VÕ TIẾN
Thảo luận kiến thức CNTT trường BK về KHMT(CScience), KTMT(CEngineering)
https://www.facebook.com/groups/khmt.ktmt.cse.bku

Cấu Trúc Dữ Liệu và Giải Thuật (DSA)

DSA2 - HK242

QUẢN LÝ KHO HÀNG SỬ DỤNG HASH, HEAP, TREE, GRAPH

Thảo luận kiến thức CNTT trường BK
về KHMT(CScience), KTMT(CEngineering)
https://www.facebook.com/groups/khmt.ktmt.cse.bku

TP. HỒ CHÍ MINH, THÁNG 4/2025

Mục lục

1 Giới thiệu cơ bản về cấu trúc hash **2**

2 Lý Thuyết BTL dùng Chaining (dùng danh sách liên kết) **4**

3 Bảng băm (Hash Map) **7**

3.1 Thuộc Tính . . . 7

3.2 Các Hàm cần triển khai . . . 8

3.2.1 Hàm khởi tạo . . . 8

3.2.2 Hàm khởi tạo với param là một XMap khác . . . 8

3.2.3 Hàm operator= . . . 8

3.2.4 Hàm hủy . . . 8

3.2.5 Hàm clear . . . 9

3.2.6 Hàm put . . . 10

3.2.7 Hàm get . . . 12

3.2.8 Hàm remove . . . 14

3.2.9 Hàm containsKey . . . 16

3.2.10 Hàm containsValue . . . 17

3.2.11 Hàm keys . . . 18

3.2.12 Hàm values . . . 19

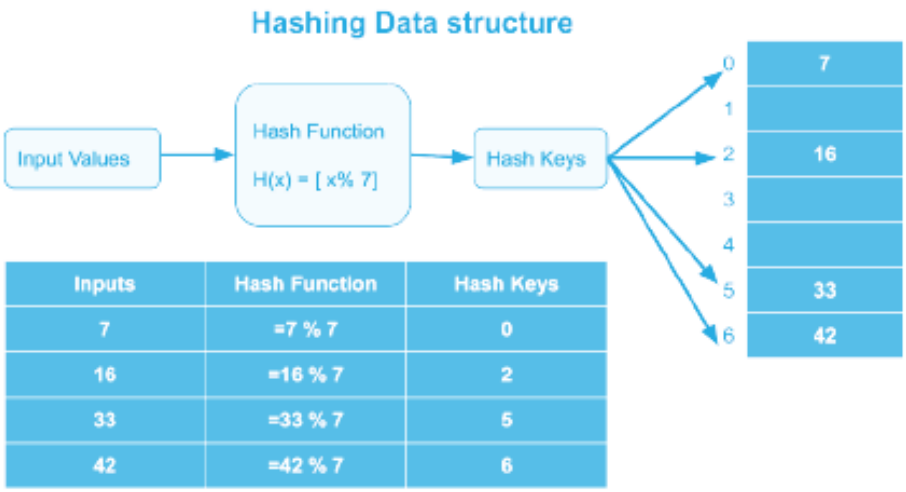
3.2.13 Hàm clashes . . . 20

3.3 API của DLinkedList . . . 21

3.4 xMap . . . 21

1 Giới thiệu cơ bản về cấu trúc hash

Cấu trúc hash (hay còn gọi là bảng băm) là một cấu trúc dữ liệu được sử dụng để lưu trữ và truy xuất dữ liệu một cách nhanh chóng thông qua các cặp key-value. Thay vì phải tìm kiếm qua từng phần tử trong danh sách (như trong mảng hay danh sách liên kết), cấu trúc hash sử dụng một hàm băm (hash function) để ánh xạ key tới một chỉ mục (index) cụ thể trong một mảng để lưu trữ dữ liệu.



Hình 1: *Hashing*

Một số vấn đề của hash

1. **Hàm băm (Hash function):** Đây là hàm chịu trách nhiệm ánh xạ một key tới một chỉ mục trong mảng. Hàm băm lý tưởng sẽ phân phối các key đều đặn để giảm thiểu các xung đột.
2. **Mảng (Hash table):** Dữ liệu được lưu trữ trong một mảng, và các chỉ mục của mảng được tính toán thông qua hàm băm. Mỗi phần tử trong mảng có thể là một giá trị duy nhất hoặc là một cấu trúc dữ liệu phức tạp hơn như danh sách liên kết để xử lý các xung đột.
3. **Xung đột (Collision):** Xung đột xảy ra khi hai key khác nhau được ánh xạ tới cùng một chỉ mục trong mảng. Có hai phương pháp phổ biến để xử lý xung đột:
   - **Phương pháp chuỗi (Chaining):** Tại mỗi chỉ mục trong mảng, nếu có nhiều hơn một phần tử, một danh sách liên kết sẽ được sử dụng để lưu trữ tất cả các phần tử có cùng chỉ mục.
   - **Phương pháp địa chỉ mở (Open addressing):** Khi có xung đột, một thuật toán sẽ được sử dụng để tìm một vị trí trống gần đó trong mảng để lưu trữ phần tử mới.
4. **Tải trọng (Load factor):** Đây là tỷ lệ giữa số lượng phần tử hiện có và kích thước của mảng. Tải trọng cao có thể dẫn đến nhiều xung đột hơn, do đó cần phải tăng kích thước mảng (resize) khi tải trọng vượt quá một ngưỡng nhất định.

Bài Toán hay dùng là đếm số lượng các kí tự trong chuỗi

```cpp
// Hàm để tạo bảng hash từ chuỗi input
void createHashTable(const string& input, int hashTable[256]) {
    // Khởi tạo bảng hash với giá trị ban đầu là 0
    for (int i = 0; i < 256; i++) {
        hashTable[i] = 0;
    }

    // Duyệt qua từng ký tự trong chuỗi và cập nhật bảng hash
    for (char ch : input) {
        hashTable[(unsigned char)ch]++;
    }
}
```

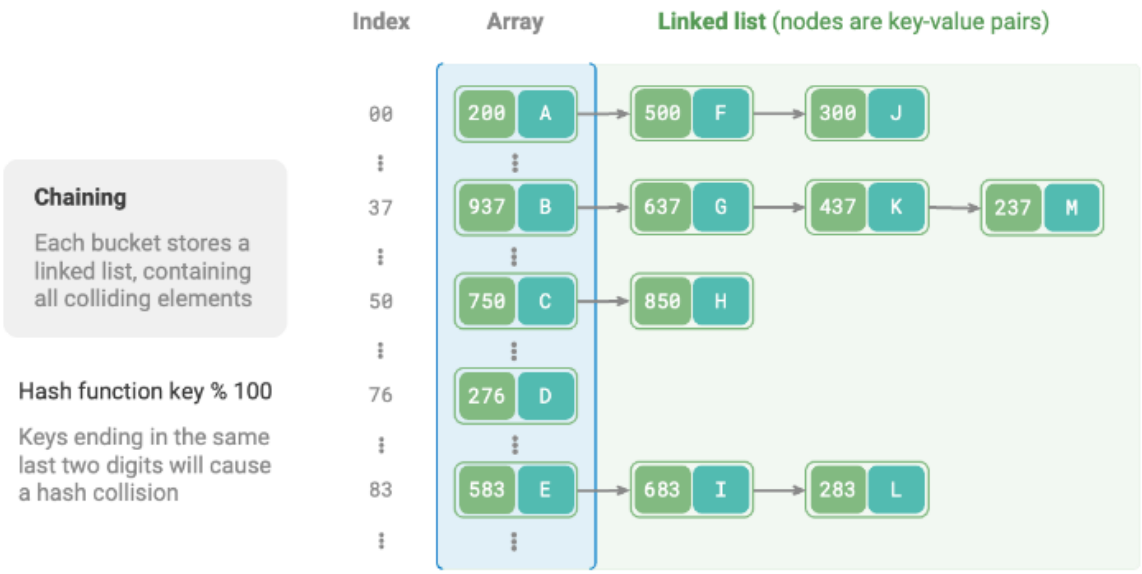
1. **Hash Function (Hàm băm)**: Hàm băm chuyển đổi mỗi ký tự trong chuỗi thành một chỉ mục trong bảng băm. Giả sử ta có bảng băm với 256 phần tử (tương ứng với số ký tự ASCII). Ví dụ:
   - Ký tự 'A' có mã ASCII là 65, hàm băm đơn giản sẽ lấy chỉ mục là 65: `hashFunction('A') = 65`.
   - Tương tự, `hashFunction('B') = 66` và `hashFunction('a') = 97`.
2. **Hash Table (Bảng băm)**: Bảng băm là một mảng với 256 phần tử. Mỗi phần tử trong bảng đại diện cho số lần xuất hiện của ký tự tương ứng trong chuỗi.
   - Khi duyệt qua chuỗi, giá trị tại các chỉ mục tương ứng của bảng băm sẽ được tăng lên.
   - Ví dụ với chuỗi `"hello"`, ta có:
     - `hashTable[104]` (đối với 'h') tăng lên 1.
     - `hashTable[101]` (đối với 'e') tăng lên 1.
     - `hashTable[108]` (đối với 'l') tăng lên 2.
     - `hashTable[111]` (đối với 'o') tăng lên 1.
3. **Collision (Xung đột)**: Xung đột xảy ra khi hai key khác nhau được ánh xạ tới cùng một chỉ mục trong bảng băm. Trong bài toán đếm ký tự, xung đột không xảy ra do mỗi ký tự ASCII được ánh xạ tới một vị trí duy nhất.
4. **Load Factor (Hệ số tải)**: Hệ số tải là tỷ lệ giữa số phần tử đã lưu trữ trong bảng băm và kích thước của bảng. Trong bài toán này, bảng băm có kích thước cố định là 256 (số lượng ký tự ASCII). Ví dụ, nếu chuỗi chứa 100 ký tự khác nhau, hệ số tải sẽ là:

$$\text{Load factor} = \frac{100}{256} \approx 0.39$$

   Hệ số tải cao có thể dẫn đến nhiều xung đột hơn (trong các bài toán phức tạp hơn). Khi hệ số tải tăng, có thể cần phải mở rộng kích thước bảng băm.

2 Lý Thuyết BTL dùng Chaining (dùng danh sách liên kết)

chaining (dùng danh sách liên kết) là một kỹ thuật được sử dụng để xử lý xung đột (collision). Xung đột xảy ra khi hai hoặc nhiều key khác nhau được ánh xạ tới cùng một chỉ mục trong bảng băm. Thay vì ghi đè lên giá trị cũ, phương pháp chaining sử dụng một danh sách liên kết tại mỗi chỉ mục để lưu trữ nhiều phần tử có cùng chỉ mục băm.



Hình 2: *Hashing*

1. **Bảng băm (Hash Table):** Vẫn là một mảng, nhưng mỗi phần tử của mảng không chỉ là một giá trị đơn lẻ mà là một danh sách liên kết. Mỗi chỉ mục trong bảng băm chứa một danh sách các phần tử có cùng giá trị băm.
2. **Hàm băm (Hash Function):** Ánh xạ key đến chỉ mục của bảng băm như thông thường. Nếu nhiều key có cùng giá trị băm, chúng được lưu trữ trong danh sách liên kết tại chỉ mục đó.
3. **Xử lý xung đột:** **Khi xung đột xảy ra (tức là nhiều key có cùng giá trị băm), các phần tử được thêm vào danh sách liên kết tại vị trí đó. Điều này đảm bảo rằng tất cả các phần tử đều được lưu trữ mà không bị ghi đè lên nhau.**

Code tương tự BTL nhưng dễ hơn

```
class HashTable {
    int numBuckets;             // Số lượng buckets trong bảng băm
    list<pair<int, int>>* table; // Con trỏ tới mảng các danh sách liên kết chứa cặp (key, value)

public:
    // Hàm khởi tạo
    HashTable(int buckets) {
        numBuckets = buckets;
        // Cấp phát động cho bảng băm (mảng các danh sách liên kết)
        table = new list<pair<int, int>>[numBuckets];
    }

    // Hàm băm: trả về chỉ mục cho một key
    int hashFunction(int key) {
        return key % numBuckets;
    }

    // Hàm thêm cặp (key, value) vào bảng băm
    void insert(int key, int value) {
        int index = hashFunction(key); // Tính chỉ mục từ key

        // Kiểm tra xem key đã tồn tại trong danh sách tại chỉ mục đó chưa
        auto it = find_if(table[index].begin(), table[index].end(),
                          [key](const pair<int, int>& p) { return p.first == key; });

        if (it != table[index].end()) {
            // Nếu key đã tồn tại, thay thế value
            it->second = value;
            cout << "Key " << key << " already exists and the value has been replaced." << endl;
        } else {
            // Nếu chưa tồn tại, thêm cặp (key, value) vào danh sách liên kết
            table[index].emplace_back(key, value);
        }
    }

    // Hàm xóa một key khỏi bảng băm
    void remove(int key) {
        int index = hashFunction(key);
        // Tìm và xóa cặp (key, value) khỏi danh sách tại chỉ mục 'index'
        table[index].remove_if([key](const pair<int, int>& p) { return p.first == key; });
    }

    // Hàm hủy (giải phóng bộ nhớ động)
    ~HashTable() {
        delete[] table;
    }
};
```

Giải thích Mã Nguồn

- **Cấu trúc dữ liệu:**
  - Mỗi phần tử trong bảng băm là một danh sách liên kết chứa các cặp $(key, value)$. Chúng ta sử dụng `std::pair` để lưu trữ cặp này.
- **Hàm băm (`hashFunction`):**
  - Hàm băm ánh xạ `key` tới chỉ mục trong bảng băm bằng cách sử dụng phép toán chia dư.
- **Thêm cặp $(key, value)$ (`insert`):**
  - Tính chỉ mục từ `key`.
  - Sử dụng `find_if` để kiểm tra xem `key` đã tồn tại trong danh sách liên kết tại chỉ mục đó hay chưa.
  - Nếu đã tồn tại, thay thế giá trị `value`; nếu chưa tồn tại, thêm cặp $(key, value)$ vào danh sách liên kết.
- **Xóa cặp $(key, value)$ (`remove`):**
  - Tìm và xóa cặp $(key, value)$ khỏi danh sách nếu `key` trùng bằng cách sử dụng `remove_if`.

Kết quả

```
// Khởi tạo bảng băm với 10 buckets
HashTable ht(10);

// Thêm các cặp (key, value) vào bảng băm
ht.insert(15, 100);
ht.insert(25, 200);
ht.insert(35, 300);
ht.insert(10, 400);
ht.insert(20, 500);

// Thêm lại cặp (key, value) với key 25 (trùng)
ht.insert(25, 250); // Giá trị của key 25 sẽ được thay thế
```

Kết Quả

```
Key 25 already exists and the value has been replaced.

Hash table after insertion and replacement:
Bucket 0: NULL
Bucket 1: NULL
Bucket 2: NULL
Bucket 3: NULL
Bucket 4: NULL
Bucket 5: (15, 100) -> (25, 250) -> (35, 300) -> NULL
Bucket 6: NULL
Bucket 7: NULL
Bucket 8: NULL
Bucket 9: (10, 400) -> (20, 500) -> NULL
```

3 Bảng băm (Hash Map)

3.1 Thuộc Tính

1. **DLinkedList<Entry*>* table:** Mảng chứa danh sách liên kết kép, mỗi phần tử là một con trỏ đến `Entry`, nơi `Entry` lưu trữ cặp khóa-giá trị.
2. **int capacity:** Sức chứa hiện tại của bảng băm, xác định kích thước tối đa có thể chứa các phần tử.
3. **int count:** Số lượng phần tử hiện có trong bảng băm.
4. **float loadFactor:** Hệ số tải, xác định mức độ sử dụng không gian trước khi cần mở rộng. Số lượng phần tử không được vượt quá `loadFactor` × `capacity`.
5. **int (*hashCode)(K&, int):** Con trỏ hàm tính toán vị trí lưu trữ của khóa trong bảng băm dựa trên kích thước bảng.
6. **bool (*keyEqual)(K&, K&):** Con trỏ hàm so sánh hai khóa để kiểm tra tính bằng nhau, được sử dụng khi kiểu khóa không hỗ trợ toán tử `==`. **giống bài tập lớn trước ta gọi hàm keyEQ nếu muốn so sánh chứ không cần gọi trực tiếp con trỏ hàm (xem lại pdf BTL1)**

```
bool keyEQ(K& lhs, K& rhs) {
  if (keyEqual != 0)
    return keyEqual(lhs, rhs);
  else
    return lhs == rhs;
}
```

7. **bool (*valueEqual)(V&, V&):** Con trỏ hàm so sánh hai giá trị để kiểm tra tính bằng nhau, được sử dụng khi kiểu giá trị không hỗ trợ toán tử `==`. **giống bài tập lớn trước ta gọi hàm valueEQ nếu muốn so sánh chứ không cần gọi trực tiếp con trỏ hàm (xem lại pdf BTL1)**

```
bool valueEQ(V& lhs, V& rhs) {
  if (valueEqual != 0)
    return valueEqual(lhs, rhs);
  else
    return lhs == rhs;
}
```

8. **void (*deleteKeys)(xMap<K, V>*):** Con trỏ hàm để giải phóng bộ nhớ cho các khóa, cần thiết khi kiểu khóa là con trỏ.
9. **void (*deleteValues)(xMap<K, V>*):** Con trỏ hàm để giải phóng bộ nhớ cho các giá trị, cần thiết khi kiểu giá trị là con trỏ.
10. **class Entry**
    (a) **class Entry:** Lớp này đại diện cho một cặp khóa-giá trị trong bảng băm.
    (b) **K key:** `private` Thuộc tính khóa, kiểu dữ liệu là `K`, được sử dụng để xác định vị trí của phần tử trong bảng băm.
    (c) **V value:** `private` Thuộc tính giá trị, kiểu dữ liệu là `V`, lưu trữ giá trị tương ứng với khóa.
    (d) **friend class xMap<K,V>:** Khai báo lớp `xMap<K,V>` là lớp bạn (friend) của `Entry`, cho phép `xMap<K,V>` truy cập trực tiếp vào các thuộc tính riêng tư của `Entry`.
    (e) **Entry(K key, V value):** `public` Hàm khởi tạo nhận vào khóa và giá trị để tạo một đối tượng `Entry`. Nó gán giá trị khóa và giá trị tương ứng cho các thuộc tính `key` và `value`.

3.2 Các Hàm cần triển khai

3.2.1 Hàm khởi tạo

```
template <class K, class V>
xMap<K, V>::xMap(int (*hashCode)(K&, int), float loadFactor,
                 bool (*valueEqual)(V& lhs, V& rhs),
                 void (*deleteValues)(xMap<K, V>*),
                 bool (*keyEqual)(K& lhs, K& rhs),
                 void (*deleteKeys)(xMap<K, V>* pMap)) {
  // TODO YOUR CODE IS HERE
}
```

Gợi ý chi tiết

- Gán các params được truyền vào
- Khởi tạo count bằng 0 và capacity bằng 10
- Khởi tạo new giá trí table với 10 phần tử array

3.2.2 Hàm khởi tạo với param là một XMap khác

```
template <class K, class V>
xMap<K, V>::xMap(const xMap<K, V>& map) {
  // TODO YOUR CODE IS HERE
}
```

Gợi ý chi tiết

- Gọi hàm **copyMapFrom** để copy

3.2.3 Hàm operator=

```
template <class K, class V>
xMap<K, V>& xMap<K, V>::operator=(const xMap<K, V>& map) {
  // TODO YOUR CODE IS HERE
  return *this;
}
```

Gợi ý chi tiết

- Gọi hàm xóa để xóa tất cả các phần tử trước đó để tránh leak memory, gọi hàm **removeInternalData**
- gọi hàm **copyMapFrom** để copy

3.2.4 Hàm hủy

```
template <class K, class V>
xMap<K, V>::~xMap() {
  // TODO YOUR CODE IS HERE
  this->removeInternalData();
}
```

Gợi ý chi tiết

- Gọi hàm **removeInternalData**

3.2.5 Hàm clear

```
template <class K, class V>
void xMap<K, V>::clear() {
  // TODO YOUR CODE IS HERE
}
```

Gợi ý chi tiết

- Gọi hàm `removeInternalData` để giải phóng bộ nhớ của tất cả các phần tử trong bảng băm.
- Khởi tạo lại bảng băm rỗng, với `capacity` là 10.

3.2.6 Hàm put

```
template <class K, class V>
V xMap<K, V>::put(K key, V value) {
  int index = this->hashCode(key, capacity);
  V retValue = value;
  // TODO YOUR CODE IS HERE

  return retValue;
}
```

Gợi ý chi tiết

- **Tính toán địa chỉ băm**: Sử dụng hàm `hashCode` để tính toán chỉ số (hash) cho `key` trong bảng băm dựa trên `capacity`.
- **Truy xuất danh sách liên kết tại chỉ số băm**: Từ chỉ số băm đã tính, truy xuất danh sách liên kết tại vị trí tương ứng trong bảng băm (`table`).
- **Tìm kiếm khóa trong danh sách liên kết**:
  - Nếu tìm thấy `key`, cập nhật giá trị mới (`value`) và trả về giá trị cũ (`oldValue`).
  - Nếu không tìm thấy `key`, tạo một đối tượng `Entry<K, V>` mới với cặp `<key, value>` và thêm nó vào danh sách liên kết (**Thêm vào cuối danh sách**)
- **Tăng số lượng phần tử trong bảng băm**: Sau khi thêm cặp `<key, value>` mới, tăng biến đếm `count` và gọi hàm `ensureLoadFactor()` để kiểm tra và mở rộng bảng băm nếu cần thiết.
- Cập nhật **clashes** trường hợp đụng độ khi **this->table[index]** có tồn tại
- **Trả về giá trị**: Nếu không có giá trị cũ để trả về (trong trường hợp key chưa tồn tại trước đó), hàm sẽ trả về giá trị mặc định của kiểu `V()`.

Gợi ý khác muốn làm cách khác

- dùng **foreach** trong danh sách liên kết để trả về **Entry** và truy cập vào key kiểm tra xem đúng không.
- có thể dùng hàm **contiains** hay **indexOf** để tìm xem có tồn tại không với phương thức truyền vào để so sánh là **keyEQ**
- Dùng & để khởi tạo danh sách liên kết để có thể cập nhật được

```
DLinkedList<Entry* > &temp = this->table[index];
```

Ví Dụ

- `put(5, "Apple")`
- `put(15, "Banana")`
- `put(5, "Orange")`
- `put(2, "Apple")`

Các bước thực thi của mỗi lệnh sẽ như sau:

1. **Lệnh** `put(5, "Apple")`:
   - Tính băm cho khóa 5: `hash = 5 % 10 = 5`.
   - Lấy danh sách liên kết tại chỉ số băm 5 từ bảng băm. Danh sách tại vị trí này hiện rỗng.

- Vì không tìm thấy khóa 5, tạo một `Entry` mới với cặp `<5, "Apple">` và thêm nó vào danh sách liên kết tại vị trí 5.
- Tăng số lượng phần tử trong bảng băm lên 1 và kiểm tra hệ số tải. Hệ số tải vẫn dưới ngưỡng cho phép, nên không cần mở rộng bảng băm.

2. **Lệnh `put(15, "Banana")`:**
   - Tính băm cho khóa 15: `hash = 15 % 10 = 5`. ı

     tem Lấy danh sách liên kết tại chỉ số băm 5. Hiện tại, danh sách này chứa một phần tử là `<5, "Apple">`.
   - Tìm kiếm khóa 15 trong danh sách. Không tìm thấy, nên tạo một `Entry` mới với cặp `<15, "Banana">` và thêm vào danh sách liên kết.
   - Tăng số lượng phần tử trong bảng băm lên 2 và kiểm tra hệ số tải. Hệ số tải vẫn trong ngưỡng cho phép.
3. **Lệnh `put(5, "Orange")`:**
   - Tính băm cho khóa 5: `hash = 5 % 10 = 5`.
   - Lấy danh sách liên kết tại chỉ số băm 5. Hiện tại, danh sách này chứa hai phần tử: `<5, "Apple">` và `<15, "Banana">`.
   - Tìm thấy khóa 5 trong danh sách, cập nhật giá trị cũ từ `"Apple"` thành `"Orange"` và trả về giá trị cũ là `"Apple"`.
4. **Lệnh `put(2, "Apple")`:**
   - Tính băm cho khóa 2: `hash = 2 % 10 = 2`.
   - Lấy danh sách liên kết tại chỉ số băm 2 từ bảng băm. Danh sách tại vị trí này hiện rỗng.
   - Vì không tìm thấy khóa 2, tạo một `Entry` mới với cặp `<2, "Apple">` và thêm nó vào danh sách liên kết tại vị trí 2.
   - Tăng số lượng phần tử trong bảng băm lên 3 và kiểm tra hệ số tải. Hệ số tải vẫn dưới ngưỡng cho phép, nên không cần mở rộng bảng băm.

Hash Table State

Hash table after insertion and replacement:
- Bucket 0: NULL
- Bucket 1: NULL
- Bucket 2: (2, "Apple") -> NULL
- Bucket 3: NULL
- Bucket 4: NULL
- Bucket 5: (5, "Orange") -> (15, "Banana") -> NULL
- Bucket 6: NULL
- Bucket 7: NULL
- Bucket 8: NULL
- Bucket 9: NULL

3.2.7 Hàm get

```
template <class K, class V>
V& xMap<K, V>::get(K key) {
  int index = hashCode(key, capacity);
  // TODO YOUR CODE IS HERE

  // key: not found
  stringstream os;
  os << "key (" << key << ") is not found";
  throw KeyNotFound(os.str());
}
```

Gợi ý chi tiết

- **Tính toán địa chỉ băm**: Sử dụng hàm `hashCode` để tính chỉ số băm cho `key`. Từ đó, lấy danh sách liên kết tại vị trí tính được trong bảng băm (`table`).
- **Tìm kiếm khóa trong danh sách**:
  - Nếu tìm thấy khóa (`key`), trả về giá trị tương ứng (`value`).
  - Nếu không tìm thấy khóa trong danh sách, thực hiện bước (c).
- **Ngoại lệ**: Nếu không tìm thấy `key`, ném ngoại lệ `KeyNotFound` để báo lỗi.
- dùng **foreach** trong danh sách liên kết và gọi hàm **get**
- có thể dùng hàm **get**

Gợi ý hàm cần gọi : foreach, get()

Ví Dụ

- `put(5, "Apple")`
- `put(15, "Banana")`
- `put(5, "Orange")`
- `put(2, "Apple")`
- `get(15)`
- `get(1)`
- `get(2)`

Các bước thực thi của mỗi lệnh sẽ như sau:

1. **Lệnh `get(15)`:**
   - Tính băm cho khóa 15: `hash = 15 % 10 = 5`.
   - Lấy danh sách liên kết tại chỉ số băm 5 từ bảng băm. Hiện tại, danh sách này chứa các phần tử: `<5, "Orange">` và `<15, "Banana">`.
   - Tìm kiếm khóa 15 trong danh sách. Tìm thấy khóa 15 với giá trị tương ứng là `"Banana"`.
   - Trả về giá trị `"Banana"`.
2. **Lệnh `get(1)`:**
   - Tính băm cho khóa 1: `hash = 1 % 10 = 1`.
   - Lấy danh sách liên kết tại chỉ số băm 1 từ bảng băm. Danh sách tại vị trí này hiện rỗng.
   - Tìm kiếm khóa 1 trong danh sách. Không tìm thấy khóa này.

- Ném ngoại lệ `KeyNotFound`.

3. **Lệnh** `get(2)`**:**
   - Tính băm cho khóa 2: `hash = 2 % 10 = 2`.
   - Lấy danh sách liên kết tại chỉ số băm 2 từ bảng băm. Danh sách tại vị trí này chứa một phần tử: `<2, "Apple">`.
   - Tìm kiếm khóa 2 trong danh sách. Tìm thấy khóa 2 với giá trị tương ứng là `"appale"`.
   - Trả về giá trị `"Apple"`.

3.2.8 Hàm remove

```
template <class K, class V>
V xMap<K, V>::remove(K key, void (*deleteKeyInMap)(K)) {
  int index = hashCode(key, capacity);
  // TODO YOUR CODE IS HERE

  // key: not found
  stringstream os;
  os << "key (" << key << ") is not found";
  throw KeyNotFound(os.str());
}

template <class K, class V>
bool xMap<K, V>::remove(K key, V value, void (*deleteKeyInMap)(K),
                        void (*deleteValueInMap)(V)) {
  int index = hashCode(key, capacity);
  // TODO YOUR CODE IS HERE

  // key: not found
  stringstream os;
  os << "key (" << key << ") is not found";
  throw KeyNotFound(os.str());
}
```

Gợi ý chi tiết

- Sử dụng hàm băm `hashCode` để tính toán địa chỉ của `key` và lấy danh sách tại địa chỉ vừa tìm được.
- Tìm xem `key` đã được chứa trong danh sách chưa.
  - Nếu tìm thấy:
    1. Backup giá trị để trả về.
    2. Giải phóng `key` nếu `deleteKeyInMap` và `deleteValueInMap` khác `NULL`.

```
if (deleteKeyInMap) {
    deleteKeyInMap(/*TODO*/)
}
if (deleteValueInMap) {
    deleteValueInMap(/*TODO*/)
}
```

    3. Gỡ bỏ `Entry` (chứa cặp `<key, value>`) ra khỏi danh sách và giải phóng vùng nhớ của `Entry`. Gợi ý: Sử dụng hàm `removeItem` trên danh sách để vừa gỡ bỏ `Entry` và vừa giải phóng bộ nhớ của `Entry` bằng cách truyền con trỏ đến hàm `xMap<K,V>::deleteEntry` vào hàm `removeItem`.

```
static void deleteEntry(Entry* ptr) { delete ptr; }
```

  - Nếu không tìm thấy: Ném ra ngoại lệ `KeyNotFound`.

Gợi ý hàm cần gọi : foreach, contains(keyEQ), indexOf(keyEQ), removeAt()

Ví Dụ

- `put(5, "Apple")`
- `put(15, "Banana")`
- `put(5, "Orange")`
- `put(2, "Apple")`
- `remove(15)`
- `remove(1)`
- `remove(2)`

1. **Lệnh `remove(15)`:**
   - Tính băm cho khóa 15: `hash = 15 % 10 = 5`.
   - Lấy danh sách liên kết tại chỉ số băm 5 từ bảng băm. Danh sách này hiện có các phần tử: `<5, "Orange">` và `<15, "Banana">`.
   - Tìm kiếm khóa 15 trong danh sách. Tìm thấy khóa 15 với giá trị tương ứng là `"Banana"`.
   - Backup giá trị `"Banana"` để trả về.
   - Giải phóng vùng nhớ cho `key` nếu `deleteKeyInMap` khác `nullptr`.
   - Gỡ bỏ `Entry` chứa cặp `<15, "Banana">` ra khỏi danh sách và giải phóng vùng nhớ của `Entry`.
2. **Lệnh `remove(1)`:**
   - Tính băm cho khóa 1: `hash = 1 % 10 = 1`.
   - Lấy danh sách liên kết tại chỉ số băm 1 từ bảng băm. Danh sách tại vị trí này hiện rỗng.
   - Tìm kiếm khóa 1 trong danh sách. Không tìm thấy khóa này.
   - Ném ngoại lệ `KeyNotFound`.
3. **Lệnh `remove(2)`:**
   - Tính băm cho khóa 2: `hash = 2 % 10 = 2`.
   - Lấy danh sách liên kết tại chỉ số băm 2 từ bảng băm. Danh sách tại vị trí này chứa phần tử: `<2, "Apple">`.
   - Tìm kiếm khóa 2 trong danh sách. Tìm thấy khóa 2 với giá trị tương ứng là `"Apple"`.
   - Backup giá trị `"Apple"` để trả về.
   - Giải phóng vùng nhớ cho `key` nếu `deleteKeyInMap` khác `nullptr`.
   - Gỡ bỏ `Entry` chứa cặp `<2, "Apple">` ra khỏi danh sách và giải phóng vùng nhớ của `Entry`.

3.2.9 Hàm containsKey

```
template <class K, class V>
bool xMap<K, V>::containsKey(K key) {
  int index = hashCode(key, capacity);
  // TODO YOUR CODE IS HERE
}
```

Gợi ý chi tiết

- Sử dụng hàm băm `hashCode` để tính toán địa chỉ của `key` và lấy danh sách tại địa chỉ vừa tìm được.
- Tìm `key` trong danh sách ở trên và trả về kết quả tương ứng:
  - Nếu tìm thấy `key`, trả về `true`.
  - Nếu không tìm thấy `key`, trả về `false`.

Gợi ý hàm cần gọi : foreach, contains(keyEQ), indexOf(keyEQ)

Ví Dụ

- `put(5, "Apple")`
- `put(15, "Banana")`
- `put(5, "Orange")`
- `put(2, "Apple")`
- `containsKey(15)`
- `containsKey(1)`
- `containsKey(2)`

1. **Lệnh `containsKey(15)`:**
   - Tính băm cho khóa 15: `hash = 15 % 10 = 5`.
   - Lấy danh sách liên kết tại chỉ số băm 5 từ bảng băm. Danh sách này hiện có các phần tử: `<5, "Orange">` và `<15, "Banana">`.
   - Tìm kiếm khóa 15 trong danh sách. Tìm thấy khóa 15.
   - Trả về `true` vì khóa 15 tồn tại trong bảng băm.
2. **Lệnh `containsKey(1)`:**
   - Tính băm cho khóa 1: `hash = 1 % 10 = 1`.
   - Lấy danh sách liên kết tại chỉ số băm 1 từ bảng băm. Danh sách tại vị trí này hiện rỗng.
   - Tìm kiếm khóa 1 trong danh sách. Không tìm thấy khóa này.
   - Trả về `false` vì khóa 1 không tồn tại trong bảng băm.
3. **Lệnh `containsKey(2)`:**
   - Tính băm cho khóa 2: `hash = 2 % 10 = 2`.
   - Lấy danh sách liên kết tại chỉ số băm 2 từ bảng băm. Danh sách tại vị trí này chứa phần tử: `<2, "Apple">`.
   - Tìm kiếm khóa 2 trong danh sách. Tìm thấy khóa 2.
   - Trả về `true` vì khóa 2 tồn tại trong bảng băm.

3.2.10 Hàm containsValue

```
template <class K, class V>
bool xMap<K, V>::containsValue(K key) {
  int index = hashCode(key, capacity);
  // TODO YOUR CODE IS HERE
}
```

Gợi ý chi tiết

- duyệt qua tất cả index trong array
- Tìm kiếm từng phần tử trong array kiểm tra values có tồn tại hay không để trả về

Gợi ý hàm cần gọi : foreach, contains(valueEQ), indexOf(valueEQ)

Ví Dụ

- `put(5, "Apple")`
- `put(15, "Banana")`
- `put(5, "Orange")`
- `put(2, "Apple")`
- `containsValue("Apple")`
- `containsValue("Orange")`
- `containsValue( "Coconut")`

1. **Lệnh** `containsValue("Apple")`:
   - Lặp qua từng chỉ số băm từ 0 đến 9 để kiểm tra tất cả các danh sách liên kết trong bảng băm.
   - Tại chỉ số băm 2, tìm thấy phần tử `<2, "Apple">`.
   - Trả về `true` vì giá trị `"Apple"` tồn tại trong bảng băm.
2. **Lệnh** `containsValue("Orange")`:
   - Lặp qua từng chỉ số băm từ 0 đến 9 để kiểm tra tất cả các danh sách liên kết trong bảng băm.
   - Tại chỉ số băm 5, tìm thấy phần tử `<5, "Orange">`.
   - Trả về `true` vì giá trị `"Orange"` tồn tại trong bảng băm.
3. **Lệnh** `containsValue("Coconut")`:
   - Lặp qua từng chỉ số băm từ 0 đến 9 để kiểm tra tất cả các danh sách liên kết trong bảng băm.
   - Không tìm thấy giá trị `"Coconut"` trong bất kỳ danh sách nào.
   - Trả về `false` vì giá trị `"Coconut"` không tồn tại trong bảng băm.

3.2.11 Hàm keys

```
template <class K, class V>
DLinkedList<K> xMap<K, V>::keys() {
  // TODO YOUR CODE IS HERE
}
```

Gợi ý chi tiết

- duyệt foreach qua tất cả phần tử array là lấy index

Gợi ý hàm cần gọi : foreach, get()

Ví Dụ

- `put(5, "Apple")`
- `put(15, "Banana")`
- `put(5, "Orange")`
- `put(2, "Apple")`
- `keys()`

1. **Lệnh `keys()`:**
   - Lặp qua tất cả các chỉ số băm từ 0 đến 9 để thu thập các khóa.
   - Tại chỉ số băm 2, tìm thấy khóa 2.
   - Tại chỉ số băm 5, tìm thấy khóa 5 và 15.
   - Tạo một danh sách chứa tất cả các khóa tìm được: `[2, 5, 15]`.
   - Trả về danh sách các khóa: `[2, 5, 15]`.

3.2.12 Hàm values

```
template <class K, class V>
DLinkedList<V> xMap<K, V>::values() {
  // TODO YOUR CODE IS HERE
}
```

Gợi ý chi tiết

- duyệt foreach qua tất cả phần tử array là lấy index

Gợi ý hàm cần gọi : foreach, get()

Ví Dụ

- `put(5, "Apple")`
- `put(15, "Banana")`
- `put(5, "Orange")`
- `put(2, "Apple")`
- `keys()`

Các bước thực thi của mỗi lệnh sẽ như sau:

1. **Lệnh `values()`:**
   - Lặp qua tất cả các chỉ số băm từ 0 đến 9 để thu thập các giá trị.
   - Tại chỉ số băm 2, tìm thấy giá trị `"Apple"`.
   - Tại chỉ số băm 5, tìm thấy giá trị `"Orange"` và `"Banana"`.
   - Tạo một danh sách chứa tất cả các giá trị tìm được: `["Apple", "Orange", "Banana"]`.
   - Trả về danh sách các giá trị: `["Apple", "Orange", "Banana"]`.

3.2.13 Hàm clashes

```
template <class K, class V>
DLinkedList<V> xMap<K, V>::values() {
  // TODO YOUR CODE IS HERE
}
```

Gợi ý chi tiết

- Với mỗi danh sách (địa chỉ trong bảng băm), đếm số phần tử có trong danh sách liên kết.

Gợi ý hàm cần gọi : foreach, count()

Ví Dụ

- `put(5, "Apple")`
- `put(15, "Banana")`
- `put(5, "Orange")`
- `put(2, "Apple")`
- `keys()`

1. **Lệnh `clashes()`:**
   - Tạo một danh sách liên kết rỗng để lưu trữ số phần tử tại từng địa chỉ.
   - Lặp qua tất cả các chỉ số băm từ 0 đến 9:
     - Tại chỉ số 0: Không có phần tử, thêm 0 vào danh sách.
     - Tại chỉ số 1: Không có phần tử, thêm 0 vào danh sách.
     - Tại chỉ số 2: Có một phần tử `<2, "Apple">`, thêm 1 vào danh sách.
     - Tại chỉ số 3: Không có phần tử, thêm 0 vào danh sách.
     - Tại chỉ số 4: Không có phần tử, thêm 0 vào danh sách.
     - Tại chỉ số 5: Có hai phần tử `<5, "Orange">` và `<15, "Banana">`, thêm 2 vào danh sách.
     - Tại chỉ số 6: Không có phần tử, thêm 0 vào danh sách.
     - Tại chỉ số 7: Không có phần tử, thêm 0 vào danh sách.
     - Tại chỉ số 8: Không có phần tử, thêm 0 vào danh sách.
     - Tại chỉ số 9: Không có phần tử, thêm 0 vào danh sách.
   - Danh sách kết quả sẽ là: `[0, 0, 1, 0, 0, 2, 0, 0, 0, 0]`.
   - Trả về danh sách liên kết chứa số phần tử tại từng địa chỉ.

3.3 API của DLinkedList

1. **add(T e)**: Thêm phần tử `e` vào cuối danh sách.
2. **add(int index, T e)**: Thêm phần tử `e` vào vị trí chỉ định bởi `index`.
3. **removeAt(int index)**: Xóa và trả về phần tử tại vị trí `index`.
4. **removeItem(T item, void (*removeItemData)(T)=0)**: Xóa phần tử `item` khỏi danh sách và có thể xóa dữ liệu kèm theo.
5. **empty()**: Trả về `true` nếu danh sách rỗng, ngược lại trả về `false`.
6. **size()**: Trả về số lượng phần tử trong danh sách.
7. **clear()**: Xóa tất cả các phần tử, đưa danh sách về trạng thái ban đầu.
8. **get(int index)**: Trả về tham chiếu đến phần tử tại vị trí `index`.
9. **indexOf(T item)**: Trả về vị trí của phần tử `item`, hoặc `-1` nếu không tìm thấy.
10. **contains(T item)**: Trả về `true` nếu danh sách chứa phần tử `item`, ngược lại trả về `false`.
11. **toString(string (*item2str)(T&)=0)**: Trả về chuỗi mô tả danh sách bằng cách chuyển từng phần tử thành chuỗi.

3.4 xMap

```
void ensureLoadFactor(int minCapacity);
void rehash(int newCapacity);
void removeInternalData();
void copyMapFrom(const xMap<K, V>& map);
void moveEntries(DLinkedList<Entry*>* oldTable, int oldCapacity,
                 DLinkedList<Entry*>* newTable, int newCapacity);
```

1. **moveEntries :** Di chuyển tất cả các phần tử trong bảng băm cũ (oldTable) sang bảng băm mới (newTable).
   - **oldTable:** Bảng băm cũ chứa các mục cần di chuyển.
   - **oldCapacity:** Dung lượng của bảng cũ.
   - **newTable:** Bảng băm mới mà các mục sẽ được di chuyển tới.
   - **newCapacity:** Dung lượng của bảng mới.
2. **ensureLoadFactor :** Đảm bảo rằng số lượng phần tử trong bảng băm không vượt quá giới hạn của `loadFactor * capacity`.
   - **current_size:** Kích thước hiện tại của bảng băm, tức là số phần tử đã được lưu trữ.
3. **rehash :** Tạo một bảng băm mới với dung lượng lớn hơn và di chuyển tất cả các phần tử từ bảng băm cũ sang bảng mới.
   - **newCapacity:** Dung lượng mới của bảng băm.
4. **removeInternalData :** Xóa toàn bộ dữ liệu trong bảng băm, bao gồm các khóa, giá trị và các mục (entry), và giải phóng bộ nhớ của bảng băm.
   - Không có tham số cụ thể, nhưng có hai biến chức năng:
   - **deleteKeys:** Hàm được gọi để xóa dữ liệu khóa (nếu có).
   - **deleteValues:** Hàm được gọi để xóa dữ liệu giá trị (nếu có).

5. **copyMapFrom :** Xóa bảng băm hiện tại và sao chép tất cả các mục từ bảng băm đầu vào (sao chép nông).
    - **map:** Bảng băm đầu vào từ đó các mục sẽ được sao chép.