Лабораторная работа IV. Динамические структуры данных для реализации поиска.

Задача: познакомить студентов с классическими динамическими структурами данных, лежащими в основе «множеств».

Введение

Назовём «множеством» некоторый набор элементов. Для работы с «множеством» удобно определить следующие операции:

- создание пустого множества;
- проверка множества на то, что оно не пустое;
- выбор представителя из множества (предъявить любой элемент, который есть в множестве, если оно не пустое);
- проверка наличия некоторого элемента key в множестве;
- добавление элемента в множество;
- удаление элемента из множества.

Формализуем «интерфейс», набор функций, с помощью которого мы будем оперировать множеством целых чисел:

```
1 struct Set;
2 Set set_new();
3 bool set_is_empty(Set const &set);
4 int set_example(Set const &set);
5 bool set_contains(Set const &set, int key);
6 Set& set_insert(Set &set, int key);
7 Set& set_remove(Set &set, int key);
8 Set& set_erase(Set &set);
```

- **Set set_new()** создаёт новое пустое множество, возвращает новую структуру типа **struct Set**;
- **bool set_is_empty(Set const &set)** возвращает **true**, если множество пусто, иначе - **false**;

- **int set_example(Set const &set)** возвращает произвольный элемент из множества, если оно не пусто, если множество путо, поведение функции *не определено*;
- **bool set_contains(Set const &set, int key)** возвращает **true**, если элемент **key** содержится в множестве, иначе - **false**;
- **Set& set_insert(Set &set, int key)** добавляет в множество элемент **key** и возвращает ссылку на изменённую структуру множества, если элемент **key** уже содержится в множестве, то ничего не делает;
- **Set& set_remove(Set &set, int key)** удаляет элемент **key** и возвращает ссылку на изменённую структуру, если элемента не было в множестве, то ничего не делает;
- **Set& set_erase(Set &set)** удаляет все элементы из множества и возвращает всю занятую память, возвращает изменённую структуру.

Нам необходимо описать структуру **Set**, её поля, и определить функции. Это можно сделать множеством различных способов. Например, можно реализовать все описанные выше функции, используя массив как структуру данных для хранения элементов.

```
1  struct Set {
2      int elements[100];
3      unsigned size = 0;
4  };
5
6  Set set_new() {
7      return Set{};
8  }
9
10 bool set_is_empty(Set const &set) {
11     return 0 == set.size;
12 }
13
14 int set_example(Set const &set) {
15     return set.elements[0];
16 }
17
18 bool set_contains(Set const &set, int key) {
```

```
19      for (unsigned idx = 0; idx != set.size; ++idx)
20          if (set.elements[idx] == key)
21              return true;
22      return false;
23  }
24
25  Set& set_insert(Set &set, int key) {
26      if (set_contains(set, key)) return set;
27      set.elements[set.size++] = key;
28      return set;
29  }
30
31  Set& set_remove(Set &set, int key) {
32      unsigned shift_idx = 0;
33      for (; shift_idx != set.size; ++shift_idx)
34          if (key == set.elements[shift_idx])
35              break;
36      if (set.size == shift_idx) return set;
37
38      for(; shift_idx != set.size - 1; ++shift_idx)
39          set.elements[shift_idx] = set.elements[shift_idx + 1];
40      --set.size;
41      return set;
42  }
43
44  Set& set_erase(Set &set) {
45      set.size = 0;
46      return set;
47  }
```

В этом случае асимптотические сложности операций по отношению к количеству элементов оказываются следующими:

- создание пустого множества - $O(1)$;
- проверка множества на пустоту - $O(1)$;
- взятие произвольного представителя из непустого множества - $O(1)$;
- проверка наличия элемента в множестве - $O(N)$;
- добавление элемента в множество - $O(N)$;
- удаление элемента из множества - $O(N)$;
- удаление всего множества - $O(1)$.

Чаще всего мы будем выполнять операции поиска, добавления и удаления элементов.

Хеш-таблица (3 - 5)

Мы можем реализовать абстракцию множества на базе хеш-таблицы. Напишите код для хеш-таблицы с разрешением коллизий методом цепочек или методом открытой адресации, которая будет автоматически увеличивать размер необходимой памяти и проводить процедуру *перехеширования* (rehash). Вы можете воспользоваться кодом массива, динамически меняющего свой размер, из предыдущей лабораторной работы. На основе хеш-таблицы реализуйте абстракцию множества, написав все необходимые функции, перечисленные во введении. Прямым измерением времени проверьте асимптотические сложности операций поиска, добавления и удаления элементов из хеш-таблицы. В качестве значений **хеш-фунции** выберите целое беззнаковое число, соответствующее двоичной записи целого числа со знаком (это можно сделать с помощью **union**).

Бинарное дерево поиска (6 - 10)

Мы можем реализовать абстракцию множества и на базе бинарного дерева, если хотим получить определённый порядок хранения. Реализуйте множество на базе бинарного дерева поиска без самобалансировки и на базе самобалансирующегося дерева поиска, красно-чёрного или АВЛ деревьев. Сравните асимптотические сложности операций поиска, добавления и удаления для двух случаев: элементы поступают упорядоченным образом, элементы поступают случайным образом.

Дополнение

Следует обязательно дополнить решение тестами правильной работы каждой отдельной функции.