#3: Search

Алгоритмы для решения многих задач становятся намного проще и эффективней, если удается подобрать или создать такую организацию данных, которая была бы ориентирована на решение такой задачи. Классическим примером такой задачи является задача поиска. Для эффективного ее решения известны структуры данных: бинарное дерево поиска и хеш-таблица.

В вашу задачу входит реализация АТД ассоциативного массива.

Ассоциативный массив

```
template<typename K, typename V>
class Dictionary final
```

должен поддерживать следующие операции:

- `Dictionary();`
  Создает пустой ассоциативный массив.
- `~Dictionary();`
  Деструктор освобождает память, выделенную под хранение элементов. При необходимости, при освобождении памяти вызываются деструкторы хранимых элементов.
- `void put(const K& key, const V& value);`
  Добавляет переданную пару ключ-значение в ассоциативный массив. Если такой ключ уже существует, связанное с ним значение должно быть заменено на переданное.
- `void remove(const K& key);`
  Удаляет элемент с указанным ключом из ассоциативного массива.
- `bool contains(const K& key);`
  Возвращает true, если в ассоциативном массиве существует элемент с указанным ключом.
- `const T& operator[](const K& key) const;`
  `T& operator[](const K& key);`
  Оператор индексирования позволяет обратиться к элементу ассоциативного массива по ключу для чтения и для записи. Если элемента с таким ключом нет, то неконстантный оператор должен его создать и

добавить в ассоциативный массив со значением по умолчанию. Константная версия оператора должна просто вернуть значение по умолчанию.

- `int size() const;`
  Возвращает текущий размер (количество реально существующих в ассоциативном массиве элементов).

- `Iterator iterator();`
  `const Iterator iterator() const;`
  Возвращает итератор, указывающий на первый элемент массива.

Аналогично динамическому массиву из работы 2, необходимо либо предоставить правильный механизм копирования и присваивания, либо запретить его.

Итератор должен быть реализован как внутренний для `Dictionary<K, V>` класс, и предоставлять следующий интерфейс:

- `const K& key() const;`
  Получает ключ в текущей позиции итератора.

- `const V& get() const;`
  Получает значение в текущей позиции итератора.

- `void set(const V& value);`
  Устанавливает значение в текущей позиции итератора, сохраняя значение ключа. Ключ отдельно изменить нельзя.

- `void next();`
  Перемещает текущую позицию итератора на следующий элемент ассоциативного массива. Порядок обхода неважен, но итератор должен гарантировать, что будут просмотрены все элементы массива, и каждый элемент будет встречен только один раз.

- `void prev();`
  Перемещает текущую позицию итератора на предыдущий элемент.

- `bool hasNext() const;`
  Возвращает `true`, если итератор может перейти к следующему элементу, или `false`, если итератор позиционирован на последний элемент.

- `bool hasPrev() const;`
  Возвращает `true`, если итератор может перейти к предыдущему элементу, или `false`, если итератор позиционирован на первый элемент.

Пример использования: поместим в ассоциативный массив свойства для персонажа игры и выведем их на экран.

```
Dictionary<std::string, int> npc;
npc.put(“health”, 10);
npc.put(“armor”, 20);
npc.put(“ammo”, 5);

for (auto it = a.iterator(); it.hasNext(); it.next())
  std::cout << it.key() << “ “ << it.get() << std::endl;
```

АТД должен быть реализован на основе красно-черного дерева (CLRS: RB Tree), левосклоняющегося красно-черного дерева (Sedgewick: LLRB Tree) или AVL-дерева. Можно считать, что тип K является упорядоченным и предоставляет правильную перегрузку `operator<()` и `operator==()`.

Для всех реализованных методов ассоциативного массива должны быть созданы необходимые модульные тесты.