Алгоритмы и структуры данных-2

SET 5. Задача A1.

Весна 2024. Клычков М. Д.

```
INSERT(key):
    ind = hash(key) mod M

    while (table[ind] != NULL)
        if (table[ind] == key) return
        ind = (ind + 1) mod M

    table[ind] = key
```

```
DELETE(key):
    ind = hash(key) mod M

    while (table[ind] != NULL)
        if (table[ind] == key)
            table[ind] = ERASED
            return
        ind = (ind + 1) mod M
```

```
SEARCH(key):
    ind = hash(key) mod M

    while (table[ind] != NULL)
        if (table[ind] == key)
            return true
        ind = (ind + 1) mod M
    return false
```

Рис. 1: Код из условия

Пункт 1. Сразу заметим, что в методе `INSERT` не происходит проверки на переполнение хеш-таблицы, а соответственно, можно предположить, что не происходит и перехеширования (предполагаем, что структура данных содержит только приведенные в условии методы). Это сразу заставляет задуматься о **переполнении** хеш-таблицы.

Например, последовательная вставка элементов, хеши которых полностью покрывают отрезок возможных значений хеш-функции $[0; m-1]$. Такими вставками мы добьемся такого состояния хеш-таблицы, при котором невозможно вставить новое значение. Конкретный пример: при ключе `int` и хеш-функции $h(key) = (key \mod m)$ выполняем вставки

$$\texttt{INSERT}(0), \texttt{INSERT}(1), \ldots, \texttt{INSERT}(m-1).$$

Теперь при последующей вставке $\texttt{INSERT}(m)$ произойдет зацикливание.

Более того, учитывая несовершенность условия цикла `while` метода `INSERT` мы войдем в бесконечный цикл, зацикленно пробегая по всем ячейкам хеш-таблицы, так как условие выхода из цикла

```
table[ind] == nullptr
```

никогда не выполнится.

Заметим, что аналогичная проблема зацикливания при заполненной хеш-таблице свойственна и другим двум методам структуры: `DELETE` и `SEARCH`. Возможное решение этой проблемы будет приведено в **Пункте 2**.

Еще один источник потенциальных проблем — значение `ERASED` удаленных элементов структуры. Снова обратимся к методу `INSERT`: при поиске свободной ячейки для вставляемого элемента не учитывается `ERASED`. Таким образом, «умершие» элементы продолжают хранится в хеш-таблице до полного ее уничтожения, занимая память, которая может быть переиспользована для новых элементов структуры.

Приведем конкретный пример: заполним хеш-таблицу тем же образом, что и в примере выше. Теперь, имея полностью заполненную хеш-таблицу последовательно выполним удаления:

$$\texttt{DELETE}(0), \texttt{DELETE}(1), \ldots, \texttt{DELETE}(m-1).$$

Несмотря на то что все элементы были удалены и логический размер структуры равен 0, любая вставка, удаление или поиск введут программу в бесконечный цикл.

Пункт 2. Сначала поборемся с проблемой зацикливания, которая происходит в каждом из приведенных методов. Предлагается обходить таблицу при помощи цикла `for` вместо `while` с целью ограничить максимальное количество шагов цикла. Будем итерироваться по переменной `diff`, которая принимает значения в полуинтервале $[0; m)$, и получать очередной индекс `ind` как сумма хеша и `diff` по модулю `M`.

Ниже в качестве примера приведена исправленная функция `SEARCH` на `C++` (остальные функции должны быть исправлены аналогично):

```
bool Search(int key) const {
    auto hash = Hash(key);

    for (int diff = 0; diff < m_; ++diff) {
        int i = (hash + diff) % m_;

        if (table_[i] == nullptr) {
            return false;
        }
        if (table_[i]->key == key) {
            return true;
        }
    }
    return false;
}
```

Теперь решим проблему, связанную с `ERASED`. Перепишем код `INSERT` так, чтобы запись в ячейки `ERASED` была разрешена.

```
bool Insert(int key) {
    auto hash = Hash(key);

    for (int diff = 0; diff < m_; ++diff) {
        int i = (hash + diff) % m_;

        if (table_[i] == nullptr || table_[i] == ERASED) {
            table_[i] = new KeyValT(key);
            return true;
        }
    }
    return false;
}
```