데이터 구조 Assn4

20180551 이준석

POVIS ID : ljs9904ljs

Problem 1 (R-6.1)

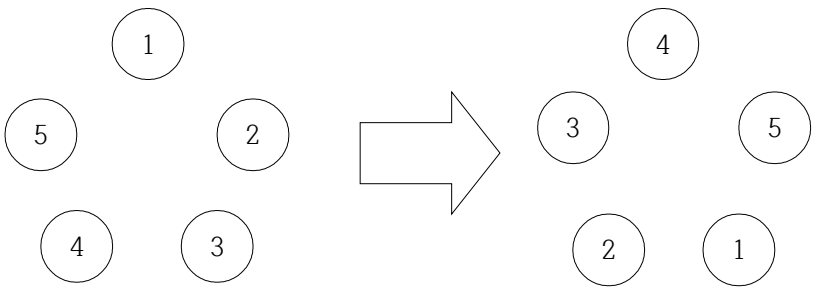
```
int arr[10];
int i;
STACK* s = (STACK*)malloc(sizeof(STACK));//LIFO인 스택을 이용한다.

for (i = 0; i < 10; i++) {
        arr[i] = i;
}
for (i = 0; i < 10; i++) {
        push(s, arr[i]); //배열에 저장되어 있는 값을 스택에 전부 복사
}
for (i = 0; i < 10; i++) {
        pop(s, arr[i]); //스택에 저장되어 있는 값을 배열의 처음부터 저장
}
```

스택은 마지막에 저장된 원소가 가장 첫 번째로 나간다는 특성을 갖고 있다. 그 점을 이용하여 스택에 배열의 모든 원소를 저장해두고 다시 스택의 모든 원소를 배열의 첫 칸부터 저장하면 배열을 역순으로 만들 수 있다.

Problem 2 (R-6.3)

```
for (i = 0; i < n; i++) { //원형 링크드 리스트들을 생성해서 값 저장
        add(c1, i);
        add(c2, 0);
}
for (i = 0; i < (n - d - 1); i++){ //cursor를 (n-d-1)칸만큼 이동시킨다.
        c1->cursor = c1->cursor->next;
}
for (i = 0; i < n; i++) { //cursor의 앞에 있는 것들을 하나씩 제거하며 배열에 저장
        remove(c1, arr[i]);
}
for (i = 0; i < n; i++) { //배열에 저장되어 있는 값을 원형 리스트에 옮긴다.
        c2->cursor->data = arr[i];
        c2->cursor = c2->cursor->next;
}
```

cursor는 원형 리스트에서 현재 가리키고 있는 node 이다. 처음으로 생성된 node를 가리키고 있다.
add는 cursor의 앞에 node를 추가하는 함수이다.
remove는 cursor의 앞의 node를 제거하는 함수이다.



시계 방향으로 회전하는 원형 리스트에서 d = 2 일 때, 위의 그림처럼 바뀌어야 한다. 그러기 위해서는 cursor를 n-d만큼 이동 시킨 뒤(위의 코드에서는 cursor의 앞에서 remove되므로 n-d-1칸 이동시켰다.) 거기서부터 시계 방향으로 값들을 읽어 나가면서 배열에 임시로 저장해두어야 한다. 그리고 그것을 다시 원형 리스트에 원래 1이 저장되어 있던 node부터 옮겨 저장함으로써 d칸만큼 회전을 시킨다.

Problem 3 (C-6.9)

```
void insertFront(int e) {
        static NODE* positionFront;

        if (empty) {
                NODE* firstNode = new NODE;
                firstNode->data = e;
                positionFront = firstNode;
        }
        else {
                insert(positionFront, e);
                --positionFront;
        }
}
void insertBack(int e) {
        static NODE* positionBack;

        if (empty) {
                NODE* firstNode = new NODE;
                firstNode->data = e;
                positionBack = firstNode;
        }
        else {
                ++postionBack;
                insert(positionBack, e);
        }
}
```

empty는 list가 비어있다면 1을 return하고 node가 1개 이상 존재한다면 0을 return하는 함수이다.

insert는 position과 element를 parameter로 받아서 position의 전에(앞에) node를 삽입하는 함수이다.

positionFront와 positionBack은 node의 위치를 저장하는 변수이다. 매번 함수가 실행될 때마다 위치를 기반으로 node를 삽입하므로 static 변수로 설정하여 함수가 사라져도 저장해놓은 위치가 사라지지 않도록 한다.

insertFront와 insertBack에서 첫 node는 무조건 insert의 사용이 아닌 개별적인 할당이 필요하다. 왜냐하면 insert가 원래 위치를 입력받아서 그 전에 삽입하는 함수이기 때문이다. 첫 node를 개별적으로 생성한 뒤에 그것 앞뒤로 insert를 해야만 한다. 그래서 두 함수 모두 empty == 1 일 때 첫 node를 생성하고 그것의 위치를 positionFront 혹은 positionBack에 저장한다. 다음에 각각의 함수를 호출할 때는 empty == 0 이므로 Front일때는 position의 전에 삽입 후 position을 왼쪽으로 이동시키고 Back일때는 position을 오른쪽으로 이동시키고 삽입한다. 이렇게하면 empty와 insert만으로 insertFront와 insertBack을 구현할 수 있다.

Problem 4 (C-6.15)

An array is *sparse* if most of its entries are NULL. A list L can be used to implement such an array, A, efficiently. In particular, for each nonnull cell $A[i]$, we can store an entry (i,e) in L, where e is the element stored at $A[i]$. This approach allows us to represent A using $O(m)$ storage, where m is the number of nonnull entries in A. Describe and analyze efficient ways of performing the functions of the vector ADT on such a representation.

데이터가 저장되어 있지 않은 칸이 많은 배열은 기록되어 있는 데이터에 비해 메모리 공간 낭비가 크다. 그리고 만약 이 배열에서 특정한 원소를 찾으려 한다면 수없이 많은 빈칸을 검사하는 비효율적인 방식을 취해야 한다. 이러한 낭비를 줄이기 위해서는 데이터들을 찾아서 따로 기록해두고 사용하는 것이 효율적인 방식일 것이다.

문제의 질문은 이러한 방식으로 vector ADT의 함수들을 어떻게 구현할 수 있겠느냐는 것이다. 먼저 vector에 저장되어 있는 데이터들을 빈 칸 없이 모으는 과정이 선행되어야 한다. 왜냐하면 이 과정을 거치지 않고 vector의 함수들을 사용한다는 것은 원래의 비효율적인 방식으로 사용한다는 뜻이 되기 때문이다.

총 n개의 칸을 갖고, m개의 데이터가 있는 칸을 갖는 vector를 생각해보자.

모으는 과정의 시간복잡도를 생각해보면, n칸을 탐색하며 값이 있을 때마다 다른 곳에 저장해두는 방식으로 모으는 것이기 때문에 모으는 과정은 시간복잡도가 O(n)이다.

vector ADT는 at(i), set(i, e), insert(i, e), erase(i)가 있다. at과 set은 원래의 시간복잡도가 O(1)이다. 따라서 메모리가 지나치게 부족한 것이 아니라면 모으는 과정을 거치는 것이 오히려 비효율적인 방식이다. 하지만 insert와 erase는 시간복잡도가 O(n)이다. 그러므로 삽입과 삭제를 여러 번 한다면 모으는 과정을 미리 거쳐놓는 것이 시간복잡도가 O(m)이 되므로 효율적이다.

정리하자면 at과 set만을 사용할 vector에서는 모으는 과정을 거치지 않는 것이 더 효율적이다. 그러나 insert와 erase도 사용할 vector라면 미리 한 번 모으는 과정을 거치는 것이 효율적이다. 단, insert 혹은 erase를 단 한 번만 할 것이라면 모으는 과정이 O(n)이고 insert나 erase도 O(n)이기 때문에 모으지 않는 것이 더 효율적이다.

Problem 5 (C-6.18)

A에 포함되어 있는 (x, y)의 y와 B에 포함되어 있는 (y, z)의 y가 같음을 이용해서 (x, y, z)를 만들어야 한다. 결국 A의 원소와 같은 원소가 B에 있는지를 찾아야하는 문제로 귀결된다. 즉, 탐색에 있어서 가장 효율적인 알고리즘을 찾아내야 한다.

일반적으로 순차 탐색과 이진 탐색 두 가지의 방법이 존재하는데, 순차 탐색은 O(n)이고 이진 탐색은 O(logn)이다. 그런데 A의 모든 원소를 하나하나 가져오는 과정은 O(n)이므로 순차 탐색과 섞어 쓰면 O(n^2)이 되어 버린다. 따라서 O(nlogn)인 합병 정렬이나 퀵 정렬을 이용하여 B를 정렬한 뒤에 이진 탐색을 사용하여 탐색하는 것이 효율적인 알고리즘이 될 것이다. 그러면서 공통적인 y를 갖는 A와 B의 원소가 출현한다면 그것들을 (x, y, z)로 합쳐주면 된다.

정리하자면,

-정렬할 때 O(nlogn)

-원소를 하나하나 가져올 때 O(n)과 이진 탐색의 O(logn)이 동시에 이루어지므로 O(nlogn)

둘을 합쳐서 총 시간복잡도는 O(nlogn)이 된다.