perplexity

Báo cáo kỹ thuật chi tiết: Cấu trúc dữ liệu Danh sách liên kết (Linked List) trong C++

1. Tổng quan và Định nghĩa

Danh sách liên kết là gì?

Danh sách liên kết (Linked List) là một cấu trúc dữ liệu tuyến tính động, trong đó các phần tử (được gọi là **nodes**) không được lưu trữ tại các vị trí bộ nhớ liền kề. Thay vào đó, mỗi node chứa dữ liệu và một **con trỏ (pointer)** trỏ đến node tiếp theo trong chuỗi[1][2][3].

Bản chất của danh sách liên kết là:

- **Cấu trúc động**: Kích thước có thể thay đổi trong quá trình thực thi chương trình
- **Lưu trữ phi liên tục**: Các node có thể nằm ở bất kỳ đâu trong bộ nhớ heap
- **Liên kết thông qua con trỏ**: Các node được kết nối với nhau thông qua địa chỉ bộ nhớ

So sánh trực diện: Danh sách liên kết vs Mảng

Đặc điểm	Mảng (Array)	Danh sách liên kết
Cấp phát bộ nhớ	Liên tục (contiguous)	Rải rác (scattered)
Vị trí lưu trữ	Stack/Heap (tĩnh)	Heap (động)
Kích thước	Cố định tại compile time	Thay đổi tại runtime
Truy cập ngẫu nhiên	Có (O(1))	Không (O(n))
Cache locality	Tốt	Kém
Overhead bộ nhớ	Thấp	Cao (con trỏ)
Tính linh hoạt	Hạn chế	Cao

Bảng so sánh độ phức tạp thời gian (Big-O):

Thao tác	Mảng (Array)	Danh sách liên kết đơn
Truy cập phần tử (theo chỉ số)	O(1)	O(n)
Tìm kiếm phần tử	O(n)	O(n)
Chèn ở đầu	O(n)	O(1)
Chèn ở cuối	O(1)	O(1)*
Chèn ở vị trí k	O(n)	O(n)

Thao tác	Mảng (Array)	Danh sách liên kết đơn
Xóa ở đầu	O(n)	O(1)
Xóa ở cuối	O(1)	O(n)
Xóa ở vị trí k	O(n)	O(n)

Chú thích: O(1) khi có pointer đến cuối danh sách (pTail)

2. Cấu trúc nền tảng - Node

Thành phần của Node

Mỗi node trong danh sách liên kết bao gồm hai thành phần chính[1][2]:

1. **Data (Dữ liệu)**: Lưu trữ thông tin thực tế của phần tử
2. **Pointer/Link (Con trỏ)**: Lưu địa chỉ của node tiếp theo trong danh sách

Định nghĩa struct Node trong C++

```
// Cấu trúc Node cơ bản cho danh sách liên kết đơn
struct Node {
    int data;           // Phần dữ liệu (có thể là bất kỳ kiểu dữ liệu nào)
    Node* next;         // Con trỏ trỏ đến node tiếp theo

    // Constructor
    Node(int value) : data(value), next(nullptr) {}
};

// Hoặc sử dụng class
class Node {
public:
    int data;
    Node* next;

    Node(int value) : data(value), next(nullptr) {}
};
```

3. Các loại Danh sách liên kết phổ biến

3.1 Danh sách liên kết đơn (Singly Linked List)

Cấu trúc: Mỗi node chỉ có một con trỏ trỏ đến node tiếp theo[1][2].

Sơ đồ minh họa:

```
[Data|Next] -> [Data|Next] -> [Data|Next] -> NULL
     Node1         Node2         Node3
```

Đặc điểm:

- Duyệt chỉ theo một hướng (từ đầu đến cuối)
- Tiết kiệm bộ nhớ (ít con trỏ hơn)
- Thao tác đơn giản hơn

3.2 Danh sách liên kết đôi (Doubly Linked List)

Cấu trúc: Mỗi node có hai con trỏ - một trỏ đến node tiếp theo và một trỏ đến node trước đó[2].

Sơ đồ minh họa:

```
NULL <- [Prev|Data|Next] <-> [Prev|Data|Next] <-> [Prev|Data|Next] -> NULL
              Node1                Node2                Node3
```

Ưu điểm so với liên kết đơn:

- Duyệt được theo cả hai hướng (tiến và lùi)
- Xóa node dễ dàng hơn khi đã biết địa chỉ node
- Thích hợp cho các ứng dụng cần duyệt ngược (undo/redo)

3.3 Danh sách liên kết vòng (Circular Linked List)

Cấu trúc: Node cuối cùng trỏ về node đầu tiên, tạo thành một vòng tròn[4].

Ứng dụng chính:

- Quản lý tài nguyên theo vòng tròn (round-robin scheduling)
- Buffer vòng (circular buffer)
- Trò chơi nhiều người chơi (lượt chơi)

4. Các thao tác cơ bản trên Danh sách liên kết đơn

4.1 Cấu trúc LIST và khởi tạo

```
struct LIST {
    Node* pHead;    // Con trỏ đến node đầu tiên
    Node* pTail;    // Con trỏ đến node cuối cùng
};

// Khởi tạo danh sách rỗng
void CreateEmptyList(LIST& L) {
    L.pHead = nullptr;
    L.pTail = nullptr;
}

// Tạo node mới
Node* CreateNode(int x) {
```

```
    Node* p = new Node(x);
    if (p == nullptr) {
        exit(1);    // Không đủ bộ nhớ
    }
    return p;
}
```

4.2 Duyệt danh sách (Traversal)

Thuật toán:

1. Bắt đầu từ node đầu tiên (pHead)
2. Duyệt qua từng node cho đến khi gặp NULL
3. Xử lý dữ liệu tại mỗi node

```
void PrintList(const LIST& L) {
    Node* current = L.pHead;

    if (current == nullptr) {
        cout << "Danh sách rỗng." << endl;
        return;
    }

    cout << "Danh sách: ";
    while (current != nullptr) {
        cout << current->data << " -> ";
        current = current->next;
    }
    cout << "NULL" << endl;
}
```

4.3 Chèn phần tử (Insertion)

4.3.1 Chèn vào đầu danh sách

Thuật toán[1]:

1. Tạo node mới
2. Nếu danh sách rỗng: node mới trở thành cả head và tail
3. Ngược lại: liên kết node mới với head cũ, cập nhật head

```
void AddHead(LIST& L, int x) {
    Node* p = CreateNode(x);

    if (L.pHead == nullptr) {
        // Danh sách rỗng
        L.pHead = p;
        L.pTail = p;
    } else {
        // Danh sách có phần tử
```

```
        p->next = L.pHead;
        L.pHead = p;
    }
}
```

4.3.2 Chèn vào cuối danh sách

```
void AddTail(LIST& L, int x) {
    Node* p = CreateNode(x);

    if (L.pHead == nullptr) {
        // Danh sách rỗng
        L.pHead = p;
        L.pTail = p;
    } else {
        // Danh sách có phần tử
        L.pTail->next = p;
        L.pTail = p;
    }
}
```

4.3.3 Chèn tại vị trí K

Thuật toán:

1. Tìm node tại vị trí K-1
2. Liên kết node mới với node tại vị trí K
3. Cập nhật liên kết của node tại vị trí K-1

```
void InsertAtPosition(LIST& L, int x, int position) {
    if (position == 0) {
        AddHead(L, x);
        return;
    }

    Node* current = L.pHead;
    for (int i = 0; i < position - 1 && current != nullptr; i++) {
        current = current->next;
    }

    if (current == nullptr) {
        cout << "Vị trí không hợp lệ!" << endl;
        return;
    }

    Node* newNode = CreateNode(x);
    newNode->next = current->next;
    current->next = newNode;

    // Cập nhật pTail nếu chèn vào cuối
    if (newNode->next == nullptr) {
        L.pTail = newNode;
```

```
    }
}
```

4.4 Xóa phần tử (Deletion)

4.4.1 Xóa phần tử đầu

Thuật toán[1]:

1. Kiểm tra danh sách có rỗng không
2. Lưu node cần xóa
3. Cập nhật head trỏ đến node tiếp theo
4. Giải phóng bộ nhớ

```
bool RemoveHead(LIST& L, int& x) {
    if (L.pHead == nullptr) {
        return false;   // Danh sách rỗng
    }

    Node* p = L.pHead;
    x = p->data;        // Lưu dữ liệu
    L.pHead = L.pHead->next;

    // Nếu danh sách chỉ có 1 phần tử
    if (L.pHead == nullptr) {
        L.pTail = nullptr;
    }

    delete p;
    return true;
}
```

4.4.2 Xóa phần tử có giá trị X

```
bool RemoveX(LIST& L, int x) {
    if (L.pHead == nullptr) {
        return false;
    }

    // Trường hợp xóa phần tử đầu
    if (L.pHead->data == x) {
        int temp;
        return RemoveHead(L, temp);
    }

    // Tìm node trước node cần xóa
    Node* current = L.pHead;
    while (current->next != nullptr && current->next->data != x) {
        current = current->next;
    }
```

```
    if (current->next == nullptr) {
        return false;   // Không tìm thấy
    }

    Node* nodeToDelete = current->next;
    current->next = nodeToDelete->next;

    // Cập nhật pTail nếu xóa phần tử cuối
    if (nodeToDelete == L.pTail) {
        L.pTail = current;
    }

    delete nodeToDelete;
    return true;
}
```

4.5 Tìm kiếm phần tử (Search)

Thuật toán[1]:

1. Duyệt từ đầu danh sách
2. So sánh dữ liệu tại mỗi node với giá trị cần tìm
3. Trả về địa chỉ node nếu tìm thấy, nullptr nếu không

```
Node* SearchNode(const LIST& L, int x) {
    Node* current = L.pHead;

    while (current != nullptr && current->data != x) {
        current = current->next;
    }

    return current;     // Trả về địa chỉ node hoặc nullptr
}

// Phiên bản trả về vị trí
int SearchPosition(const LIST& L, int x) {
    Node* current = L.pHead;
    int position = 0;

    while (current != nullptr) {
        if (current->data == x) {
            return position;
        }
        current = current->next;
        position++;
    }

    return -1;  // Không tìm thấy
}
```

5. Ứng dụng thực tế

5.1 Lịch sử duyệt web (Browser History)

Danh sách liên kết đôi được sử dụng để implement chức năng "Back" và "Forward" trong trình duyệt web[5][6]:

```
class BrowserHistory {
private:
    struct HistoryNode {
        string url;
        HistoryNode* prev;
        HistoryNode* next;

        HistoryNode(string u) : url(u), prev(nullptr), next(nullptr) {}
    };

    HistoryNode* current;

public:
    BrowserHistory(string homepage) {
        current = new HistoryNode(homepage);
    }

    void visit(string url) {
        HistoryNode* newPage = new HistoryNode(url);
        newPage->prev = current;
        current->next = newPage;
        current = newPage;
    }

    string back() {
        if (current->prev != nullptr) {
            current = current->prev;
        }
        return current->url;
    }

    string forward() {
        if (current->next != nullptr) {
            current = current->next;
        }
        return current->url;
    }
};
```

5.2 Chức năng Undo/Redo trong Text Editor

Sử dụng stack (được implement bằng linked list) để lưu trạng thái các thao tác[7][8]:

```
class TextEditor {
private:
    stack<string> undoStack;
```

```
    stack<string> redoStack;
    string currentText;

public:
    void write(char c) {
        undoStack.push(currentText);    // Lưu trạng thái hiện tại
        currentText += c;
        // Xóa redo stack khi có thao tác mới
        while (!redoStack.empty()) {
            redoStack.pop();
        }
    }

    void undo() {
        if (!undoStack.empty()) {
            redoStack.push(currentText);
            currentText = undoStack.top();
            undoStack.pop();
        }
    }

    void redo() {
        if (!redoStack.empty()) {
            undoStack.push(currentText);
            currentText = redoStack.top();
            redoStack.pop();
        }
    }
};
```

5.3 Quản lý bộ nhớ trong hệ điều hành

Hệ điều hành sử dụng linked list để quản lý các khối bộ nhớ tự do (free memory blocks)[9][10]:

- **Free list**: Danh sách các khối bộ nhớ có sẵn
- **Memory allocation**: Tìm và cấp phát khối phù hợp
- **Memory deallocation**: Trả khối về free list

5.4 Danh sách phát nhạc (Music Playlist)

Danh sách liên kết vòng thích hợp cho việc phát nhạc liên tục:

```
class MusicPlaylist {
private:
    struct Song {
        string title;
        Song* next;

        Song(string t) : title(t), next(nullptr) {}
    };

    Song* current;
```

```
public:
    void addSong(string title) {
        Song* newSong = new Song(title);
        if (current == nullptr) {
            current = newSong;
            current->next = current;    // Tạo vòng tròn
        } else {
            newSong->next = current->next;
            current->next = newSong;
        }
    }

    string nextSong() {
        if (current != nullptr) {
            current = current->next;
            return current->title;
        }
        return "";
    }
};
```

6. Chương trình mẫu hoàn chỉnh

```
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* next;

    Node(int value) : data(value), next(nullptr) {}
};

struct LIST {
    Node* pHead;
    Node* pTail;
};

class LinkedList {
private:
    LIST list;

public:
    LinkedList() {
        list.pHead = nullptr;
        list.pTail = nullptr;
    }

    ~LinkedList() {
        clear();
    }

    Node* createNode(int x) {
        return new Node(x);
    }
```

```
void addHead(int x) {
    Node* p = createNode(x);
    if (list.pHead == nullptr) {
        list.pHead = list.pTail = p;
    } else {
        p->next = list.pHead;
        list.pHead = p;
    }
}

void addTail(int x) {
    Node* p = createNode(x);
    if (list.pHead == nullptr) {
        list.pHead = list.pTail = p;
    } else {
        list.pTail->next = p;
        list.pTail = p;
    }
}

void display() {
    Node* current = list.pHead;
    while (current != nullptr) {
        cout << current->data << " -> ";
        current = current->next;
    }
    cout << "NULL" << endl;
}

bool removeX(int x) {
    if (list.pHead == nullptr) return false;

    if (list.pHead->data == x) {
        Node* temp = list.pHead;
        list.pHead = list.pHead->next;
        if (list.pHead == nullptr) {
            list.pTail = nullptr;
        }
        delete temp;
        return true;
    }

    Node* current = list.pHead;
    while (current->next != nullptr && current->next->data != x) {
        current = current->next;
    }

    if (current->next == nullptr) return false;

    Node* nodeToDelete = current->next;
    current->next = nodeToDelete->next;
    if (nodeToDelete == list.pTail) {
        list.pTail = current;
    }
    delete nodeToDelete;
```

```
        return true;
    }

    void clear() {
        while (list.pHead != nullptr) {
            Node* temp = list.pHead;
            list.pHead = list.pHead->next;
            delete temp;
        }
        list.pTail = nullptr;
    }
};

int main() {
    LinkedList myList;

    // Thêm phần tử
    myList.addTail(1);
    myList.addTail(2);
    myList.addTail(3);
    myList.addHead(0);

    cout << "Danh sách ban đầu: ";
    myList.display();

    // Xóa phần tử
    myList.removeX(2);
    cout << "Sau khi xóa 2: ";
    myList.display();

    return 0;
}
```

7. Kết luận

Danh sách liên kết là một cấu trúc dữ liệu mạnh mẽ và linh hoạt, đặc biệt phù hợp cho các ứng dụng cần:

- Kích thước dữ liệu thay đổi động
- Chèn/xóa phần tử thường xuyên ở đầu danh sách
- Không cần truy cập ngẫu nhiên theo chỉ số

Mặc dù có những hạn chế về cache locality và overhead bộ nhớ, linked list vẫn là công cụ không thể thiếu trong toolkit của lập trình viên, đặc biệt trong việc implement các cấu trúc dữ liệu phức tạp hơn như stack, queue, và graph.