정에 해당하는데, 이는 모든 요소들을 한번만 탐색하고 지나간 것과 동일하므로 선형의 시간에 비례한다. 따라서 이 삽입의 경우 시간 복잡도는 우선순위 큐 안의 요소들의 개수 n에 대하여 O(n)에 해당한다.

4. delete

Delete 연산의 경우 알고리즘은 크게 비슷한 형태를 띄고 있다. 우선순위 큐 안의 요소가 아무것도 없는 빈 배열의 경우 아무 작업도 하지 않고 바로 함수를 종료한다. 그 외의 경우는 우선순위 큐가 배열이기 때문에 해당 배열의 요소를 인덱스를 통해서 바로 접근할 수 있다는 점을 이용하여, 해당 인덱스의 값을 뽑은 후 빈 공간을 shift 작업을 통해 비어있는 부분을 다시 채우는 방식으로 이루어진다.

4.I. delete_min()

```
143  int delete_min(int Queue[], int* n) { // Time complexity of O(n); due to shifting operations
144      // do nothing if Queue is empty
145      if ((*n) == 0) return -1;
146
147      // variables initialization
148      int minValue = Queue[0];
149
150      // shifting
151      for (int i = 0; i < (*n) - 1; i++) { // start at the point of minimum element
152          Queue[i] = Queue[i + 1]; // shifting value to the right
153      }
154      (*n)--; // decrease n
155
156      return minValue;
157  }
```

최솟값을 삭제하는 과정은 다음과 같다. 비어있는 큐일 경우 진행하지 않고 함수를 종료하며, 그 외의 경우에는 가장 최솟값은 배열의 맨 앞에 위치하기 때문에 이 값을 삭제하면 된다. 이후 배열의 맨 앞 인덱스가 비어있기 때문에, 이 부분을 삭제한 이후 모든 배열의 요소들을 한 칸씩 당겨준 후 배열의 크기를 하나씩 줄이며 함수를 종료한다.

이 삭제 작업의 경우 탐색에는 바로 배열의 인덱스를 가지고 접근하면 되기에 O(1)의 시간이 소요되지만, shift 과정으로 인해 모든 배열의 요소들을 한 번씩 탐색해야 하는 과정이 필요하게 된다. 따라서 이 shift의 과정에서 O(n)의 시간 복잡도가 걸리게 된다. 따라서 최솟값 삭제 함수의 총 시간 복잡도는 worst case, best case의 경우에 상관없이 우선순위 큐 안의 요소들의 개수 n에 대하여 O(n)에 속하게 된다.

4.II. delete_max()

```
159  int delete_max(int Queue[], int* n) { // Time complexity of O(1)
160      // do nothing if Queue is empty
161      if ((*n) == 0) return -1;
162
163      // variables initialization
164      int maxValue = Queue[--(*n)]; // decrease n
165
166      return maxValue;
```

최댓값을 삭제하는 경우는 다음과 같다. 큐가 비어있을 경우 아무 작업도 진행하지 않으며, 그렇지 않을 경우 오름차순으로 큐가 이미 정렬되어 있기에 항상 배열의 마지막 인덱스에 최댓값이 저장되어 있다. 따라서 이 값을 삭제하기 위해 배열의 크기인 n을 1 줄여주기만 하면 된다. 따라서 최댓값 삭제 함수는 O(1)의 시간 복잡도를 가지게 된다.

4.III. delete_median()

```
169 int delete_median(int Queue[], int* n) { // Time complexity of O(n); due to shifting operations
170     // do nothing if Queue is empty
171     if ((*n) == 0) return -1;
172 
173     // variables initialization
174     int medianValue = Queue[((*n) - 1) / 2];
175 
176     // shifting
177     for (int i = ((*n) - 1) / 2; i < (*n) - 1; i++) { // start at the point of median element
178         Queue[i] = Queue[i + 1]; // shifting value to the right
179     }
180     (*n)--; // decrease n
```

중간값을 삭제하는 함수의 개요는 다음과 같다. 비어있는 큐가 들어왔을 경우 아무 작업도 진행하지 않으며, 그렇지 않은 경우 큐는 이미 정렬되어 있기 때문에 중간값은 항상 배열의 크기의 절반을 한 인덱스에 있을 것이다. 배열의 중간값이 2개일 경우 더 작은 값을 취하기로 되어 있으므로, 배열의 크기를 n이라 할 때 중간값의 인덱스 공식은 $\lfloor n / 2 \rfloor$와 같다. 따라서 해당 인덱스를 제거하고, 그 이후의 모든 요소들을 한 칸씩 shift를 하면서 삭제 작업을 완료한다.

이 함수의 경우 중간값 자체를 찾는 것은 인덱스를 통해 한 번에 접근하므로 O(1)의 시간 복잡도가 소요되지만 중간 이후의 모든 인덱스를 한 번씩 탐색하며 shift 작업을 수행해야 하므로, shift 작업에 대한 시간 복잡도는 배열의 크기 n에 대하여 O(n)에 속하게 된다. 따라서 중간값 삭제 함수 전체의 총 시간 복잡도는 배열의 크기 n에 대하여 O(n)의 시간 복잡도를 갖게 된다.

5. find

Find 함수는 모두 간단한 방식으로 동작한다. 메인 함수에서 비어있지 않은 큐에 대해서만 이 find 함수들을 호출하도록 설계하였으므로, 이 find 함수들은 비지 않은 배열을 받는다는 전제 하에 작업을 수행한다. 또한 delete 함수에서 설명하였듯이, 배열이 이미 정렬되어 있으므로 인덱스를 통한 직접적인 접근이 가능하다. 따라서 모든 find함수의 시간 복잡도는 주어진 배열 크기에 상관없이 항상 O(1)의 시간 복잡도를 가지게 된다.

5.I. find_min()

```
185 int find_min(int Queue[], int n) { // Time complexity of O(1)
186     return Queue[0];
187 }
```

오름차순으로 정렬되어 있기에 맨 앞, 즉 0번째 인덱스에 최솟값이 저장되어 있으므로 해당 인덱스의 값을 반환하며 함수가 종료된다.

5.II. find_max()

```
189   int find_max(int Queue[], int n) { // Time complexity of O(1)
190       return Queue[n - 1];
191   }
```

오름차순으로 정렬되어 있기에 맨 뒤, 즉 (n – 1)번째 인덱스에 최솟값이 저장되어 있으므로 해당 인덱스의 값을 반환하며 함수가 종료된다.

5.III. find_median()

```
193   int find_median(int Queue[], int n) { // Time complexity of O(1)
194       return Queue[(n - 1) / 2];
195   }
```

큐는 이미 오름차순으로 정렬되어 있기 때문에 중간값은 항상 배열의 크기의 절반을 한 인덱스에 있을 것이다. 배열의 중간값이 2개일 경우 더 작은 값을 취하기로 되어 있으므로, 배열의 크기를 n이라 할 때 중간값의 인덱스 공식은 ⌊n / 2⌋와 같다. 따라서 해당 인덱스의 값을 반환하며 함수를 종료한다.