JooHyun – Lee (comkiwer)

TS박테리아

Hancom Education Co. Ltd.

Problem

Problem

TS박테리아

TS연구소에 박테리아를 보관하고 반출하는 프로그램을 만들려고 한다.

연구소에 저장하는 박테리아는 N 가지 종류가 있다.

박테리아에는 생명력이 있으며, 보관을 시작하는 시각부터 halfTime 시간이 경과할 때 마다 생명력이 절반으로 줄어든다.

생명력에 소수점이 생기면 소수점은 버린다.

예를 들어 생명력이 100.5 일 경우 생명력은 100이 된다.

박테리아의 생명력이 9 이하가 되면 해당 박테리아를 폐기 처리된다.

박테리아 반출이 요청되면 박테리아 중에서 생명력이 가장 작은 박테리아가 반출된다.

Problem

연구소에는 매 시간 다음과 같은 일이 반복된다.

1. 연구소에 있는 박테리아가 보관을 시작하는 시각부터
   `halfTime` 시간이 경과할 때 마다 생명력이 절반으로 줄어든다.
2. 생명력이 `9` 이하인 박테리아는 폐기 처리된다.
3. 새로운 박테리아가 보관되거나 연구소에 있는 박테리아를 반출한다.

이러한 연구소를 관리하는 프로그램을 작성하자.

아래 `API` 설명을 참조하여 각 함수를 구현하라.

Problem

1. **void init(int N, char nameList[][], int halfTime[]) :**

   각 테스트 케이스의 처음에 호출된다.

   처음 연구소는 비어있고, 현재 시각은 `0` **이다.**

   연구소에는 `N` **개 종류의 박테리아를 보관할 수 있다.**

   박테리아의 이름은 `nameList[]` **이고,**
   박테리아의 생명력이 절반으로 줄어드는 시간은 각각 `halfTime[]` **이다.**

   박테리아의 이름은 `nameList[0], nameList[1], …, nameList[N - 1]` **이고,**
   모두 서로 다른 이름이다.

   박테리아의 생명력이 절반이 되는 시간은
   `halfTime[0], halfTime[1], …, halfTime[N - 1]` **이다.**

Problem

1. **void init(int N, char nameList[][], int halfTime[]) : 계속**

예를 들어, 0번째 박테리아의 이름은 nameList[0]이고,
생명력이 절반으로 줄어드는 시간은 halfTime[0]이다.

nameList[] 은 길이가 3 이상 9 이하의 영문 소문자이고,
'\0' 문자로 끝나는 문자열이다.

Parameters

N : 연구소에 보관하는 박테리아의 종류 (1 ≤ N ≤ 100)

nameList[] : 박테리아의 이름 (3 ≤ nameList[]의 길이 ≤ 9)

halfTime[] : 박테리아의 생명력이 절반이 되는 시간 (10 ≤ halfTime[] ≤ 10,000)

Problem

2. **void keepBacteria(int stamp, char name[], int life, int count) :**

stamp 시각에 life 생명력의 name 박테리아를 count 개 보관한다.

name 박테리아의 생명력은 stamp 시각에 life 이고, 이후 halfTime 시간마다 현재 생명력이 절반으로 감소한다.

예를 들어, 171 생명력 박테리아의 halfTime 값이 10 이면,

stamp 시각부터 10 시간마다 생명력은 171 → 85 → 42 → 21 → 10 → 5 되고, 생명력이 5 되면 박테리아는 폐기된다.

name 은 길이가 3 이상 9 이하의 영문 소문자이고, '\0' 문자로 끝나는 문자열이다.

Parameters

- stamp : 박테리아를 연구소에 보관하는 시각 (1 ≤ stamp ≤ 1,000,000)
- name[] : 박테리아의 이름 (3 ≤ name의 길이 ≤ 9)
- life : 박테리아의 생명력 (100 ≤ life ≤ 50,000)
- count : 박테리아의 개수 (1 ≤ count ≤ 1,000)

Problem

3. **int takeOut(int stamp, int count) :**

stamp 시각에 연구소에서 생명력이 가장 작은 순서로 박테리아 count 개 반출하고,
반출한 박테리아의 생명력 합을 반환한다.
만약 박테리아의 보관 개수가 count 개 미만인 경우 보관중인 박테리아를 모두 반출한다.
생명력이 같은 박테리아가 있을 경우 보관 시각이 먼저인 박테리아가 우선 반출된다.
stamp 시각에 생명력이 9 이하의 박테리아는 폐기 처분되어 포함되지 않는다.
함수 호출 시, 연구소에 박테리아가 1개 이상 있음이 보장된다.

Parameters

stamp : 박테리아의 반출 시각 (1 ≤ stamp ≤ 1,000,000)
count : 박테리아의 반출 개수 (1 ≤ count ≤ 100)

Returns

반출하는 박테리아의 생명력 합

Problem

4. **int getBacteria(int stamp, char name[]) :**

stamp 시각에 연구소에 있는 name 박테리아의 개수를 반환한다.

stamp 시각에 생명력이 9 이하의 박테리아는 폐기 처분되어 포함되지 않는다.

연구소에 name 박테리아의 개수가 0 일수도 있다.

Parameters

stamp : 연구소에 있는 name 박테리아의 개수를 확인하는 시각 (1 ≤ stamp ≤ 1,000,000)

name[] : 박테리아의 이름 (3 ≤ name의 길이 ≤ 9)

Returns

연구소에 있는 name 박테리아의 개수

Problem

TS박테리아

[제약사항]

1. 각 테스트 케이스 시작 시 `init()` 함수가 호출된다.
2. 미생물의 이름은 길이가 `3` 이상 `9` 이하의 영문 소문자이고,
   '`\0`' 문자로 끝나는 문자열이다.
3. 각 테스트 케이스에서 보관되는 미생물의 종류는 최대 `100` 개이다.
4. 각 테스트 케이스에서 `stamp` 값은 항상 증가하는 값이다.
5. 각 테스트 케이스에서 `getBacteria()` 함수의 호출 횟수는 최대 `15,000` 이다.
6. 각 테스트 케이스에서 `takeOut()` 함수의 호출 횟수는 최대 `15,000` 이다.
7. 각 테스트 케이스에서 모든 함수의 호출 횟수 합은 최대 `50,000` 이다.

Problem analysis

Problem analysis : 예제

Order	Function	Note	Figure
1	init(3, {“aaa”, “bbb”, “ccc”}, {55, 44, 33})		
2	keepBacteria(10, “aaa”, 110, 3)		
3	keepBacteria(12, “bbb”, 120, 2)		
4	keepBacteria(19, “ccc”, 160, 4)		
5	keepBacteria(20, “aaa”, 105, 2)		
6	keepBacteria(25, “bbb”, 109, 3)		
7	takeOut(120,1)	return 20	**[Fig. 1]**
8	takeOut(150, 1)	return 15	
9	takeOut(151,1)	return 10	**[Fig. 2]**
10	getBacteria(184, “ccc”)	return 0	
11	keepBacteria(200, “aaa”, 300, 1)		
12	keepBacteria(201, “bbb”, 310, 1)		
13	keepBacteria(202, “ccc”, 153, 2)		
14	keepBacteria(203, “ccc”, 140, 2)		**[Fig. 3]**
15	takeOut(310,3)	return 53	
16	getBacteria(311, “aaa”)	return 1	
17	getBacteria(312, “ccc”)	return 1	**[Fig. 4]**

[Table 1]

[Fig. 1] ~ [Fig. 4]는 보관소에 있는 박테리아를 보여준다. (괄호 안의 숫자는 박테리아의 생명력이다.)

Order 7. takeOut(120, 1) 에서 생명력이 20으로 제일 낮은 “ccc” 박테리아 1개가 반출된다. ([Fig. 1] 빨간색 네모)

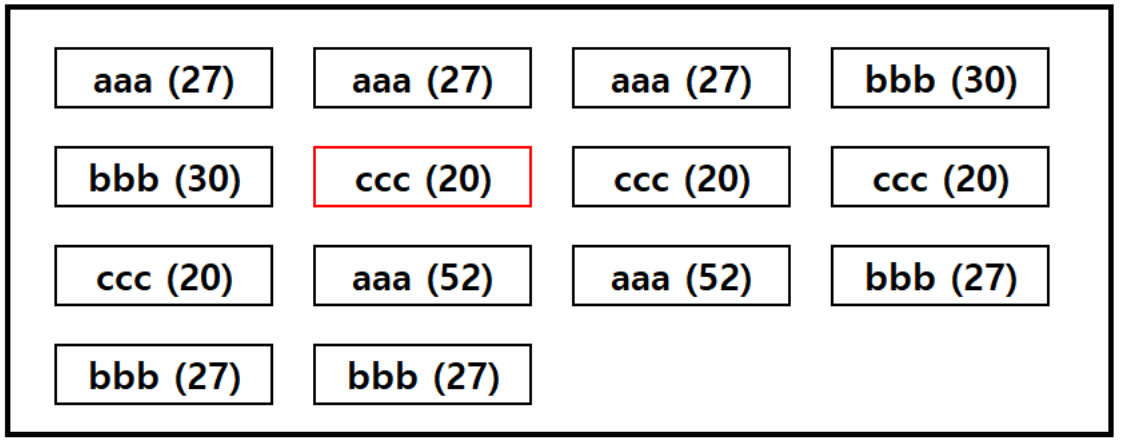

Fig. 1

Problem analysis : 예제

Order	Function	Note	Figure
1	init(3, {“aaa”, “bbb”, “ccc”}, {55, 44, 33})		
2	keepBacteria(10, “aaa”, 110, 3)		
3	keepBacteria(12, “bbb”, 120, 2)		
4	keepBacteria(19, “ccc”, 160, 4)		
5	keepBacteria(20, “aaa”, 105, 2)		
6	keepBacteria(25, “bbb”, 109, 3)		
7	takeOut(120,1)	return 20	**[Fig. 1]**
8	takeOut(150, 1)	return 15	
9	takeOut(151,1)	return 10	**[Fig. 2]**
10	getBacteria(184, “ccc”)	return 0	
11	keepBacteria(200, “aaa”, 300, 1)		
12	keepBacteria(201, “bbb”, 310, 1)		
13	keepBacteria(202, “ccc”, 153, 2)		
14	keepBacteria(203, “ccc”, 140, 2)		**[Fig. 3]**
15	takeOut(310,3)	return 53	
16	getBacteria(311, “aaa”)	return 1	
17	getBacteria(312, “ccc”)	return 1	**[Fig. 4]**

[Table 1]

Order 9. takeOut(151, 1) **에서 생명력이** 10**으로 제일 낮은** “ccc” **박테리아** 1**개가 반출된다.** ([Fig. 2] **빨간색 네모**)

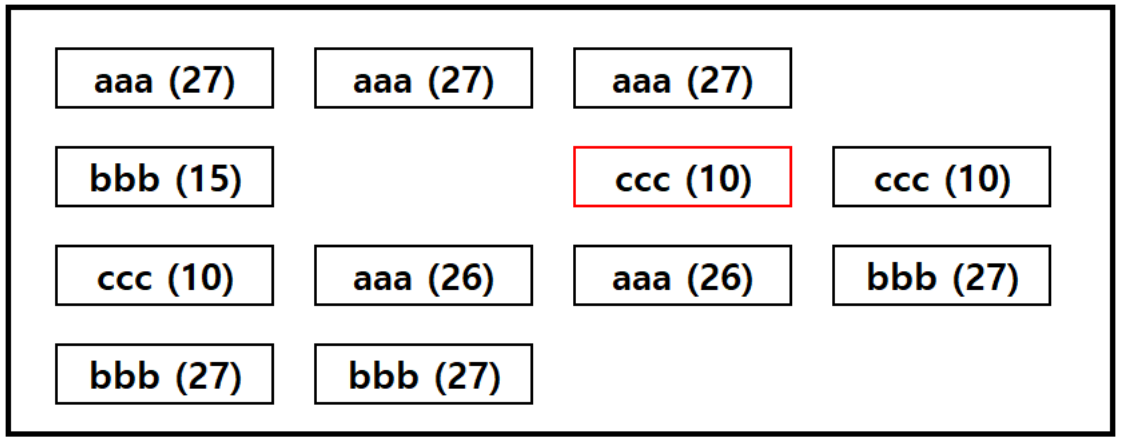

Fig. 2

Problem analysis : 예제

Order	Function	Note	Figure
1	init(3, {“aaa”, “bbb”, “ccc”}, {55, 44, 33})		
2	keepBacteria(10, “aaa”, 110, 3)		
3	keepBacteria(12, “bbb”, 120, 2)		
4	keepBacteria(19, “ccc”, 160, 4)		
5	keepBacteria(20, “aaa”, 105, 2)		
6	keepBacteria(25, “bbb”, 109, 3)		
7	takeOut(120,1)	return 20	[Fig. 1]
8	takeOut(150, 1)	return 15	
9	takeOut(151,1)	return 10	[Fig. 2]
10	getBacteria(184, “ccc”)	return 0	
11	keepBacteria(200, “aaa”, 300, 1)		
12	keepBacteria(201, “bbb”, 310, 1)		
13	keepBacteria(202, “ccc”, 153, 2)		
14	keepBacteria(203, “ccc”, 140, 2)		[Fig. 3]
15	takeOut(310,3)	return 53	
16	getBacteria(311, “aaa”)	return 1	
17	getBacteria(312, “ccc”)	return 1	[Fig. 4]

[Table 1]

Order 14. keepBacteria(203, “ccc”, 140, 2) 에서 생명력이 140 인 “ccc” 박테리아 2개가 보관소에 추가된다.
([Fig. 2] 노란색 네모)

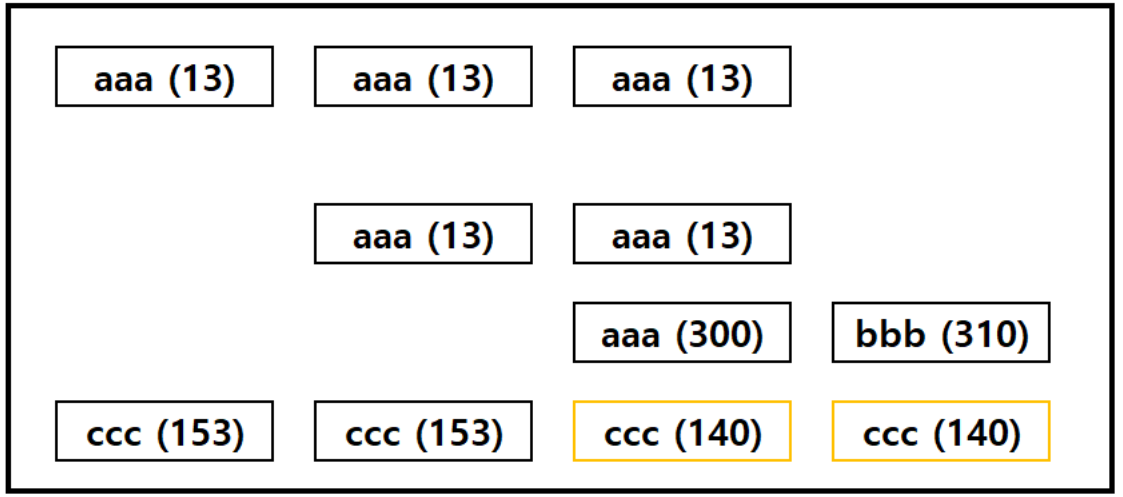

Fig. 3

Problem analysis : 예제

Order	Function	Note	Figure
1	init(3, {“aaa”, “bbb”, “ccc”}, {55, 44, 33})		
2	keepBacteria(10, “aaa”, 110, 3)		
3	keepBacteria(12, “bbb”, 120, 2)		
4	keepBacteria(19, “ccc”, 160, 4)		
5	keepBacteria(20, “aaa”, 105, 2)		
6	keepBacteria(25, “bbb”, 109, 3)		
7	takeOut(120,1)	return 20	**[Fig. 1]**
8	takeOut(150, 1)	return 15	
9	takeOut(151,1)	return 10	**[Fig. 2]**
10	getBacteria(184, “ccc”)	return 0	
11	keepBacteria(200, “aaa”, 300, 1)		
12	keepBacteria(201, “bbb”, 310, 1)		
13	keepBacteria(202, “ccc”, 153, 2)		
14	keepBacteria(203, “ccc”, 140, 2)		**[Fig. 3]**
15	takeOut(310,3)	return 53	
16	getBacteria(311, “aaa”)	return 1	
17	getBacteria(312, “ccc”)	return 1	**[Fig. 4]**

[Table 1]

`Order 17. getBacteria(312, “ccc”)` **에서 보관소에 있는 박테리아는** `[Fig. 4]` **와 같다.**

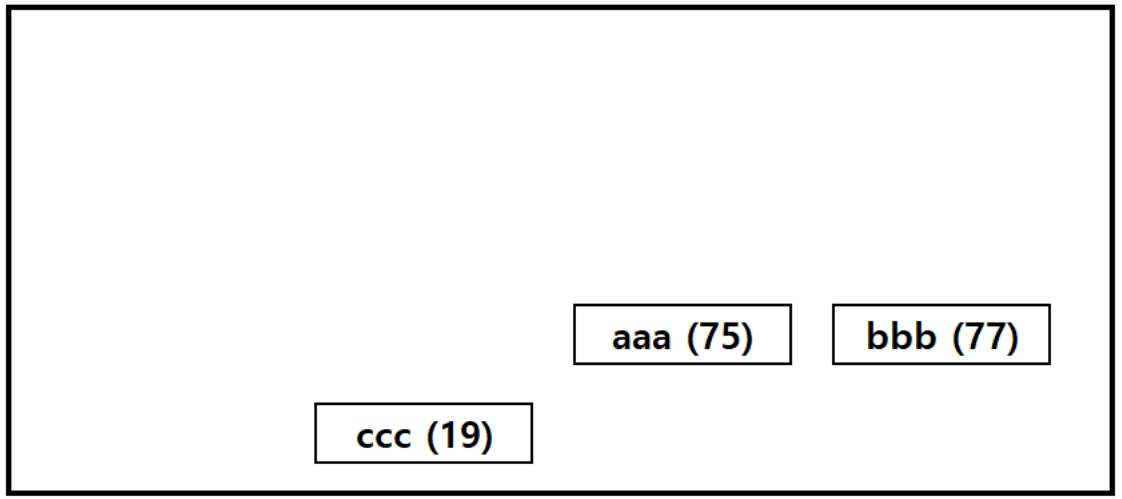

Fig. 4

Problem analysis : API 및 제약조건

TS박테리아

- 전체 함수 호출 수는 50,000이다.

- init(N, nameList[], halfTime[]) :
  N(최대100)종류의 박테리아 정보가 주어진다.
  문자열 이름으로 주어지면 박테리아 별 반감기가 있다.
  반감기 마다 수명이 절반으로 줄어든다. 수명이 9이하 남으면 폐기된다.

- int getBacteria(stamp, name) : 최대 15,000번 호출
  stamp 시각에 연구소에 남아 있는 name바이러스 수를 반환한다.

- int takeout(stamp, count) : 최대 15,000번 호출
  stamp 시각에 연구소에 남아 있는 바이러스 중에
  1순위 생명력이 가장 작은, 2순위 보관 시각이 빠른 바이러스 count개를 반출한다.

- void keepBacteria (stamp, name, life, count) : 언급이 없다. -> 5만이 될 수 있다.
  stamp 시각에 life 생명력의 name 박테리아를 count 개 보관한다.

Problem analysis : API 및 제약조건

- 시간에 흐름에 따라 처리해야 하는 time flow 문제이다.
- 따라서 반감기가 빨리 돌아오는 우선순위로 처리하여
  바이러스의 수명을 업데이트 할 수 있다.

- 하지만 바이러스의 초기 수명과 바이러스별 반감기가 다르므로
  반감기가 돌아오는 순서와 남은 수명이 작은 순서가 항상 일치한다고 할 수 없다.

- 그렇다고 남은 수명이 짧은 순으로 정렬할 수도 없다.
  남은 수명은 짧은데 반감기가 긴 박테리아와
  남은 수명은 긴데 반감기가 짧은 박테리아가 시간이 지나며
  남은 수명의 상태가 반전될 수 있기 때문이다.

Problem analysis : API 및 제약조건

- 반감기 업데이트 시간의 오름차순을 우선순위로 하는 것이 적절한데
  이것 만으로는 takeOut(stamp, count)에 적절히 대응할 수 없다.

- 어떻게 두 가지 우선순위를 모두 만족하도록
  할 수 있을까?

Solution sketch

Solution sketch

TS박테리아

- 박테리아 이름이 init()에서 문자열로 주어지는데 이는 다음과 같이 처리할 수 있다.
  1. 최대 100개 이므로 배열에 순차적으로 저장하고
     필요할 때마다 매번 순차 탐색으로 찾는 방법을 사용할 수 있다.
  2. 사용자 정의 해시, unordered_map<string, int>를 사용할 수 있다.
     unordered_map<string, int> 이 권장된다.

- 반감기가 돌아오는 시간의 오름차순 우선순위와
  남은 생명력의 오름차순 우선순위를 동시에 만족하는 하나의 자료구조는
  설계하기 쉽지 않다.

- 따라서 두 개의 우선순위를 위하여
  두 개의 우선순위 큐를 사용하는 전략을 사용해 보자.

Solution sketch

- 먼저 keepBacteria로 주어지는 데이터에 keepID를 부여하고
  이를 배열의 index로 하여 다음과 같은 클래스로 정보를 저장할 수 있다.

```
struct Bacteria {
    // tick : 수명을 절반으로 줄여야 하는 시각
    // life : 현재 남은 수명
    // cnt  : 박테리아 수
    // bid  : renumbering 박테리아 종류id : halfLife[], remainCnt[]의 index로 사용
    int tick, life, cnt, bid;
    int update() {
        tick += halfLife[bid]; // 다음 업뎃 시각
        life /= 2;             // 수명 줄이기
        if (life < 10) {       // 폐기 처분하기
            remainCnt[bid] -= cnt;
            life = cnt = 0;
        }
        return life && cnt;    // 수명과 마리수가 남아 있어야 참.
    }
}B[BLM];  // origin 상태, B[idx]에서 idx는 keepID(keep()호출시 마다 순차부여)
```

Solution sketch

TS박테리아

- 위 데이터를 database로 하여 아래와 같은 우선순위 큐를 사용할 수 있다.

```
// timpHeap으로 time flow 를 처리하여 B[]의 상태를 관리
// lifeHeap은 lazy 업데이트
priority_queue<pii, vector<pii>, greater<pii>> timeHeap; //<time, keepID>
priority_queue<pii, vector<pii>, greater<pii>> lifeHeap; //<life, keepID>
```

- timeHeap으로 <반감기 업데이트 시각, 입력 순번> 의 minheap을 유지한다.
- lifeHeap으로 <잔여 생명력, 입력 순번(keepID)> 의 minheap을 유지한다.

- 매 함수 호출에 stamp를 기준시각으로
  B[입력 순번(keepID)]의 life, cnt를 업데이트 하면서
  lifeHeap에 대하여 lazy update 또는 real time update한다.

Solution sketch

TS박테리아

- 여기서는 lazy update하는 방법으로 다루어 보자.

- takeOut()함수가 호출되는 경우 lifeHeap에서
  생명력의 오름차순 데이터를 사용할 때,
  B[입력 순번(keepID)] database와 생명력이 일치하는지 확인하여
  유효성 검사를 통과한 데이터를 사용한다.

- timeHeap으로 반감기 데이터를 업데이트하는 경우
  lifeHeap 에 추가하여 넣는다.
  이 경우 [입력 순번(keepID)]은 같으나 잔여생명력은 다른
  여러 데이터가 lifeHeap에 존재할 수 있다.

Solution sketch

- 이제 각 함수 별 할 일을 정리해 보자.

void init(int N, char nameList[][], int halfTime[])

- ✓ renumbering 해시 함수, N, keepID(입력 순번)등을 초기화 한다.
- ✓ timeHeap과 lifeHeap을 초기화 한다.
- ✓ 이름에 renumbering id를 부여한다.
- ✓ 바이러스 종별 반감기를 백업해둔다.
- ✓ 바이러스 종별 남은 수 remainCnt[]를 0으로 초기화 한다.
  remainCnt[]는 getBacteria()함수 호출시 사용된다.

Solution sketch

void keepBacteria(int stamp, char name[], int life, int count)

- ✓ `update(stamp)`**를 실행한다.**
- ✓ `int bid = hmap[name];`**으로 박테리아 종류** `renumbering id`**를 얻는다.**
- ✓ `remainCnt[bid] += count;` **로** `bid`**박테리아의 개수 증가**
- ✓ **새로 보관되는 박테리아이므로** `B[++keepID]`**에 정보를 저장한다.**

  `B[++keepID] = {stamp +` **반감기,** `life, count, bid};`
- ✓ `timeHeap`**에** `{stamp +` **반감기,** `keepID}`**를 저장한다.**
- ✓ `lifeHeap`**에** `{life, keepID}` **를 저장한다.**

Solution sketch

- init()을 제외한 모든 함수의 첫 행에서 호출하는 update(stamp) 함수는 다음과 같이 설계할 수 있다.

```
void update(int baseTick) {         // 시간 흐름에 따른 업데이트
    int tick, kid;
    // 기준시각 baseTick까지 업데이트
    while (!timeHeap.empty() && timeHeap.top().first <= baseTick) {
        tie(tick,  kid) = timeHeap.top(); // 시간우선순위
        timeHeap.pop();
        if (B[kid].update()) {    // 반감기가 지났음에도 유효하다면
            timeHeap.push({ B[kid].tick, kid }); // 다시 넣기(time flow update)
            lifeHeap.push({ B[kid].life, kid }); // 추가적으로 넣기(lazy update)
        }
    }
}
```

Solution sketch

int takeOut(int stamp, int count)

- ✓ update(stamp)를 실행한다.
- ✓ lifeHeap에 자료가 있고 count가 남아 있는 동안 다음을 실행한다.

```
auto t = lifeHeap.top();
lifeHeap.pop();
Bacteria& bact = B[t.second];  // 가장 우선순위가 높다고 판단되는 데이터
if (t.first == bact.life) {    // 현재 데이터가 유효하다면
    int minCnt = min(count, bact.cnt); // 둘 중 적은수 구하기
    ret += minCnt * bact.life; // 박테리아 생명의 합
    count -= minCnt;           // 처리해야 할 수 업데이트
    bact.cnt -= minCnt;        // 현재 박테리아 남은 수 업데이트
    remainCnt[bact.bid] -= minCnt;
    if (bact.cnt) { //박테리아가 남은 경우 다시 lifeHeap에 추가
        lifeHeap.push({ bact.life, t.second });
    }
}
```

Solution sketch

int getBacteria(int stamp, char name[])

- ✓ `update(stamp)`를 실행한다.
- ✓ **int bid = hmap[name]**
- ✓ **단지 remainCnt[bid]를 반환한다.**

Solution sketch

[Summay]

- **우선순위자료구조를 사용할 수 있다.**

- **하나의 자료를 서로 다른 두 가지 우선순위로 다룰 수 있다.**

- **시간의 흐름에 따른 문제는 처음 만나는 경우 대처가 쉽지 않을 수 있다.**
  미리 준비해 두자.

- **time flow 최근 문제**
  - ✓ 20220218 : [H2210]**나무를 지키는 경찰** : pq, time flow, bfs(dfs)
  - ✓ 20211203 : [H2151]Process Scheduler : time flow, queue

Code example 1

: PQ

unordered_map

lazy update

Code example1


```cpp
#include <queue>
#include <vector>
#include <string>
#include <tuple>
#include <algorithm>
#include <unordered_map>
using namespace std;

const int LM = 101;
const int BLM = 50005;
using pii = pair<int, int>;
int N;              // 박테리아 종류 수
int keepID;         // keep()함수 호출시 순차적으로 부여한 박테리아 아이디
int halfLife[LM];   // 반감기
int remainCnt[LM]; // 박테리아 종류별 남은수. get()함수에서 사용.
unordered_map<string, int> hmap; // 박테리아 이름에 부여한 renumbering id
```

Code example1

TS박테리아

```
struct Bacteria {
    // tick: 수명이 절반으로 줄어드는 시각
    // life : 남은 수명
    // cnt  : 박테리아 수
    // bid  : renumbering id : halfLife[], remainCnt[]의 index로 사용
    int tick, life, cnt, bid;
    int update() {
        tick += halfLife[bid]; // 다음 업뎃 시각
        life /= 2;             // 수명 줄이기
        if (life < 10) {       // 폐기 처분하기
            remainCnt[bid] -= cnt;
            life = cnt = 0;
        }
        return life && cnt;    // 수명과 마리수가 남아 있어야 참.
    }
}B[BLM];  // origin 상태, B[idx]에서 idx는 keepID(keep()호출시 마다 순차부여)
```

Code example1

TS박테리아

```
// timpHeap으로 time flow 를 처리하여 B[]의 상태를 관리
// lifeHeap은 lazy 업데이트
priority_queue<pii, vector<pii>, greater<pii>> timeHeap; //<time, keepID>
priority_queue<pii, vector<pii>, greater<pii>> lifeHeap; //<life, keepID>

void init(int N, char nameList[100][10], int halfTime[100]) {
    ::N = N, keepID = 0;
    hmap.clear();
    timeHeap = {}, lifeHeap = {};
    for (int i = 0; i < N; ++i) {
        hmap[nameList[i]] = i;     // 이름에 renumbering id
        halfLife[i] = halfTime[i]; // 반감기 백업
        remainCnt[i] = 0;          // 종류별 남은수 초기화
    }
}
```

Code example1


```
void update(int baseTick) {         // 시간 흐름에 따른 업데이트
    int tick, kid;
    // 기준시각 baseTick까지 업데이트
    while (!timeHeap.empty() && timeHeap.top().first <= baseTick) {
        tie(tick, kid) = timeHeap.top(); // 시간우선순위
        timeHeap.pop();
        if (B[kid].update()) {    // 반감기가 지났음에도 유효하다면
            timeHeap.push({ B[kid].tick, kid  }); // 다시 넣기(time flow update)
            lifeHeap.push({ B[kid].life, kid  }); // 추가적으로 넣기(lazy update)
        }
    }
}

void keepBacteria(int stamp, char name[10], int life, int count) {
    update(stamp);
    int bid = hmap[name]; // 박테리아 종류, renumbering id

    // 새로운 박테리아 등록
    remainCnt[bid] += count; // bid종류 박테리아의 남은수 업데이트
    B[++keepID] = { stamp + halfLife[bid], life, count, bid }; // origin data
    timeHeap.push({ stamp + halfLife[bid], keepID });
    lifeHeap.push({ life, keepID });
}
```

Code example1

TS박테리아

```
int takeOut(int stamp, int count) {
    update(stamp);
    int ret = 0;
    while (!lifeHeap.empty() && count) {
        auto t = lifeHeap.top();
        lifeHeap.pop();
        Bacteria& bact = B[t.second];  // 가장 우선순위가 높다고 판단되는 데이터
        if (t.first == bact.life) {    // 현재 데이터가 유효하다면
            int minCnt = min(count, bact.cnt); // 둘 중 적은수 구하기
            ret += minCnt * bact.life; // 박테리아 생명의 합
            count -= minCnt;           // 처리해야 할 수 업데이트
            bact.cnt -= minCnt;        // 현재 박테리아 남은 수 업데이트
            remainCnt[bact.bid] -= minCnt;
            if (bact.cnt) { //박테리아가 남은 경우 다시 lifeHeap에 추가
                lifeHeap.push({ bact.life, t.second });
            }
        }
    }
    return ret;
}

int getBacteria(int stamp, char name[10]) {
    update(stamp);
    return remainCnt[hmap[name]];
}
```

Code example2

: nonSTL

real time update

Code example2


```
using LL = long long;
const int LM = 101;
const int BLM = 50005;
int N;              // 박테리아 종류 수
int keepID;         // keep()함수 호출시 순차적으로 부여한 박테리아 아이디
int halfLife[LM];   // 반감기
int remainCnt[LM];  // 박테리아 종류별 남은수. get()함수에서 사용.
LL keys[LM * 2];    // key[renumbering id] = 박테리아 이름을 27진수로...

LL getCode(char*s, LL key = 0) {
    for (; *s; ++s) key = key * 27 + *s - 96;
    return key;
}
int probing(LL k) { // linear search
    for (int i = 0; i < N; ++i)
        if (keys[i] == k) return i;
    return 0;
}
```

Code example2

TS박테리아

```
struct Bacteria {
    // tick: 수명이 절반으로 줄어드는 시각
    // life : 남은 수명
    // cnt  : 박테리아 수
    // bid  : renumbering id : halfLife[], remainCnt[]의 index로 사용
    int tick, life, cnt, bid;
    int update() {
        tick += halfLife[bid]; // 다음 업뎃 시각
        life /= 2;             // 수명 줄이기
        if (life < 10) {       // 폐기 처분하기
            remainCnt[bid] -= cnt;
            life = cnt = 0;
        }
        return life && cnt;    // 수명과 마리수가 남아 있어야 참.
    }
}B[BLM];  // origin 상태, B[idx]에서 idx는 keepID(keep()호출시 마다 순차부여)
```

Code example2


```
bool timeComp(int a, int b) {
    return B[a].tick != B[b].tick? B[a].tick < B[b].tick : a < b;
}
bool lifeComp(int a, int b) {
    return B[a].life != B[b].life ? B[a].life < B[b].life : a < b;
}
struct PQ {  // index heap : real time update
    int heap[BLM], idx[BLM], hn;
    bool(*comp)(int, int);
    void init(int flag){
        comp = flag ? lifeComp : timeComp;
        hn = 0;
    }
    void swap(int&a, int &b) {
        int t = a; a = b; b = t;
        t = idx[a], idx[a] = idx[b], idx[b] = t;
    }
    bool empty() { return hn == 0; }
    int top() { return heap[1]; }
    void push(int id) {
        heap[++hn] = id, idx[id] = hn;
        upheap(hn);
    }
    void pop() {
        erase(heap[1]);
    }
```

Code example2


```
    void upheap(int c) {
        if (c > hn) return; //*****
        for (; c > 1; c >>= 1) {
            if (comp(heap[c], heap[c / 2])) swap(heap[c], heap[c / 2]);
            else break;
        }
    }
    void downheap(int c) {
        if (c == 0) return;  //*****
        for (c *= 2; c <= hn; c <<= 1) {
            if (c < hn && comp(heap[c + 1], heap[c])) c++;
            if (comp(heap[c], heap[c / 2])) swap(heap[c], heap[c / 2]);
            else break;
        }
    }
    void update(int id) {
        upheap(idx[id]);
        downheap(idx[id]);
    }
    void erase(int id) {
        int tid = heap[hn];
        swap(heap[idx[id]], heap[hn--]);
        update(tid);
    }
};
```

Code example2


```
PQ timeHeap; // <time, keepID>
PQ lifeHeap; // <life, keepID>

void init(int N, char nameList[100][10], int halfTime[100]) {
    ::N = N, keepID = 0;
    timeHeap.init(0), lifeHeap.init(1);
    for (int i = 0; i < N; ++i) {
        keys[i] = getCode(nameList[i]); // 이름에 renumbering id
        halfLife[i] = halfTime[i];      // 반감기 백업
        remainCnt[i] = 0;               // 종류별 남은수 초기화
    }
}
```

Code example2


```
void update(int baseTick) {             // 시간 흐름에 따른 업데이트
    // 기준시각 baseTick까지 업데이트
    while (!timeHeap.empty() && B[timeHeap.top()].tick <= baseTick) {
        int kid = timeHeap.top(); // 시간우선순위 keepID
        if (kid == 14) {
            int tt = 1;
        }
        if (B[kid].update()) {    // 반감기가 지났음에도 유효하다면
            timeHeap.update( kid );
            lifeHeap.update( kid );
        }
        else {
            timeHeap.erase(kid);
            lifeHeap.erase(kid);
        }
    }
}
```

Code example2


```
void keepBacteria(int stamp, char name[10], int life, int count) {
    update(stamp);
    int bid = probing(getCode(name)); // 박테리아 종류, renumbering id

    // 새로운 박테리아 등록
    remainCnt[bid] += count; // bid종류 박테리아의 남은수 업데이트
    B[++keepID] = { stamp + halfLife[bid], life, count, bid }; // database
    timeHeap.push( keepID );
    lifeHeap.push( keepID );
    if (keepID == 14) {
        int tt = 1;
    }
}

int min(int a, int b) { return a < b ? a : b; }
```

Code example2


```
int takeOut(int stamp, int count) {
    update(stamp);
    int ret = 0;
    while (!lifeHeap.empty() && count) {
        int kid = lifeHeap.top();  // keepid
        if (kid == 14) {
            int tt = 1;
        }
        Bacteria& bact = B[kid];   // 가장 우선순위가 높은 데이터
        int minCnt = min(count, bact.cnt); // 둘 중 적은수 구하기
        ret += minCnt * bact.life; // 박테리아 생명의 합
        count -= minCnt;           // 처리해야 할 수 업데이트
        bact.cnt -= minCnt;        // 현재 박테리아 남은 수 업데이트
        remainCnt[bact.bid] -= minCnt;
        if (bact.cnt == 0) {       // 박테리아가 남지 않은 경우
            timeHeap.erase(kid);
            lifeHeap.erase(kid);
        }
    }
    return ret;
}

int getBacteria(int stamp, char name[10]) {
    update(stamp);
    return remainCnt[probing(getCode(name))];
}
```

Thank you.