LẬP TRÌNH, GIẢI THÍCH, ĐỊNH NGHĨA, VÀ TRỰC QUAN CHƯƠNG TRÌNH LRU CACHE (LEAST RECENTLY USED) C++

Dang Phuc An Khang
ankhangluonvuituoi@gmail.com
24/06/2024

Giới thiệu

Chương trình này được thiết kế để quản lý một bộ nhớ đệm `LRU (Least Recently Used)` bằng cách sử dụng danh sách liên kết đôi và bảng băm. Bộ nhớ đệm `LRU` giúp lưu trữ các mục và loại bỏ các mục ít được sử dụng gần đây nhất khi bộ nhớ đệm đầy.

1. File `header.h`

Định nghĩa các lớp

1. **Lớp `Node`**

Lớp `Node` đại diện cho một nút trong danh sách liên kết đôi, chứa thông tin về khóa `(key)`, giá trị `(value)`, và các con trỏ trỏ tới nút trước `(prev)` và nút sau `(next)` trong danh sách.

```
#ifndef HEADER_H
#define HEADER_H

#include <vector>
#include <string>
#include <iostream>
#include <iomanip>
using namespace std;

// Khai báo lớp Node cho danh sách liên kết đôi
class Node {
public:
    int key;
    string value;
    Node* prev;
    Node* next;

    Node(int k, const string& v);
};
```

- **Thuộc tính**:
  - `int key`: Khóa của `Node`, dùng để định danh `Node` trong danh sách và bảng băm.
  - `string value`: Giá trị của `Node`, lưu trữ thông tin liên quan đến khóa.
  - `Node* prev`: Con trỏ trỏ tới `Node` trước đó trong danh sách liên kết đôi.
  - `Node* next`: Con trỏ trỏ tới `Node` kế tiếp trong danh sách liên kết đôi.
- **Hàm khởi tạo**:
  - `Node(int k, const string& v)`: Khởi tạo một `Node` mới với khóa k và giá trị v.

Trực quan lớp `Node`:

Hãy hình dung mỗi `Node` như một hộp lưu trữ với 4 phần: khóa, giá trị, con trỏ tới `Node` trước, và con trỏ tới `Node` sau.

prev	key	value	next
*	1	"A"	*

2. **Lớp `DoublyLinkedList`**

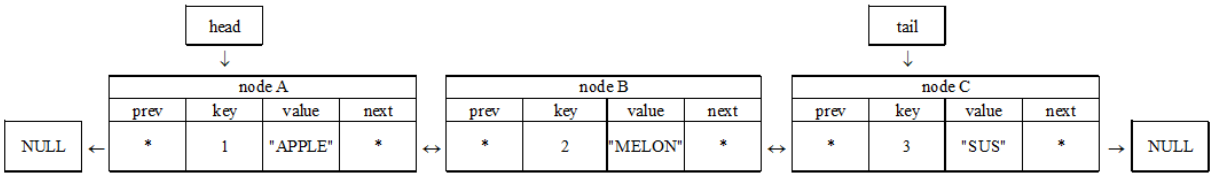
Lớp `DoublyLinkedList` quản lý các `Node` trong một danh sách liên kết đôi. Lớp này cung cấp các phương thức để thêm, di chuyển và loại bỏ các `Node` trong danh sách.

```
class DoublyLinkedList {
public:
    Node* head;  // Con trỏ tới node đầu tiên trong danh sách
    Node* tail;  // Con trỏ tới node cuối cùng trong danh sách

    DoublyLinkedList();  // Hàm khởi tạo

    void moveToFront(Node* node);  // Di chuyển một node lên đầu danh sách
    void addToFront(Node* node);   // Thêm một node vào đầu danh sách
    void remove(Node* node);       // Loại bỏ một node khỏi danh sách
};
```

- **Thuộc tính**:
  - `Node* head`: Con trỏ trỏ tới `Node` đầu tiên trong danh sách.
  - `Node* tail`: Con trỏ trỏ tới `Node` cuối cùng trong danh sách.
- **Hàm khởi tạo**:
  - `DoublyLinkedList()`: Khởi tạo một danh sách liên kết đôi mới, thiết lập `head` và `tail` là `nullptr`.
- **Phương thức `moveToFront`**:
  - `void moveToFront(Node* Node)`: Di chuyển một `Node` đã có trong danh sách lên đầu danh sách.
- **Phương thức `addToFront`**:
  - `void addToFront(Node* Node)`: Thêm một `Node` vào đầu danh sách.
- **Phương thức `remove`**:
  - `void remove(Node* Node)`: Loại bỏ một `Node` khỏi danh sách.

Trực quan lớp `DoublyLinkedList`:

Hãy hình dung danh sách liên kết đôi như một chuỗi các hộp `(Node)`, mỗi hộp chứa 4 phần và các hộp được liên kết với nhau bằng các con trỏ `prev` và `next`.



3. Lớp `HashTable`

Lớp `HashTable` được thiết kế để lưu trữ các con trỏ đến các đối tượng `Node` được sử dụng trong danh sách liên kết đôi. Nó cung cấp các phương thức để băm các khóa, truy cập các `Node` bằng khóa, và xóa các `Node` khỏi bảng băm.

```
const int TABLE_SIZE = 1000;

class HashTable {
private:
    vector<Node*> table;
    int hash(int key);

public:
    HashTable();
    Node*& operator[](int key);
    void erase(int key);
};
```

- **Thuộc tính**:
  - `vector<Node*> table`: `Vector` chứa các con trỏ tới các đối tượng `Node`.
  - `int hash(int key)`: Hàm băm để tính toán chỉ số lưu trữ cho khóa.
- **Hàm khởi tạo**:
  - `HashTable()`: Khởi tạo bảng băm với kích thước cố định là `TABLE_SIZE`.
- **Toán tử truy cập phần tử**:
  - `Node*& operator[](int key)`: Toán tử `[]` được nạp chồng để cho phép truy cập trực tiếp vào các phần tử của `table` thông qua một `key`.
- **Phương thức xóa một phần tử**:
  - `void erase(int key)`: Xóa một `Node` khỏi bảng băm.

Trực quan lớp `HashTable`:

Minh họa trực quan `HashTable`

Giả sử bảng băm có kích thước nhỏ hơn để dễ hình dung:

```
TABLE_SIZE = 10
```

Ví dụ về `HashTable` với các khóa và giá trị:

Giả sử bạn có các `Node` sau trong `DoublyLinkedList`:

1. `Node` A với `key` = 1 và `value` = "Apple"
2. `Node` B với `key` = 2 và `value` = "Banana"
3. `Node` C với `key` = 3 và `value` = "Cherry"

Bảng băm sẽ trông như thế nào?

Index	Node		
	name	**key**	**value**
0	NULL		
1	A	1	"Apple"
2	B	2	"Banana"
3	C	3	"Cherry"
4	NULL		
5	NULL		
6	NULL		
7	NULL		
8	NULL		
9	NULL		

Quá trình truy cập và thao tác với `HashTable`

1. **Thêm `Node` vào `HashTable`:**

- Khi bạn thêm `Node` A với **`key` = 1**, hàm băm tính toán **`hash(1) % 10` = 1.**
- `Node` A được lưu tại **`Index` 1** của bảng băm.

```
Node* newNode = new Node(1, "Apple");
hashTable[1] = newNode;
```

2. **Truy cập `Node` từ `HashTable`:**

- Khi bạn muốn truy cập `Node` với **`key` = 1**, bạn sử dụng toán tử `[]` của `HashTable`:
- **`Node* Node = HashTable[1];`**

3. **Xóa `Node` từ `HashTable`:**

- Khi bạn muốn xóa `Node` với **`key` = 1:**
- **`HashTable.erase(1);`**
- Bảng băm sau khi xóa `Node` A:

Index	Node		
	name	**key**	**value**
0	NULL		
1	NULL		
2	B	2	"Banana"
3	C	3	"Cherry"
4	NULL		
5	NULL		
6	NULL		
7	NULL		
8	NULL		
9	NULL		

4. **Lớp `LRUCache`**

Lớp `LRUCache` được thiết kế để quản lý một bộ nhớ đệm `LRU`. Bộ nhớ đệm này lưu trữ các mục và loại bỏ các mục ít được sử dụng gần đây nhất khi bộ nhớ đầy.

```
class LRUCache {
private:
    int capacity;            // Dung lượng tối đa của bộ nhớ đệm
    int size;                // Kích thước hiện tại của bộ nhớ đệm
    DoublyLinkedList list;   // Danh sách liên kết đôi để quản lý thứ tự các
mục
    HashTable map;           // Bảng băm để truy cập nhanh các mục

public:
    LRUCache(int cap);       // Constructor

    string get(int key);             // Lấy giá trị của một mục dựa trên khóa
    void put(int key, const string& value);  // Thêm hoặc cập nhật một mục
    void display();                   // Hiển thị nội dung của bộ nhớ đệm
};
```

- **Thuộc tính**:
  - `int capacity`: Dung lượng tối đa của bộ nhớ đệm.
  - `int size`: Kích thước hiện tại của bộ nhớ đệm.
  - `DoublyLinkedList list`: Danh sách liên kết đôi để quản lý thứ tự các mục.

    - `HashTable map`: Bảng băm để truy cập nhanh các mục.
- **Hàm khởi tạo**:
    - `LRUCache(int cap)`: Khởi tạo bộ nhớ đệm `LRU` với dung lượng tối đa là cap.
- **Phương thức `get`**:
    - `string get(int key)`: Lấy giá trị của một mục trong bộ nhớ đệm dựa trên khóa.
- **Phương thức `put`**:
    - `void put(int key, const string& value)`: Thêm một mục mới vào bộ nhớ đệm hoặc cập nhật giá trị của một mục đã tồn tại.
- **Phương thức `display`**:
    - `void display()`: Hiển thị nội dung của bộ nhớ đệm.

Trực quan lớp `LRUCache`:

Hãy hình dung bộ nhớ đệm `LRU` như một hệ thống quản lý sách trong thư viện. Sách được thêm vào đầu danh sách nếu chúng được truy cập hoặc thêm mới, và sách ít được sử dụng nhất sẽ bị loại bỏ khi danh sách đầy.

Mã hoàn chỉnh của `header.h` với các định nghĩa hàm:

```cpp
#define HEADER_H
#include <vector>
#include <string>
#include <iostream>
#include <iomanip>
using namespace std;

class Node {
public:
    int key;
    string value;
    Node* prev;
    Node* next;
    Node(int k, const string& v);
};

class DoublyLinkedList {
public:
    Node* head;
    Node* tail;
    DoublyLinkedList();
    void moveToFront(Node* node);
    void addToFront(Node* node);
    void remove(Node* node);
};

const int TABLE_SIZE = 1000;
class HashTable {
private:
    vector<Node*> table;
    int hash(int key);

public:
    HashTable();
    Node*& operator[](int key);
    void erase(int key);
};
```

```
class LRUCache {
private:
    int capacity;
    int size;
    DoublyLinkedList list;
    HashTable map;

public:
    LRUCache(int cap);
    string get(int key);
    void put(int key, const string& value);
    void display();
};

// Định nghĩa constructor của Node
Node::Node(int k, const string& v) : key(k), value(v), prev(nullptr),
next(nullptr) {}

// Định nghĩa constructor của DoublyLinkedList
DoublyLinkedList::DoublyLinkedList() : head(nullptr), tail(nullptr) {}

// Định nghĩa các hàm của DoublyLinkedList
void DoublyLinkedList::moveToFront(Node* node) {
    if (node == head) return;

    // Ngắt liên kết của node hiện tại với các node trước và sau nó trong
danh sách.
    if (node->prev) node->prev->next = node->next;
    if (node->next) node->next->prev = node->prev;
    if (node == tail) tail = tail->prev;

    // Đặt node này lên đầu danh sách liên kết.
    node->next = head;
    node->prev = nullptr;
    if (head) head->prev = node;
    head = node;
    if (!tail) tail = head;
}

void DoublyLinkedList::addToFront(Node* node) {
    node->prev = nullptr;
    node->next = head;
    if (head) head->prev = node;
    head = node;
    if (!tail) tail = node;
}
```

```cpp
void DoublyLinkedList::remove(Node* node) {
    if (node->prev) node->prev->next = node->next;
    if (node->next) node->next->prev = node->prev;
    if (node == head) head = node->next;
    if (node == tail) tail = node->prev;
    delete node;
}

// Định nghĩa Constructor của HashTable
HashTable::HashTable() : table(TABLE_SIZE, nullptr) {}

// Định nghĩa hàm băm của HashTable
int HashTable::hash(int key) {
    return key % TABLE_SIZE;
}

// Định nghĩa toán tử truy cập phần tử của HashTable
Node*& HashTable::operator[](int key) {
    return table[hash(key)];
}

// Định nghĩa phương thức xóa phần tử của HashTable
void HashTable::erase(int key) {
    int idx = hash(key);
    table[idx] = nullptr;
}

// Định nghĩa constructor của LRUCache
LRUCache::LRUCache(int cap) : capacity(cap), size(0) {}

// Định nghĩa các hàm của LRUCache
string LRUCache::get(int key) {
    Node* result = map[key];
    if (!result) {
        return ""; // Không tìm thấy mục
    } else {
        list.moveToFront(result);
        return result->value;
    }
}

void LRUCache::put(int key, const string& value) {
    Node* node = map[key];
    if (node) {
        node->value = value;
        list.moveToFront(node);
    } else {
        if (size == capacity) {
            Node* toRemove = list.tail;
            map.erase(toRemove->key);
            list.remove(toRemove);
            size--;
        }
```

```
}

void LRUCache::display() {
    Node* curr = list.head;
    cout << left << setw(10) << "ID" << setw(20) << "NAME" << "PRIORITY" <<
endl;
    cout << "----------------------------------------" << endl;
    int priority = 1;
    while (curr) {
        cout << left << setw(10) << curr->key << setw(20) << curr->value <<
priority << endl;
        curr = curr->next;
        priority++;
    }
    cout << endl;
}

#endif
```

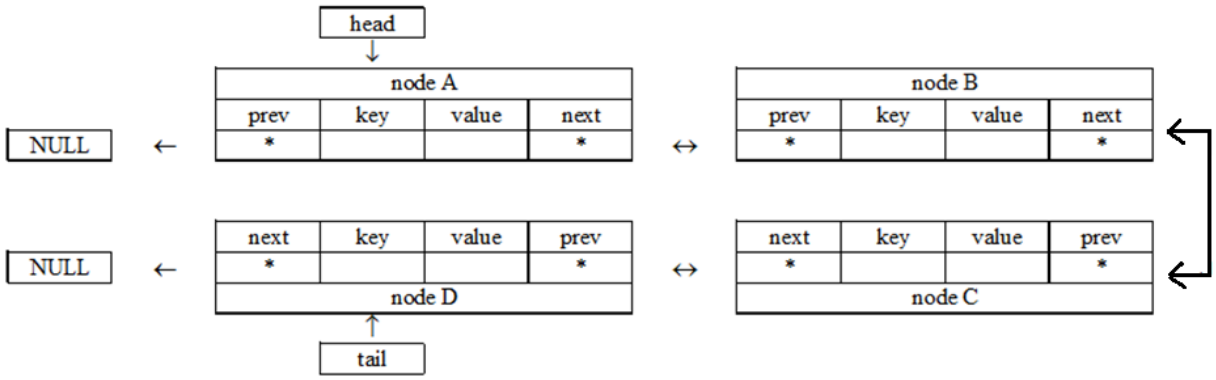
Giải thích trực quan cho từng phương thức:

1. Phương thức `moveToFront` của `DoublyLinkedList`

Mã nguồn của phương thức `moveToFront`

```
void DoublyLinkedList::moveToFront(Node* node) {
    if (node == head) return;  // Nếu node đã ở đầu danh sách, không cần di
chuyển

    // Ngắt liên kết của node hiện tại với các node trước và sau nó trong
danh sách
    if (node->prev) node->prev->next = node->next;
    if (node->next) node->next->prev = node->prev;
    if (node == tail) tail = tail->prev;  // Nếu node hiện tại là tail, cập
nhật tail

    // Đặt node này lên đầu danh sách liên kết
    node->next = head;
    node->prev = nullptr;
    if (head) head->prev = node;
    head = node;
    if (!tail) tail = head;  // Nếu tail rỗng (danh sách trống), tail cũng
trỏ tới node mới
}
```

- **Giả sử chúng ta có danh sách liên kết đôi sau:**



- **Node A**: `prev` = `nullptr`, `next` = B
- **Node B**: `prev` = A, `next` = C
- **Node C**: `prev` = B, `next` = D
- **Node D**: `prev` = C, `next` = `nullptr`

Bước 1: Kiểm tra nếu `Node` đã là `head`

```
if (node == head) return;
```

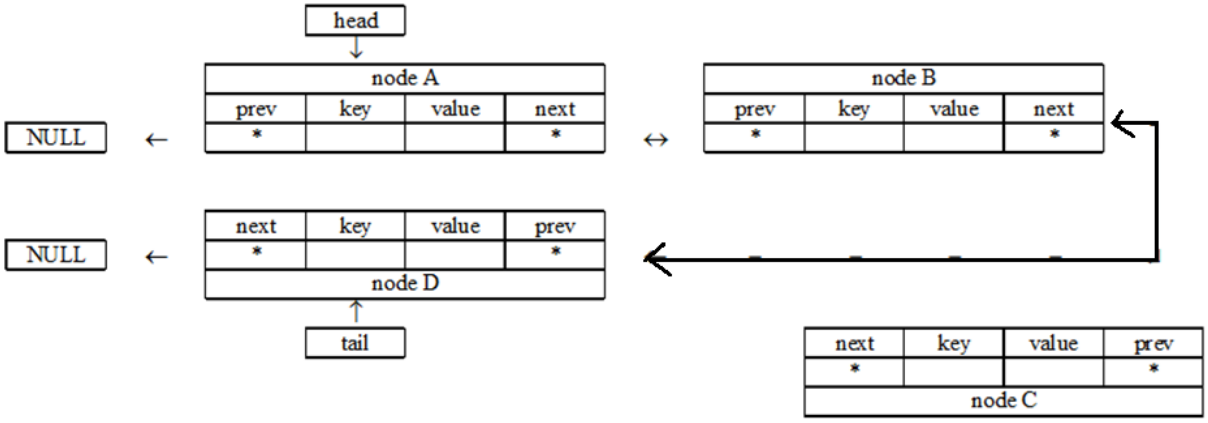
Nếu `Node` đã là `Node` đầu tiên `(head)`, không cần di chuyển và trả về ngay lập tức.

Bước 2: Ngắt liên kết của `Node` với các `Node` xung quanh

```
if (node->prev) node->prev->next = node->next;
if (node->next) node->next->prev = node->prev;
if (node == tail) tail = tail->prev;
```

- **Ngắt liên kết `prev`**:
  - Nếu `Node->prev` không phải là `nullptr`, cập nhật `Node->prev->next` để bỏ qua `Node` hiện tại và trỏ tới `Node->next`.
  - Ví dụ: B`->next` trỏ đến C thì B`->next` sẽ trỏ đến D.
- **Ngắt liên kết `next`**:
  - Nếu `Node->next` không phải là `nullptr`, cập nhật `Node->next->prev` để bỏ qua `Node` hiện tại và trỏ tới `Node->prev`.
  - Ví dụ: D`->prev` trỏ đến C thì D`->prev` sẽ trỏ đến B.
- **Cập nhật `tail`**:
  - Nếu `Node` hiện tại là `Node` cuối cùng `(tail)`, cập nhật `tail` để trỏ tới `Node` trước đó `(tail->prev)`.

Trạng thái sau bước 2:

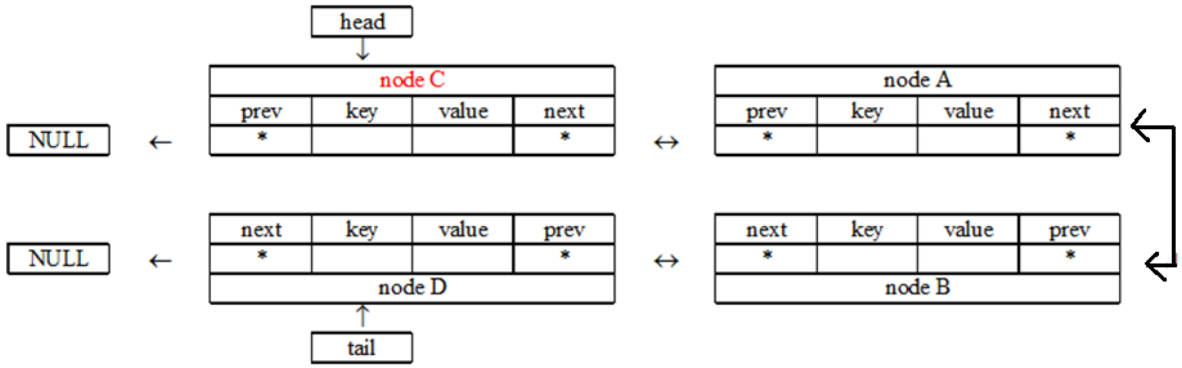


Bước 3: Đặt `Node` này lên đầu danh sách

```
node->next = head;
node->prev = nullptr;
if (head) head->prev = node;
head = node;
if (!tail) tail = head;
```

- **Cập nhật `next` và `prev` của `Node`**:
  - `Node->next` = `head`: `Node` hiện tại trỏ đến `Node` đầu tiên cũ.
  - `Node->prev` = `nullptr`: `Node` hiện tại không có `Node` trước nó.
- **Cập nhật `head` cũ**:
  - Nếu `head` không phải là `nullptr`, cập nhật `head->prev` để trỏ tới `Node` hiện tại.
- **Cập nhật `head` mới**:
  - `head` = `Node`: `Node` hiện tại trở thành `Node` đầu tiên trong danh sách.
- **Cập nhật `tail` nếu cần**:
  - Nếu danh sách ban đầu rỗng (`tail` là `nullptr`), `tail` cũng trỏ tới `Node` hiện tại.

Trạng thái cuối cùng:



Tóm tắt

- **Ngắt liên kết**: Ngắt liên kết của `Node` cần di chuyển với các `Node` trước và sau nó.
- **Cập nhật `next` và `prev`**: Đặt `next` của `Node` cần di chuyển trỏ đến `Node` đầu tiên cũ và `prev` của `Node` cần di chuyển là `nullptr`.
- **Cập nhật `head`**: Đặt `head` trỏ đến `Node` cần di chuyển.
- **Cập nhật `prev` của `Node` đầu tiên cũ**: Nếu danh sách không rỗng, đặt `prev` của `Node` đầu tiên cũ trỏ đến `Node` cần di chuyển.
- **Cập nhật `tail` nếu cần**: Nếu danh sách ban đầu rỗng, đặt `tail` trỏ đến `Node` cần di chuyển.

Phương thức `moveToFront` đảm bảo rằng `Node` được di chuyển lên đầu danh sách và các con trỏ `prev` và `next` của các `Node` liên quan được cập nhật chính xác để duy trì cấu trúc danh sách liên kết đôi.

2. Phương thức `addToFront` của `DoublyLinkedList`

Phương thức `addToFront` trong một `DoublyLinkedList` là một thủ tục thường được sử dụng để thêm một `Node` mới vào đầu danh sách liên kết đôi. Phương thức này cập nhật các con trỏ phù hợp để đảm bảo `Node` mới được thêm một cách đúng đắn và trở thành `Node` đầu tiên trong danh sách.

Mã nguồn của phương thức `addToFront`

```
void DoublyLinkedList::addToFront(Node* node) {
    node->prev = nullptr;  // Node mới không có node trước nó
    node->next = head;     // Node mới trỏ tới node hiện tại mà head đang trỏ tới
    if (head) head->prev = node; // Nếu head không rỗng, cập nhật prev của node hiện tại mà head đang trỏ tới
    head = node;           // Cập nhật head để trỏ tới node mới
    if (!tail) tail = node; // Nếu tail rỗng (danh sách trống), tail cũng trỏ tới node mới
}
```

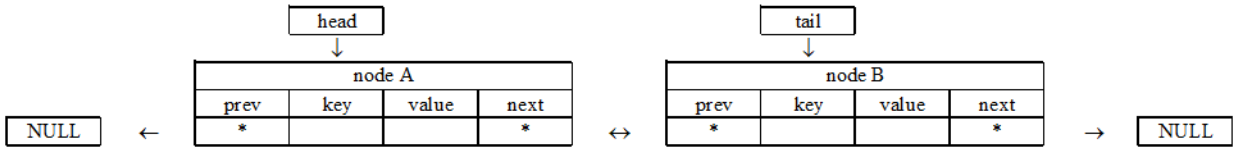
Giải thích từng bước

1. **Thiết lập `prev` của `Node` mới**:
   - `Node->prev = nullptr;`
   - `Node` mới sẽ được thêm vào đầu danh sách, do đó không có `Node` nào đứng trước nó. Con trỏ `prev` của `Node` mới sẽ được thiết lập là `nullptr`.
2. **Thiết lập `next` của `Node` mới và cập nhật `prev` của `head` cũ (nếu có)**:
   - `Node->next = head;`
   - Con trỏ `next` của `Node` mới trỏ tới `Node` hiện tại mà `head` đang chỉ tới, tức là `Node` đầu tiên của danh sách trước khi thêm `Node` mới.
   - `if (head) head->prev = Node;`
   - Nếu danh sách không rỗng (`head` không phải là `nullptr`), thì con trỏ `prev` của `Node` đầu tiên hiện tại sẽ được cập nhật để trỏ ngược lại tới `Node` mới.
3. **Cập nhật `head` để chỉ tới `Node` mới**:
   - `head = Node;`
   - Sau khi đã thiết lập các con trỏ `next` và `prev`, con trỏ `head` của danh sách được cập nhật để trỏ tới `Node` mới, làm cho `Node` này trở thành `Node` đầu tiên trong danh sách.
4. **Kiểm tra và cập nhật `tail` nếu danh sách ban đầu rỗng**:
   - `if (!tail) tail = Node;`
   - Nếu danh sách ban đầu rỗng (`tail` là `nullptr`), thì `tail` cũng cần được cập nhật để trỏ tới `Node` mới, vì giờ đây nó là `Node` duy nhất trong danh sách.

Trực quan hóa từng bước

Giả sử bạn bắt đầu với một danh sách có hai `Node` và muốn thêm một `Node` mới (`"Node N"`) vào đầu danh sách:

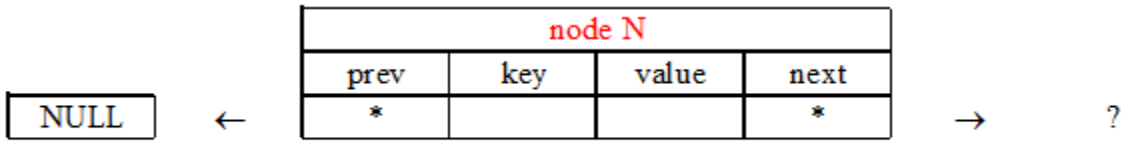
Ban đầu:



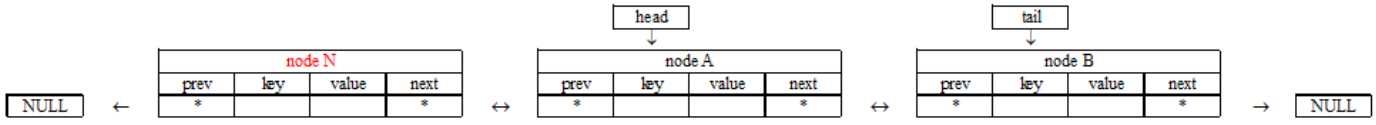
- **`Node` A**: `prev` = `nullptr`, `next` = B
- **`Node` B**: `prev` = A, `next` = `nullptr`

Thêm "`Node` N" vào đầu danh sách:

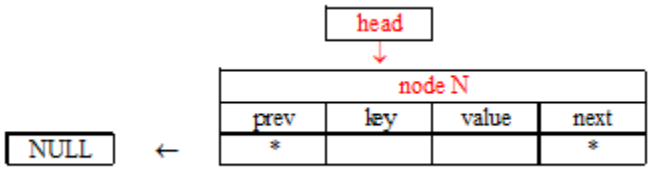
Bước 1: Thiết lập `prev` của `Node` N:



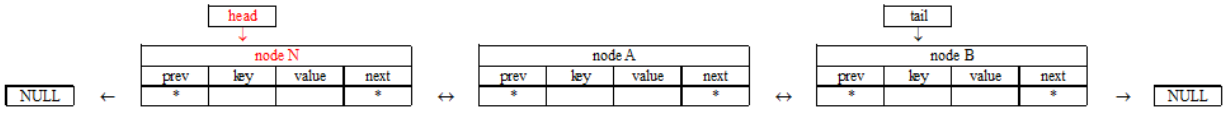
Bước 2: Thiết lập `next` của `Node` N và cập nhật `prev` của `Node` A:



Bước 3: Cập nhật `head` để chỉ tới `Node` N:



Kết quả:



- **Node N**: `prev` = `nullptr`, `next` = A
- **Node A**: `prev` = N, `next` = B
- **Node B**: `prev` = A, `next` = `nullptr`

Phương thức `addToFront` đảm bảo `Node` mới được thêm một cách nhanh chóng và hiệu quả vào đầu danh sách, và cập nhật các con trỏ để duy trì tính toàn vẹn của danh sách liên kết đôi.

3. Phương thức `remove` của `DoublyLinkedList`

Phương thức `remove` trong `DoublyLinkedList` được thiết kế để xóa một `Node` khỏi danh sách liên kết đôi. Quá trình này bao gồm việc ngắt liên kết `Node` đó với các `Node` lân cận và cập nhật con trỏ `head` hoặc `tail` nếu cần thiết.

Mã nguồn của phương thức `remove`

```
void DoublyLinkedList::remove(Node* node) {
    if (!node) return; // Kiểm tra node có tồn tại không

    // Ngắt liên kết của node với node trước và sau nó
    if (node->prev) node->prev->next = node->next;
    if (node->next) node->next->prev = node->prev;

    // Cập nhật head hoặc tail nếu cần
    if (node == head) head = node->next;
    if (node == tail) tail = node->prev;

    // Giải phóng bộ nhớ
    delete node;
}
```

Giải thích từng bước

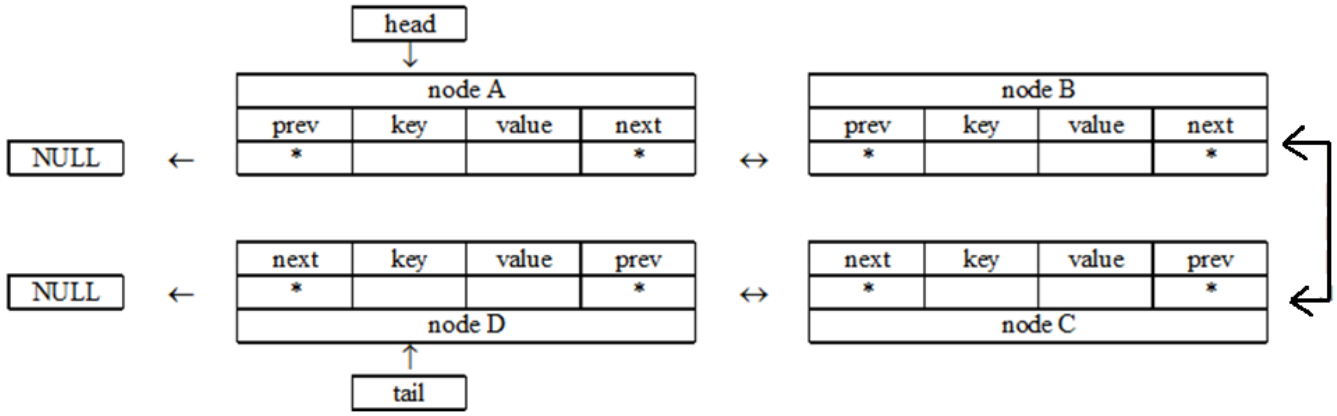
1. **Kiểm tra `Node`**:
    - Đầu tiên, kiểm tra xem `Node` có tồn tại không `(if (!Node) return;)`. Nếu `Node` không tồn tại, phương thức sẽ trả về ngay lập tức.
2. **Ngắt liên kết `Node`**:
    - Ngắt liên kết của `Node` với `Node` trước nó `(Node->prev)` và `Node` sau nó `(Node->next)`:
        - Nếu `Node` có `Node` trước `(Node->prev)`, cập nhật `next` của `Node` trước để trỏ đến `Node` sau của `Node` hiện tại: `Node->prev->next = Node->next`.
        - Nếu `Node` có `Node` sau `(Node->next)`, cập nhật `prev` của `Node` sau để trỏ đến `Node` trước của `Node` hiện tại: `Node->next->prev = Node->prev`.

3. **Cập nhật `head` và `tail`:**
    - Nếu `Node` là `head` của danh sách, cập nhật `head` để trỏ đến `Node` sau của `Node` hiện tại: `if (Node == head) head = Node->next`.
    - Nếu `Node` là `tail` của danh sách, cập nhật `tail` để trỏ đến `Node` trước của `Node` hiện tại: `if (Node == tail) tail = Node->prev`.
4. **Xóa `Node`:**
    - Cuối cùng, giải phóng bộ nhớ được chiếm dụng bởi `Node`: `delete Node`.

Trực quan hóa quá trình xóa `Node`

Giả sử bạn có một danh sách liên kết đôi như sau và bạn muốn xóa `Node` C:

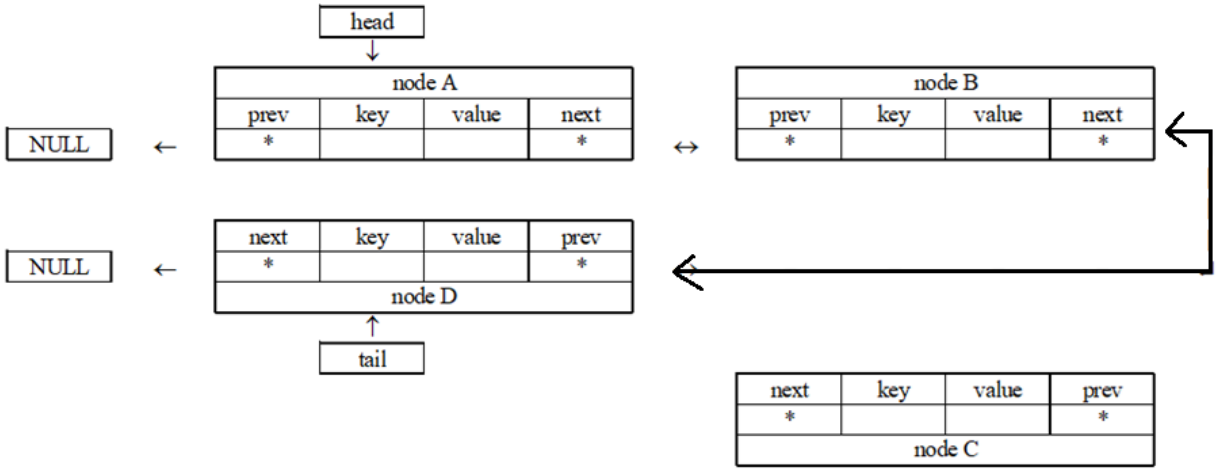
Trước khi xóa `Node` C:



- **`Node` A**: `prev` = `nullptr`, `next` = B
- **`Node` B**: `prev` = A, `next` = C
- **`Node` C**: `prev` = B, `next` = D
- **`Node` D**: `prev` = C, `next` = `nullptr`

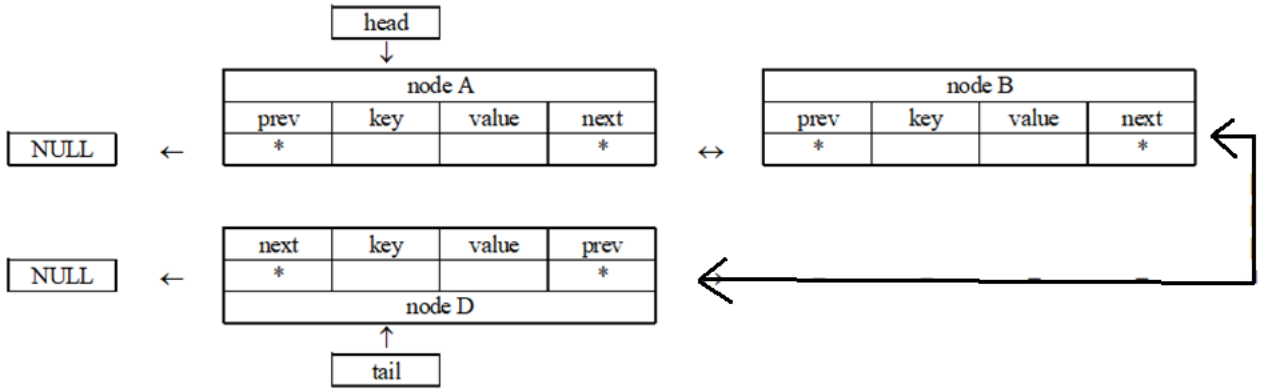
Thực hiện xóa `Node` C:

1. **Ngắt liên kết của `Node` C:**



2. **Cập nhật `head` và `tail` nếu cần**:
   - Trong trường hợp này, C không phải là `head` hoặc `tail`, nên không cần cập nhật.
3. **Giải phóng `Node` C**:
   - `delete` C;

Sau khi xóa `Node` C:



- **`Node` A**: `prev` = `nullptr`, `next` = B
- **`Node` B**: `prev` = A, `next` = D
- **`Node` D**: `prev` = B, `next` = `nullptr`

`Node` C được xóa khỏi danh sách một cách an toàn, và các liên kết giữa các `Node` còn lại được cập nhật để duy trì tính liên kết và tính toàn vẹn của danh sách. Các bước trên đảm

bảo rằng không có rò rỉ bộ nhớ và danh sách liên kết vẫn hoạt động chính xác sau khi `Node` bị xóa.

4. Phương thức khởi tạo `HashTable`

Phương thức khởi tạo của `HashTable` trong một cài đặt thông thường thiết lập bảng băm để sẵn sàng cho việc lưu trữ các `Node` hoặc các giá trị dựa trên khóa. Phương thức này thường khởi tạo một mảng hoặc một `vector` với một kích thước nhất định, thường được gọi là kích thước của bảng băm, và thiết lập mỗi phần tử trong mảng hoặc `vector` đó để ban đầu không có giá trị (tức là `nullptr`).

Mã nguồn của phương thức khởi tạo `HashTable`

```
class HashTable {
private:
    vector<Node*> table;  // Vector lưu trữ các con trỏ tới Node
    int hash(int key) {   // Hàm băm: Tính chỉ số băm dựa trên khóa
        return key % table.size();
    }

public:
    HashTable(int size = 100) : table(size, nullptr) {  // Khởi tạo vector
với kích thước và giá trị ban đầu là nullptr
    }

    // Các phương thức khác...
};
```

Giải thích từng bước

1. **Khởi tạo `vector table`:**
    - `Vector table` được khởi tạo với kích thước cụ thể `(size)` và mỗi phần tử trong `vector` này ban đầu được thiết lập là `nullptr`. Giá trị này cho biết ban đầu không có `Node` nào được lưu trữ trong bảng băm.
    - Kích thước của `vector` thường được chọn dựa trên yêu cầu hiệu suất và mức độ chiếm dụng bộ nhớ mong muốn. Một bảng băm lớn hơn có thể giảm bớt xung đột nhưng sẽ tiêu thụ nhiều bộ nhớ hơn.
2. **Hàm băm `hash(int key)`:**
    - Hàm này là một phương thức riêng của lớp `HashTable`, được sử dụng để xác định vị trí trong `vector` mà một `Node` sẽ được lưu trữ dựa trên khóa của nó.
    - Thông thường, hàm băm tính chỉ số bằng cách lấy phần dư của phép chia khóa cho kích thước của bảng băm `(key % table.size())`. Điều này đảm bảo rằng chỉ số luôn nằm trong phạm vi của kích thước `vector`.

Trực quan hóa

Giả sử bạn tạo một `HashTable` với kích thước là 10. Bảng băm sẽ trông như sau khi khởi tạo:

```
HashTable:
Index: 0 1 2 3 4 5 6 7 8 9
Value: [ ][ ][ ][ ][ ][ ][ ][ ][ ][ ]
```

- Mỗi [ ] biểu thị một vị trí trong `vector`, ban đầu tất cả đều là `nullptr`.

Sử dụng phương thức khởi tạo

Khi tạo một đối tượng `HashTable`, bạn có thể chỉ định kích thước của nó hoặc sử dụng giá trị mặc định:

```
HashTable myHashTable;          // Sử dụng kích thước mặc định là 100
HashTable customHashTable(50); // Tạo bảng băm với kích thước là 50
```

Lợi ích

Phương thức khởi tạo này giúp đảm bảo rằng bảng băm sẵn sàng để lưu trữ và truy cập các `Node` một cách hiệu quả ngay từ khi đối tượng được tạo ra. Điều này làm giảm nguy cơ xảy ra lỗi và tăng cường hiệu suất của các thao tác trên bảng băm.

5. Hàm băm `Hash` của `HashTable`

Hàm băm là một thành phần cốt lõi trong việc thiết kế và thực thi của một bảng băm. Mục đích của hàm băm là chuyển đổi một khóa đầu vào (thường là một số nguyên, chuỗi, hoặc bất kỳ đối tượng nào có thể được biểu diễn dưới dạng chuỗi) thành một chỉ số mảng (`Index`) được sử dụng để truy cập vào vị trí thích hợp trong bảng băm.

Mục đích và Yêu cầu của Hàm Băm

1. **Hiệu Quả**: Hàm băm phải nhanh chóng tính toán chỉ số từ khóa mà không gây ra độ trễ đáng kể.
2. **Phân Bố Đồng Đều**: Chỉ số được tạo ra từ hàm băm phải phân bố đều các khóa trên toàn bộ bảng băm để giảm thiểu xung đột.
3. **Nhất Quán**: Cho cùng một đầu vào, hàm băm luôn trả về cùng một chỉ số.

Mã Nguồn Cụ Thể

Xem xét một hàm băm đơn giản cho `HashTable` lưu trữ các `Node` dựa trên khóa kiểu nguyên:

```
int hash(int key) {
    return key % table.size();
}
```

Giải Thích

- **Chia lấy dư (%)**: Hàm băm sử dụng phép chia lấy dư của khóa cho kích thước của bảng băm để xác định chỉ số. Phương pháp này đảm bảo rằng chỉ số luôn nằm trong giới hạn của kích thước mảng.
- **table.size()**: Là kích thước của vector hoặc mảng được sử dụng để lưu trữ các Node. Sử dụng kích thước của bảng băm như một mẫu số đảm bảo rằng chỉ số không bao giờ vượt quá kích thước của bảng băm.

Trực quan Hàm Băm

Giả sử table.size() là 10, và bạn có các khóa như sau:

- **key**: 15
- **key**: 25
- **key**: 35

Áp dụng hàm băm:

- hash(15) = 15 % 10 = 5
- hash(25) = 25 % 10 = 5
- hash(35) = 35 % 10 = 5

Như bạn thấy trong ví dụ này, cả ba khóa đều dẫn đến cùng một chỉ số, là một ví dụ về xung đột băm. Xung đột này cần được giải quyết bằng các kỹ thuật như liên kết (chaining) hoặc địa chỉ mở (open addressing).

Tối Ưu Hóa Hàm Băm

Để cải thiện hàm băm:

- **Sử dụng Hằng Số Lớn**: Thay vì chỉ sử dụng kích thước của bảng, bạn có thể nhân khóa với một hằng số lớn trước khi lấy mô đun để cải thiện sự phân bố của chỉ số.
- **Kết Hợp Giá Trị Khóa**: Đối với khóa dạng chuỗi hoặc đối tượng phức tạp, bạn có thể kết hợp các giá trị của chúng (ví dụ: sử dụng các ký tự hoặc thuộc tính của đối tượng) để tạo ra một giá trị băm tốt hơn.

Kết Luận

Hàm băm đơn giản này là một điểm khởi đầu tốt nhưng có thể không phù hợp cho tất cả các tình huống, đặc biệt là khi kích thước bảng băm cố định hoặc khi các khóa không phân bố đồng đều. Việc lựa chọn hoặc thiết kế hàm băm phù hợp là rất quan trọng để đạt được hiệu suất tối ưu của bảng băm.

6. Toán tử `[]` của `HashTable`

Toán tử `[]` trong cài đặt `HashTable` là một tính năng rất hữu ích cho phép truy cập và quản lý các giá trị trong bảng băm một cách trực tiếp bằng khóa. Toán tử này thường được sử dụng để truy xuất và cập nhật giá trị tương ứng với một khóa cụ thể.

Mục đích của Toán tử `[]`

1. **Truy cập Dễ dàng**: Cung cấp một cách để truy cập và thao tác với dữ liệu trong bảng băm giống như truy cập một phần tử mảng thông thường.
2. **Đơn giản hóa Giao diện**: Người dùng có thể sử dụng toán tử `[]` để đọc hoặc ghi giá trị một cách rõ ràng mà không cần gọi phương thức rườm rà.
3. **Tự động xử lý Xung đột**: Có thể tích hợp các cơ chế xử lý xung đột trong việc thực thi toán tử này.

Mã Nguồn Ví dụ của Toán tử `[]`

```
class HashTable {
private:
    vector<Node*> table;
    int hash(int key) {
        return key % table.size();
    }

public:
    HashTable(int size = 100) : table(size, nullptr) {}

    Node*& operator[](int key) {
        int index = hash(key);
        if (!table[index]) {
            // Nếu không có node tại chỉ số này, tạo mới và trả về
            table[index] = new Node(key);
        }
        return table[index];
    }
};
```

Giải Thích Chi Tiết

1. **Tính Chỉ Số Băm**:

- `int Index = hash(key);`: Tính chỉ số băm của khóa bằng cách gọi hàm hash. Điều này đảm bảo rằng chỉ số luôn nằm trong phạm vi của mảng bảng băm.

2. **Truy cập hoặc Khởi Tạo `Node`**:
   - `if (!table[Index])`: Kiểm tra xem có `Node` nào tại chỉ số đã tính không. Nếu không có, một `Node` mới được khởi tạo và đặt vào vị trí này.
   - `return table[Index];`: Trả về tham chiếu tới `Node` tại chỉ số đó, cho phép cập nhật hoặc đọc giá trị.

Trường Hợp Sử Dụng

- **Đọc Giá Trị**: `Node* myNode = myHashTable[key];` sẽ trả về `Node` tương ứng với khóa nếu tồn tại, hoặc `nullptr` nếu không tồn tại.
- **Ghi Giá Trị**: `myHashTable[key] = newNode;` sẽ thiết lập giá trị của `Node` tại khóa được chỉ định. Nếu khóa chưa có, một `Node` mới sẽ được khởi tạo và giá trị sẽ được gán.

Toán tử `[]` là một công cụ mạnh mẽ trong việc cải thiện khả năng đọc và ghi dữ liệu trong bảng băm, giúp code trở nên gọn gàng và dễ đọc hơn.

7. Phương thức `erase` của `HashTable`

- **Mô tả**: Xóa một `Node` khỏi bảng băm bằng cách đặt giá trị tại chỉ số đó thành `nullptr`.

8. Phương thức khởi tạo `LRUCache`

- **Mô tả**: Khởi tạo bộ nhớ đệm `LRU` với dung lượng tối đa là cap.

9. Phương thức `get` của `LRUCache`

Phương thức `get` trong một `LRUCache (Least Recently Used` Cache`)` được sử dụng để truy xuất giá trị từ `Cache` dựa trên khóa cung cấp. Phương thức này không chỉ trả về giá trị mà còn cập nhật trạng thái của `Cache` để phản ánh rằng phần tử đã được truy cập, đưa nó lên đầu danh sách để biểu thị rằng đó là phần tử được sử dụng gần đây nhất.

Mục đích của Phương thức `get`

1. **Truy xuất Dữ liệu**: Lấy giá trị dựa trên khóa, nếu tồn tại trong cache.
2. **Cập nhật `LRU`**: Cập nhật thứ tự các phần tử trong `Cache` để phản ánh phần tử nào được sử dụng gần đây nhất.
3. **Hiệu suất**: Đảm bảo rằng các phần tử ít được sử dụng nhất có thể được xóa nếu cần thiết để làm chỗ cho phần tử mới, duy trì kích thước tối đa của cache.

Mã Nguồn Phương thức `get`

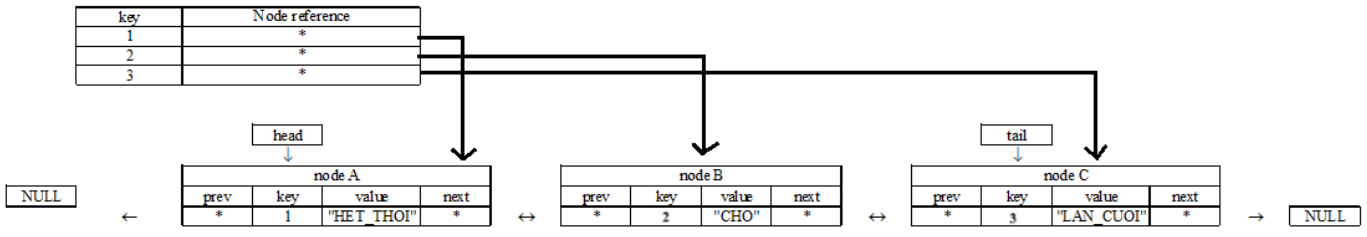
```
std::string LRUCache::get(int key) {
    Node* node = map.find(key);  // Tìm kiếm node dựa trên khóa trong
HashTable
    if (!node) {
        return "";  // Nếu không tìm thấy, trả về chuỗi rỗng
    }
    list.moveToFront(node);  // Di chuyển node đến đầu của danh sách liên kết
để đánh dấu nó là được sử dụng gần đây nhất
    return node->value;  // Trả về giá trị của node
}
```

Giải Thích Chi Tiết

1. **Tìm `Node`:**
   - `Node* Node` = map.find`(key);` - Sử dụng hàm find của `HashTable` để tìm `Node` dựa trên khóa được cung cấp.
   - Nếu `Node` không tồn tại trong `HashTable (Node == nullptr)`, phương thức trả về một chuỗi rỗng.
2. **Cập nhật `LRU (Least Recently Used)`:**
   - `list.moveToFront(Node);` - Nếu `Node` tồn tại, nó sẽ được di chuyển lên đầu của danh sách liên kết đôi để đánh dấu nó là phần tử được truy cập gần đây nhất.
   - Việc này đảm bảo rằng phần tử ít được sử dụng nhất sẽ luôn nằm ở cuối danh sách, sẵn sàng để bị loại bỏ nếu `Cache` đạt tới dung lượng tối đa.
3. **Trả về Giá Trị:**
   - `return Node->value;` - Cuối cùng, giá trị của `Node` được trả về.

Trực quan Hành Động `get`

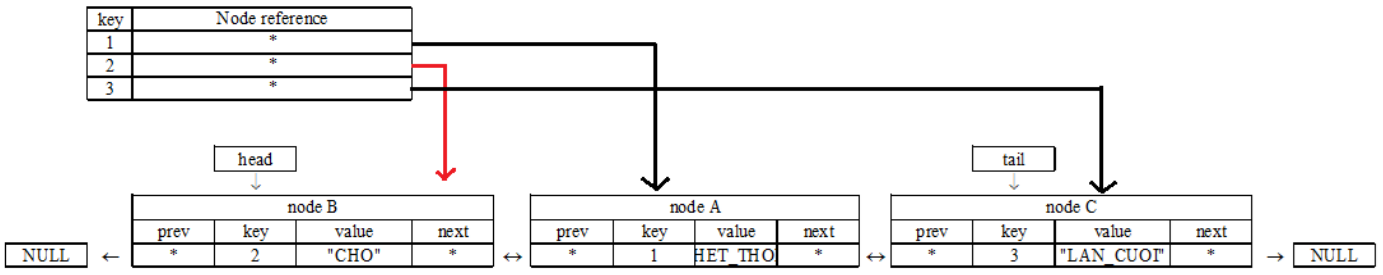
Giả sử `LRUCache` có các phần tử như sau trước khi gọi `get(2)`:



- **Gọi `get(2)`:**
   - `Node` với `Key`: 2 tìm thấy trong `HashTable`.

- `Node` này được di chuyển lên đầu danh sách, đánh dấu nó là được sử dụng gần đây nhất.

Sau khi gọi `get(2)`:



- `Node [Key: 2, Value: "Data2"]` giờ đây ở đầu danh sách, cho thấy nó là phần tử được truy cập gần đây nhất.

Phương thức `get` của `LRUCache` giúp đảm bảo rằng dữ liệu thường xuyên được truy cập sẽ luôn sẵn sàng và dễ dàng truy cập bằng cách giữ chúng ở gần đầu của danh sách. Cách tiếp cận này giúp cải thiện hiệu suất tổng thể của các hoạt động truy cập dữ liệu trong ứng dụng.

10. Phương thức `put` của `LRUCache`

Phương thức `put` trong `LRUCache` được sử dụng để thêm một phần tử mới hoặc cập nhật một phần tử hiện có trong bộ nhớ cache. Khi một phần tử mới được thêm vào, nếu bộ nhớ `Cache` đã đầy, phần tử ít được sử dụng nhất sẽ bị loại bỏ để nhường chỗ cho phần tử mới. Nếu phần tử đã tồn tại, giá trị của nó sẽ được cập nhật và phần tử đó sẽ được chuyển lên đầu danh sách để phản ánh rằng nó là phần tử được truy cập gần đây nhất.

Mục đích của Phương thức `put`

1. **Thêm hoặc Cập nhật Phần Tử**: Đảm bảo rằng tất cả các phần tử trong `Cache` đều là mới nhất và phản ánh các thay đổi mới nhất.
2. **Quản lý Kích thước Cache**: Duy trì kích thước của `Cache` không vượt quá giới hạn đã đặt bằng cách loại bỏ các phần tử ít được sử dụng nhất.
3. **Tối ưu Hóa Truy cập Dữ liệu**: Nâng cao hiệu suất truy cập dữ liệu bằng cách giữ cho các phần tử thường xuyên được sử dụng ở gần đầu danh sách.

Mã Nguồn Phương thức `put`

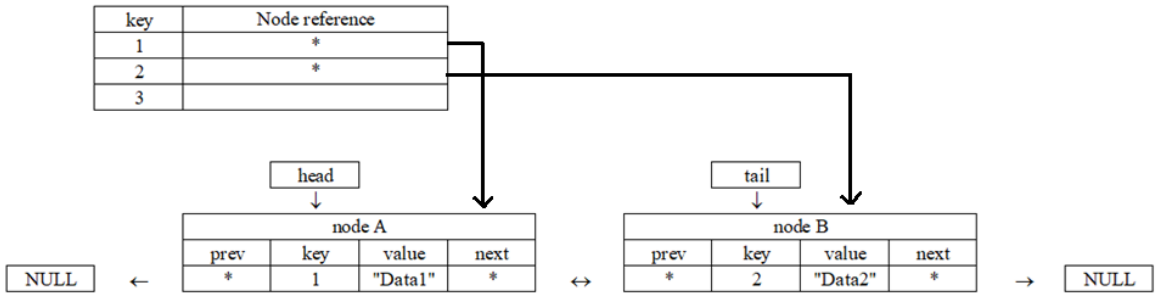
```
void LRUCache::put(int key, const std::string& value) {
    Node* node = map.find(key);  // Tìm node dựa trên khóa trong HashTable

    if (node) {
        // Nếu node đã tồn tại, cập nhật giá trị và di chuyển nó lên đầu danh
sách
        node->value = value;
        list.moveToFront(node);
    } else {
        // Nếu node không tồn tại, kiểm tra xem cache có đầy chưa
        if (list.size() == capacity) {
            // Nếu đầy, xóa phần tử ít được sử dụng nhất (tail của danh sách)
            Node* tail = list.tail;
            map.erase(tail->key);
            list.remove(tail);
        }
        // Thêm phần tử mới vào đầu danh sách
        Node* newNode = new Node(key, value);
        list.addToFront(newNode);
        map[key] = newNode;
    }
}
```

Giải Thích Chi Tiết

1. **Kiểm Tra và Cập nhật `Node` Hiện Có**:
   - `Node* Node` = map.find`(key);`: Tìm `Node` trong `Hash table`. Nếu `Node` tồn tại, nó sẽ được cập nhật và chuyển lên đầu danh sách để đánh dấu là mới được sử dụng.
   - `Node->value` = `value;`: Cập nhật giá trị của `Node`.
   - `list.moveToFront(Node);`: Di chuyển `Node` đến đầu danh sách liên kết đôi.
2. **Thêm `Node` Mới Nếu Cần**:
   - Kiểm tra kích thước của danh sách: `if (list.size()` == `capacity)`.
   - Nếu `Cache` đã đầy, loại bỏ phần tử ở cuối danh sách `(LRU)` và xoá tham chiếu của nó trong `Hash table`.
   - Tạo `Node` mới và thêm vào đầu danh sách. Cập nhật `Hash table` để bao gồm `Node` mới này.

Trực quan Hành Động `put`

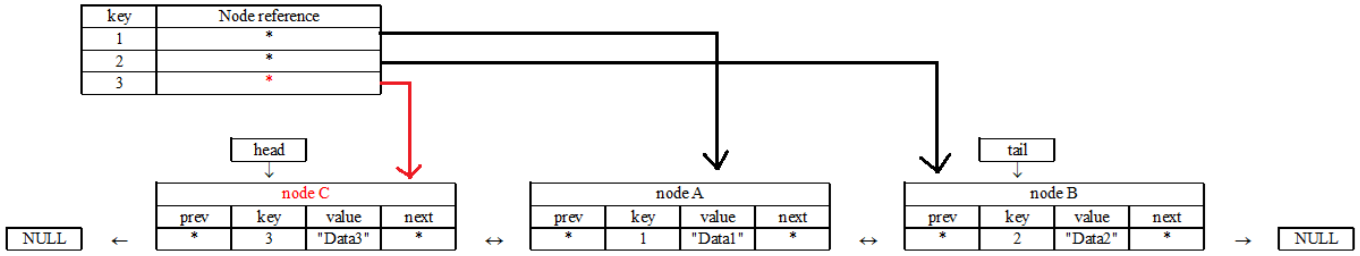
Giả sử `LRU Cache` ban đầu như sau:



Thực hiện `put(3, "Data3")`:

- **Thêm `Node` Mới**: Vì phần tử với `key` 3 không tồn tại và `Cache` chưa đầy, một `Node` mới được tạo và thêm vào đầu danh sách.
- **Cập nhật `Hash Table`**: Thêm khóa 3 và tham chiếu tới `Node` mới vào `Hash table`.

Kết quả:

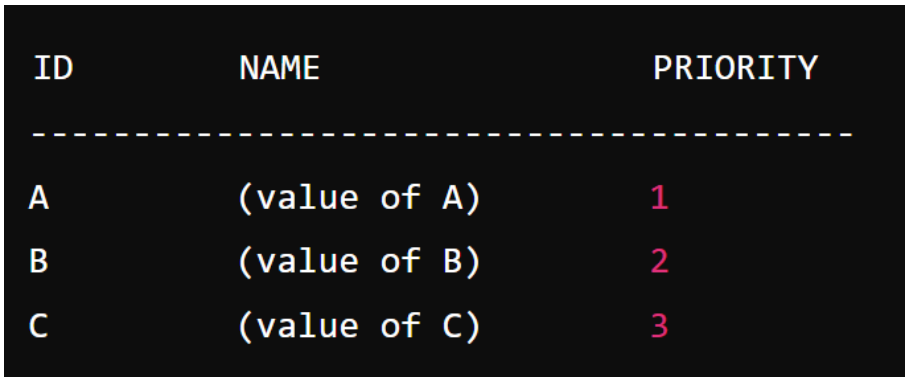


Phương thức `put` đảm bảo rằng `LRU Cache` luôn cập nhật và quản lý kích thước của nó một cách hiệu quả, làm cho việc truy cập dữ liệu nhanh chóng và phù hợp với mô hình sử dụng gần đây nhất.

11. Phương thức `display` của `LRUCache`

- **Mô tả**: Hiển thị nội dung của bộ nhớ đệm, bao gồm khóa, giá trị và mức độ ưu tiên của các mục theo thứ tự từ mục được sử dụng gần đây nhất đến ít được sử dụng nhất.
- **Trực quan**:

- Giả sử bộ nhớ đệm chứa các mục A, B, C với A là mục được sử dụng gần đây nhất.
- Gọi `display()` sẽ hiển thị:



2. File `main.cpp`

Chương trình chính

Chương trình chính trong `main.cpp` quản lý tương tác người dùng với bộ nhớ đệm `LRU` thông qua một menu đơn giản, cho phép người dùng thêm mục mới, truy xuất mục, hiển thị bộ nhớ đệm và thoát khỏi chương trình.

```cpp
#include "header.h"
#include <iostream>
#include <cstdlib>
#include <iomanip>

using namespace std;

void pauseAndClear() {
    cout << "Press Enter to continue...";
    cin.ignore();
    cin.get();
    system("CLS");
}

void menu() {
    cout << "Library Management System using LRU Cache\n";
    cout << "1. Add/Update a book\n";
    cout << "2. Get a book\n";
    cout << "3. Display cache\n";
    cout << "4. Exit\n";
    cout << "Choose an option: ";
}
        case 1:
            cout << "Enter book ID and book's name: ";
            cin >> key;
            cin.ignore();
            getline(cin, value);
            cache.put(key, value);
            cout << "Book added/updated.\n";
            pauseAndClear();
            break;
        case 2:
            cout << "Enter book ID: ";
            cin >> key;
            value = cache.get(key);
            if (value.empty()) {
                cout << "Book not found in cache.\n";
            } else {
```

```
                cout << "Book's name: " << value << endl;
            }
            pauseAndClear();
            break;
        case 3:
            cout << "Current cache content: \n";
            cache.display();
            pauseAndClear();
            break;
        case 4:
            cout << "Exiting...\n";
            return 0;
        default:
            cout << "Invalid option. Please try again.\n";
            pauseAndClear();
        }
    }

    return 0;
}
```

Giải thích trực quan cho `main.cpp`:

- **Menu chính**:
  - Người dùng được yêu cầu nhập một lựa chọn từ menu để tương tác với bộ nhớ đệm `LRU`.
  - Các lựa chọn bao gồm thêm hoặc cập nhật một cuốn sách, truy xuất một cuốn sách, hiển thị nội dung của bộ nhớ đệm, và thoát khỏi chương trình.
- **Trực quan**:
  - **Thêm hoặc cập nhật một cuốn sách**: Người dùng nhập ID và tên sách. Nếu ID đã tồn tại trong bộ nhớ đệm, thông tin sách sẽ được cập nhật; nếu không, sách mới sẽ được thêm vào.
  - **Truy xuất một cuốn sách**: Người dùng nhập ID của sách mà họ muốn truy xuất. Nếu sách có trong bộ nhớ đệm, thông tin sách sẽ được hiển thị; nếu không, một thông báo lỗi sẽ được hiển thị.
  - **Hiển thị nội dung của bộ nhớ đệm**: Danh sách các sách trong bộ nhớ đệm sẽ được hiển thị theo thứ tự từ sách được truy cập gần đây nhất đến sách ít được truy cập nhất.

Kết luận

File `main.cpp` cùng với `header.h` tạo thành một chương trình quản lý bộ nhớ đệm `LRU` hoàn chỉnh, cho phép người dùng tương tác một cách hiệu quả với dữ liệu trong bộ nhớ đệm thông qua một giao diện menu đơn giản. Chương trình này không chỉ cung cấp một cách tiện lợi để quản lý bộ nhớ đệm mà còn minh họa cách kết hợp các cấu trúc dữ liệu như danh sách liên kết đôi và bảng băm để thực hiện một giải pháp `LRU Cache` hiệu quả.