Cấu trúc dữ liệu & Giải thuật trong C++

Chương 1. Ngôn ngữ lập trình C++

1.1. Giới thiệu ngôn ngữ lập trình C++

C++ là ngôn ngữ được phát triển bởi Bjarne Stroustrup vào năm 1979 tại Bell Labs, như là một mở rộng của ngôn ngữ C. Mục tiêu của Stroustrup là tạo ra một ngôn ngữ có thể cung cấp tính linh hoạt và hiệu suất cao của C, đồng thời bổ sung các tính năng hướng đối tượng.

Một số đặc điểm chính của C++:

- **Hướng đối tượng (OOP)**: C++ hỗ trợ các khái niệm OOP như lớp, đối tượng, thừa kế, đa hình và đóng gói.
- **Quản lý bộ nhớ**: C++ cho phép quản lý bộ nhớ bằng tay thông qua con trỏ và các toán tử `new` và `delete`.
- **Thư viện phong phú**: C++ cung cấp một số lượng lớn các thư viện chuẩn (STL) cho các cấu trúc dữ liệu và thuật toán.
- **Tính tương thích ngược**: C++ tương thích với C, tức là hầu hết mã C có thể được biên dịch và chạy trong môi trường C++.
- **Mẫu (Templates)**: C++ hỗ trợ lập trình tổng quát thông qua các mẫu, cho phép tạo ra mã linh hoạt và tái sử dụng.

1.2. Nhập xuất dữ liệu trong C++

C++ sử dụng thư viện iostream để nhập xuất dữ liệu. Thư viện này cung cấp các đối tượng như `cin`, `cout`,…

- `cin`: Nhập dữ liệu từ bàn phím (chuẩn đầu vào).
- `cout`: Xuất dữ liệu ra màn hình (chuẩn đầu ra).

1.2.1. Nhập dữ liệu

```
cin >> variable;
```

Trong đó `variable` là biến mà bạn muốn gán giá trị từ đầu vào. `cin` hỗ trợ nhiều kiểu dữ liệu khác nhau như `int`, `float`, `double`, `char`, và `string`.

Riêng với `string` ta có thể nhập toàn bộ dòng (bao gồm cả khoảng trắng) bằng cách sử dụng hàm `getline`. Cú pháp: `getline(cin, variable);`

Ví dụ:
```
string fullName;
getline(cin, fullName);
```

Chú ý khi có kết hợp giữa cin và getline: Vì cin không loại bỏ ký tự newline (`\n`) khỏi bộ nhớ đệm, điều này có thể dẫn đến kết quả không mong muốn. Để tránh vấn đề này, bạn có thể sử dụng thêm một dòng `cin.ignore()` để loại bỏ ký tự newline.

Ví dụ:
```
int age;
string fullName;
cin >> age;
cin.ignore(); // Loại bỏ ký tự newline khỏi bộ đệm
getline(cin, fullName);
```

1.2.2. Xuất dữ liệu

```
cout << expression;
```

Trong đó `expression` là giá trị hoặc biến bạn muốn xuất ra màn hình.

Ví dụ:
```
int age = 20;
cout << "Age: " << age << endl;
float height = 1.65;
double weight = 54;
cout << "Height: " << height << " m" << endl;
cout << "Weight: " << weight << " kg" << endl;
char grade = 'F';
cout << "Grade: " << grade << endl;
string name = "Ha Kien";
cout << "Name: " << name << endl;
```

Trong C++, ta còn có thư viện `iomanip` cung cấp các công cụ hữu ích để định dạng đầu ra khi sử dụng `cout`. Dưới đây là một số phương thức phổ biến trong `iomanip`:

- `setw`: được sử dụng để đặt độ rộng trường của đầu ra. Nếu giá trị cần xuất nhỏ hơn độ rộng đã đặt, nó sẽ được căn chỉnh và các khoảng trắng sẽ được thêm mặc định vào bên trái.

  Ví dụ:
  ```
  int num = 123;
  cout << "Number: " << setw(10) << num << endl;
  //đầu ra: Number:        123
  ```
- `setfill` được sử dụng để đặt ký tự lấp đầy cho các khoảng trống được tạo ra bởi `setw`.

  Ví dụ:
  ```
  int num = 123;
  cout << "Number: " << setfill('*') << setw(10) << num << endl;
  //đầu ra: Number: *******123
  ```

- `setprecision` được sử dụng để đặt độ chính xác của số thập phân (số chữ số sau dấu chấm thập phân).

```
Ví dụ: double pi = 3.14159265359;
      cout << "Pi: " << setprecision(5) << pi << endl;
      //Đầu ra: Pi: 3.1416
```

- `fixed` được sử dụng để đặt định dạng số thập phân cố định.

```
Ví dụ: double pi = 3.14159265359;
      cout << "Fixed: " << fixed << setprecision(5) << pi << endl;
      //Đầu ra: Fixed: 3.14159
```

1.3. Hàm trong C++

Khi xây dựng hàm, ngoài các kiểu hàm như trong C thì C++ còn cho phép xây dựng các hàm sau đây:

- Hàm đối tham chiếu
- Hàm đối mặc định
- Nạp chồng hàm (Function Overloading)
- Hàm mẫu (Template Functions)

1.3.1. Hàm đối tham chiếu

Khai báo hàm: `DataType Func_Name(DataType & Arg_Nam,..);`

Hàm đối tham chiếu là hàm mà các tham số được truyền bằng tham chiếu thay vì truyền bằng giá trị. Điều này cho phép hàm thay đổi trực tiếp giá trị của biến được truyền vào.

Ví dụ:

```
void swap(int &a, int &b) {
      int temp = a;
      a = b;
      b = temp;
}
int main() {
      int x = 10, y = 20;
      cout << "Before swap: x = " << x << ", y = " << y << endl;
      swap(x, y);
      cout << "After swap: x = " << x << ", y = " << y << endl;
      return 0;
}
//Đầu ra: Before swap: x = 10, y = 20
          After swap: x = 20, y = 10
```

1.3.2. Hàm đối mặc định

Khai báo hàm:

```
DataType Func_Name(DataType Arg_Nam1, DataType Arg_Nam2 = value2, ...);
```

Hàm đối mặc định là hàm có thể có các tham số mặc định. Điều này cho phép bạn gọi hàm mà không cần truyền tất cả các tham số, và các tham số không được truyền sẽ sử dụng giá trị mặc định đã được định nghĩa.

Ví dụ:

```
void display(int a = 10, int b = 20) {
	cout << "a = " << a << ", b = " << b << endl;
}
int main() {
	display();//Sử dụng giá trị mặc định cho cả hai tham số
	display(30);//Sử dụng giá trị mặc định cho tham số thứ hai
	display(40, 50);//Sử dụng giá trị được truyền vào cho cả hai tham số
	return 0;
}
```

1.3.3. Nạp chồng hàm (Function Overloading)

Nạp chồng hàm là khả năng định nghĩa nhiều hàm có cùng tên nhưng khác nhau về số lượng hoặc kiểu tham số. Trình biên dịch sẽ quyết định hàm nào được gọi dựa trên danh sách tham số.

Ví dụ:

```
void print(int i) {
	cout << "Integer: " << i << endl;
}
void print(double f) {
	cout << "Float: " << f << endl;
}
void print(string s) {
	cout << "String: " << s << endl;
}
int main() {
	print(10); // Gọi hàm với tham số kiểu int
	print(3.14); // Gọi hàm với tham số kiểu double
	print("Hello"); // Gọi hàm với tham số kiểu string
	return 0;
}
```

1.3.4. Hàm mẫu (Template Functions)

Hàm mẫu cho phép bạn viết một hàm duy nhất có thể làm việc với nhiều kiểu dữ liệu khác nhau. Điều này rất hữu ích cho việc viết các hàm tổng quát.

Ví dụ:
```
template <typename T>
T add(T a, T b) {
	return a + b;
}
int main() {
	cout << "Int: " << add(10, 20) << endl;
	//Hàm mẫu cho kiểu int
	cout << "Double: " << add(3.14, 2.71) << endl;
	//Hàm mẫu cho kiểu double
	cout << "String: " << add(string("Hello "), string("World")) << endl;
	// Hàm mẫu cho kiểu string
}
```

Chương II. Cơ bản về lập trình hướng đối tượng

2.1. Lập trình hướng đối tượng là gì?

Lập trình hướng đối tượng (Object-Oriented Programming - OOP) là một phương pháp lập trình dựa trên khái niệm "đối tượng", trong đó đối tượng là một thực thể bao gồm dữ liệu (thuộc tính) và các phương thức (hành vi) để thao tác với dữ liệu đó.

2.2. Khái niệm về Lớp (Class) và Đối tượng (Object)

Lớp là một bản thiết kế hoặc khuôn mẫu cho các đối tượng. Nó định nghĩa các thuộc tính (dữ liệu) và phương thức (hành vi) mà một đối tượng có thể có.

Đối tượng là một thể hiện cụ thể của một lớp. Mỗi đối tượng có các giá trị thuộc tính riêng và có thể thực thi các phương thức của lớp.

Ví dụ:
```
class Triangle {
private:
	double a, b, c; // Cạnh a, b, c
public:
	// Phương thức để tính chu vi của tam giác
	double perimeter(){
		return a + b + c;
	}
```

```
        // Phương thức để tính diện tích của tam giác bằng định lý Heron
        double area(){
            double p = perimeter() / 2;
            return sqrt(p * (p - a) * (p - b) * (p - c));
        }
        // Phương thức để hiển thị thông tin về tam giác
        void displayInfo(){
            cout << "Side a: " << a << endl;
            cout << "Side b: " << b << endl;
            cout << "Side c: " << c << endl;
            cout << "Perimeter: " << perimeter() << endl;
            cout << "Area: " << area() << endl;
        }
};
int main() {
        // Tạo một đối tượng của lớp Triangle
        Triangle myTriangle;
        // Thiết lập giá trị cho các thuộc tính của đối tượng
        myTriangle.a = 3.0; myTriangle.b = 4.0; myTriangle.c = 5.0;
        // Hiển thị thông tin về tam giác
        myTriangle.displayInfo();
}
```

- **Thuộc tính**:
  - `a, b, c`: Các cạnh của tam giác.
- **Phương thức**:
  - `double perimeter()`: Tính chu vi của tam giác.
  - `double area()`: Tính diện tích của tam giác bằng định lý Heron.
  - `void displayInfo()`: Hiển thị thông tin về các cạnh, chu vi và diện tích của tam giác.

2.3. Cài đặt phương thức

Việc cài đặt phương thức của lớp có thể được thực hiện theo hai cách: bên trong lớp và bên ngoài lớp.

Trong lớp:

```
class ClassName {
public:
        // Khai báo và cài đặt phương thức trong lớp
```

```
        ReturnType methodName(ParameterType parameter) {
        }
};
```

Ngoài lớp:

```
class ClassName {
public:
        ReturnType methodName(ParameterType parameter);
};
// Cài đặt phương thức ngoài lớp
ClassName::methodName(ParameterType parameter) {
        // Cài đặt phương thức
}
```

2.4. Truy cập đến các phần tử trong lớp

Mức độ Truy cập:

- Public (Công khai)**:**
  - Các thành phần được khai báo `public` có thể được truy cập từ bất kỳ đâu trong chương trình, bao gồm cả từ bên ngoài lớp.
  - Sử dụng khi bạn muốn cho phép các thành phần của lớp được truy cập và sử dụng một cách tự do.
- Protected (Bảo vệ)**:**
  - Các thành phần được khai báo `protected` chỉ có thể được truy cập từ các phương thức trong lớp đó và các lớp kế thừa (lớp con).
  - Sử dụng khi bạn muốn bảo vệ các thành phần khỏi việc truy cập trực tiếp từ bên ngoài lớp nhưng vẫn cho phép các lớp kế thừa có thể truy cập.
- Private (Riêng tư)**:**
  - Các thành phần được khai báo `private` chỉ có thể được truy cập từ các phương thức trong lớp đó. Không thể truy cập trực tiếp từ bên ngoài lớp hoặc từ các lớp kế thừa.
  - Sử dụng để ẩn thông tin và bảo vệ tính toàn vẹn của dữ liệu.

Cách truy cập:

Trong C++, bạn sử dụng toán tử dấu chấm (`.`) để truy cập các thành phần (thuộc tính và phương thức) của đối tượng. Khi làm việc với con trỏ đến đối tượng, bạn sử dụng toán tử mũi tên (`->`).

Ví dụ:
```
class Person {
private:
	string name; // Thuộc tính private
	int age; // Thuộc tính private
public:
	void setInfo(const string& n, int a) {
		name = n;
		age = a;
	}
	void displayInfo(){
		cout << "Name: " << name << endl;
		cout << "Age: " << age << endl;
	}
};
int main() {
	Person person;
	person.setInfo("Alice", 30);
	person.displayInfo();
	//person.name = "Bob"; // Lỗi biên dịch
	//person.age = 25; // Lỗi biên dịch
	Person* ptrPerson = new Person;
	ptrPerson->setInfo("Bob", 25);
	ptrPerson->displayInfo();
	return 0;
}
```

2.5. Cấu tử (Constructor) và Hủy tử (Destructor)

2.5.1. Cấu tử (Constructor)

Constructor (hàm khởi tạo) là một phương thức đặc biệt được gọi tự động khi một đối tượng của lớp được tạo ra. Constructor thường được sử dụng để khởi tạo các thuộc tính của đối tượng và thực hiện bất kỳ thao tác chuẩn bị cần thiết nào.

- Khởi Tạo: Constructor được gọi ngay khi đối tượng được tạo ra, đảm bảo rằng tất cả các thuộc tính của đối tượng đều được khởi tạo đúng cách.
- Nhiều Dạng: Có thể có nhiều constructor (constructor overload) với các tham số khác nhau để khởi tạo đối tượng theo nhiều cách khác nhau.
- Không Trả Về Giá Trị: Constructor không có kiểu trả về, ngay cả `void`.

Ví dụ:
```
class Rectangle {
        private:
                double width, height;
        public:
                // Constructor không tham số
                Rectangle(){
                        this.width = this.height = 0;
                }
                // Constructor với tham số
                Rectangle(double w, double h){
                        this.width = w;
                        this.height = h;
                }
        };
```

2.5.2. Hủy tử (Destructor)

Destructor (hàm hủy) là một phương thức đặc biệt được gọi tự động khi đối tượng bị hủy hoặc khi ra khỏi phạm vi của nó. Destructor thường được sử dụng để giải phóng tài nguyên mà đối tượng đã chiếm dụng, chẳng hạn như bộ nhớ động hoặc các tài nguyên hệ thống khác.

- Dọn Dẹp: Destructor được gọi khi đối tượng bị hủy, điều này giúp dọn dẹp các tài nguyên mà đối tượng chiếm dụng.
- Một Dạng: Mỗi lớp chỉ có một destructor duy nhất. Destructor không nhận tham số và không trả về giá trị.
- Tự Động Gọi: Destructor được gọi tự động khi đối tượng ra khỏi phạm vi hoặc bị hủy. Bạn không cần phải gọi destructor trực tiếp.

Ví dụ:
```
class Rectangle {
        private:
                double width, height;
        public:
                //Constructor
                Rectangle(double w, double h){
                        this.width = w;
                        this.height = h;
                }
                // Destructor
                ~Rectangle() {
                }
        };
```

2.6. Lớp mẫu (Template class)

Lớp mẫu là một lớp có thể hoạt động với các kiểu dữ liệu khác nhau được chỉ định khi lớp được sử dụng. Thay vì định nghĩa một lớp cho mỗi kiểu dữ liệu cụ thể, bạn chỉ cần định nghĩa một lớp mẫu duy nhất.

Cú pháp:

```
template <typename T>
class ClassName {
public:
	T data; // Thành viên của lớp với kiểu dữ liệu T
	// Constructor ClassName(T value){
		data = value
	}
	// Phương thức để trả về giá trị data
	T getData(){
		return data;
	}
	// Phương thức để thiết lập giá trị data
	void setData(T value) {
		data = value;
	}
};
```

Ví dụ:

```
template <typename T>
class Box {
private:
	T content;
public:
	Box(T value){
		content = value;
	}
	T getContent(){
		return content;
	}
	void setContent(T value) {
		content = value;
	}
};
```

```
int main() {
	int Box<int> intBox(123);
	cout << "Content of intBox: " << intBox.getContent() << endl;
	Box<double> doubleBox(45.67);
	cout << "Content of doubleBox: " << doubleBox.getContent() << endl;
	Box<string> stringBox("Hello, Templates!");
	cout << "Content of stringBox: " << stringBox.getContent() << endl;
	return 0;
}
```

Chương III. Phân tích thuật toán

3.1. Thuật toán là gì?

Thuật toán (algorithm) là một chuỗi các bước hoặc hướng dẫn cụ thể, rõ ràng và hữu hạn được thiết kế để giải quyết một vấn đề hoặc thực hiện một nhiệm vụ nhất định. Các thuật toán có thể được thực hiện bằng tay hoặc thông qua máy tính, và chúng được sử dụng rộng rãi trong nhiều lĩnh vực khác nhau như toán học, khoa học máy tính, kỹ thuật, và nhiều ngành khoa học khác.

Đặc điểm của thuật toán:

- Rõ ràng và xác định: Mỗi bước trong thuật toán phải được xác định một cách rõ ràng và không có sự mơ hồ.
- Tính hữu hạn: Thuật toán phải kết thúc sau một số bước hữu hạn.
- Tính đầu vào: Thuật toán có thể có các đầu vào từ tập dữ liệu đã cho.
- Tính đầu ra: Thuật toán phải có ít nhất một kết quả đầu ra.
- Tính hiệu quả: Thuật toán nên được thiết kế sao cho hiệu quả nhất có thể, tối ưu về thời gian và không gian bộ nhớ.

3.2. Các khía cạnh cần phân tích

3.2.1. Độ phức tạp không gian (Space Complexity)

Độ phức tạp không gian của một thuật toán là lượng bộ nhớ cần thiết để thực hiện thuật toán, tính theo kích thước của đầu vào. Nó đo lường tổng bộ nhớ mà thuật toán yêu cầu trong quá trình thực thi, bao gồm cả bộ nhớ cố định và bộ nhớ động.

Các thành phần của độ phức tạp không gian:

- Bộ nhớ cố định (Fixed part):
  - Bộ nhớ cần thiết cho các biến cố định, hằng số, và mã lệnh.
  - Đây là phần bộ nhớ không phụ thuộc vào kích thước của đầu vào. Ví dụ: các biến toàn cục, hằng số, hoặc mã lệnh.

    - Thường được ký hiệu là O(1).
- Bộ nhớ động (Variable part):
    - Bộ nhớ cần thiết cho các cấu trúc dữ liệu thay đổi theo kích thước của đầu vào, chẳng hạn như mảng, danh sách liên kết, hoặc cây.
    - Phần này phụ thuộc vào kích thước của đầu vào và có thể thay đổi khi kích thước đầu vào thay đổi. Ví dụ: mảng có kích thước n sẽ cần O(n) bộ nhớ.
    - Bộ nhớ động thường được ký hiệu là O(n), O(n^2), tùy thuộc vào cách cấu trúc dữ liệu được sử dụng và cách chúng thay đổi theo kích thước đầu vào.

Cách tính độ phức tạp không gian:

- Bộ nhớ cố định: Các biến đơn giản như số nguyên, số thực, hoặc ký tự thường sử dụng O(1) bộ nhớ.
- Bộ nhớ động: Một mảng có kích thước n sẽ sử dụng O(n) bộ nhớ. Một danh sách liên kết có n phần tử cũng sẽ sử dụng O(n) bộ nhớ. Một cây có n nút sẽ sử dụng O(n) bộ nhớ.

Việc tính toán độ phức tạp không gian giúp xác định lượng bộ nhớ tối thiểu mà thuật toán yêu cầu để thực thi, từ đó giúp tối ưu hóa việc sử dụng bộ nhớ và tránh tình trạng thiếu bộ nhớ khi xử lý các tập dữ liệu lớn.

3.2.2. Độ phức tạp thời gian (Time Complexity)

Độ phức tạp thời gian của một thuật toán là số bước cần thiết để thực hiện thuật toán, tính theo kích thước của đầu vào. Nó đo lường tổng số lần thực hiện các thao tác cơ bản của thuật toán khi kích thước của đầu vào thay đổi.

Ký hiệu phổ biến:

- Big O (O-notation): Mô tả thời gian chạy trong trường hợp xấu nhất.
- Omega (Ω-notation): Mô tả thời gian chạy trong trường hợp tốt nhất.
- Theta (Θ-notation): Mô tả thời gian chạy trung bình, nghĩa là thuật toán luôn chạy trong thời gian này bất kể trường hợp nào.

Các mức độ phức tạp thời gian phổ biến:

- O(1): Thời gian chạy không phụ thuộc vào kích thước của đầu vào. Các thao tác truy cập phần tử trong mảng hoặc so sánh hai số nguyên đều có độ phức tạp thời gian O(1).
- O(log n): Thời gian chạy tỷ lệ thuận với logarit của kích thước đầu vào. Ví dụ, thuật toán tìm kiếm nhị phân có độ phức tạp thời gian O(log n).

- O(n): Thời gian chạy tỷ lệ thuận với kích thước đầu vào. Ví dụ, duyệt qua một mảng có n phần tử sẽ có độ phức tạp thời gian O(n).
- O(n log n): Thời gian chạy là sự kết hợp của O(n) và O(log n). Các thuật toán sắp xếp như merge sort và quicksort có độ phức tạp thời gian O(n log n).
- $O(n^2)$: Thời gian chạy tỷ lệ thuận với bình phương của kích thước đầu vào. Các thuật toán sắp xếp đơn giản như bubble sort, insertion sort có độ phức tạp thời gian O(n^2).
- $O(2^n)$: Thời gian chạy tăng theo hàm mũ với kích thước đầu vào, rất kém hiệu quả. Ví dụ, thuật toán quay lui (backtracking) trong một số trường hợp cụ thể có thể có độ phức tạp thời gian $O(2^n)$.
- O(n!): Thời gian chạy tăng theo giai thừa của kích thước đầu vào, cực kỳ kém hiệu quả. Một số bài toán tổ hợp cụ thể có thể có độ phức tạp thời gian O(n!).

Cách tính độ phức tạp thời gian:

- Dựa trên số lượng thao tác: Đếm số lần thực hiện các thao tác cơ bản như gán giá trị, so sánh, và các phép toán số học.
- Dựa trên vòng lặp: Xác định số lần lặp của các vòng lặp, bao gồm cả vòng lặp lồng nhau.
- Dựa trên đệ quy: Phân tích lời gọi đệ quy và xác định số lần gọi đệ quy theo chiều sâu của cây đệ quy.

Việc tính toán độ phức tạp thời gian giúp xác định tốc độ thực thi của thuật toán khi kích thước đầu vào thay đổi, từ đó giúp tối ưu hóa hiệu suất và lựa chọn thuật toán phù hợp với yêu cầu về thời gian.

3.3. Các phương pháp phân tích

3.3.1. Phương pháp thực nghiệm

Phương pháp thực nghiệm liên quan đến việc triển khai và chạy thuật toán trên các tập dữ liệu thực tế hoặc giả lập để quan sát và đo lường hiệu suất của thuật toán.

Quy trình thực hiện:

- Triển khai thuật toán: Viết mã nguồn của thuật toán trong ngôn ngữ lập trình mong muốn.
- Chuẩn bị tập dữ liệu: Lựa chọn hoặc tạo ra các tập dữ liệu đầu vào phù hợp, bao gồm các trường hợp tốt nhất, xấu nhất và trung bình.
- Chạy thuật toán: Thực hiện thuật toán với các tập dữ liệu đã chuẩn bị.
- Đo lường hiệu suất: Sử dụng các công cụ và phương pháp đo lường để ghi lại thời gian chạy, sử dụng bộ nhớ và các chỉ số khác.

- Phân tích kết quả: So sánh và phân tích kết quả thu được để đánh giá hiệu suất của thuật toán.

Ưu điểm:

- Chính xác: Cung cấp số liệu chính xác về hiệu suất của thuật toán trong thực tế.
- Trực quan: Dễ dàng quan sát và hiểu được cách thuật toán hoạt động với các tập dữ liệu cụ thể.
- Phát hiện lỗi: Giúp phát hiện và sửa lỗi trong quá trình triển khai thuật toán.

Nhược điểm:

- Tốn thời gian: Cần nhiều thời gian để triển khai, chuẩn bị dữ liệu và chạy thử.
- Phụ thuộc vào phần cứng: Hiệu suất có thể bị ảnh hưởng bởi cấu hình phần cứng và môi trường chạy thử.
- Không tổng quát: Kết quả có thể không phản ánh chính xác hiệu suất của thuật toán trên mọi tập dữ liệu và điều kiện khác nhau.

3.3.2. Phương pháp phân tích lý thuyết

Phân tích lý thuyết là phương pháp sử dụng các công cụ toán học để đánh giá hiệu suất của thuật toán mà không cần triển khai thực tế.

Quy trình thực hiện:

- Xác định các thao tác cơ bản: Chọn ra các thao tác cơ bản của thuật toán để làm cơ sở đánh giá.
- Xác định các trường hợp: Phân tích thuật toán trong các trường hợp tốt nhất, xấu nhất và trung bình.
- Tính toán độ phức tạp: Sử dụng các ký hiệu Big O, Ω, và Θ để mô tả độ phức tạp thời gian và không gian của thuật toán.
- So sánh và đánh giá: So sánh độ phức tạp của thuật toán với các thuật toán khác để đưa ra kết luận về hiệu suất tương đối.

Ưu điểm:

- Tổng quát: Cung cấp đánh giá tổng quát về hiệu suất của thuật toán mà không phụ thuộc vào môi trường cụ thể.
- Tiết kiệm thời gian: Không cần triển khai thực tế và chạy thử, giúp tiết kiệm thời gian và công sức.
- Phân tích mọi trường hợp: Cho phép đánh giá hiệu suất trong mọi trường hợp đầu vào, bao gồm trường hợp tốt nhất, xấu nhất và trung bình.

Nhược điểm:

- Trừu tượng: Có thể khó hiểu và không trực quan đối với những người không quen với toán học và lý thuyết tính toán.
- Không chính xác tuyệt đối: Mặc dù cung cấp một cái nhìn tổng quát, nhưng phân tích lý thuyết không thể thay thế hoàn toàn việc kiểm tra thực tế.
- Phức tạp: Yêu cầu kiến thức toán học sâu rộng và kỹ năng phân tích để thực hiện chính xác.

Chương IV. Các cấu trúc dữ liệu cơ bản

4.1. Cấu trúc Vector

4.1.1. Cấu trúc tuyến tính, phi tuyến

Cấu trúc dữ liệu tuyến tính là một cấu trúc mà các phần tử dữ liệu được liên kết với nhau tuần tự. Một phần tử chỉ kết nối với một phần tử trước và một tử sau.

Bằng cách tổ chức như vậy, ta có thể duyệt qua các phần tử của nó trong một lần chạy.

Ví dụ: Danh sách (lists)

Vector, chuỗi (vectors sequences)

Danh sách kiểu ngăn xếp, danh sách kiểu hàng đợi (stack, queue)

Cấu trúc dữ liệu phi tuyến là một cấu trúc mà các phần tử dữ liệu được liên kết với nhau thứ tự tuyến tính. Một phần tử có thể kết nối với nhiều phần tử trước, sau.

Ví dụ: Cây (tree)

Đồ thị (graph)

Cấu trúc dữ liệu tuyến tính và phi tuyến đều có những đặc điểm và ứng dụng riêng. Việc lựa chọn cấu trúc dữ liệu phù hợp phụ thuộc vào bài toán cụ thể và yêu cầu về hiệu suất cũng như cách tổ chức dữ liệu. Hiểu rõ về từng loại cấu trúc dữ liệu và cách chúng hoạt động sẽ giúp lập trình viên tối ưu hóa chương trình và giải quyết các vấn đề phức tạp một cách hiệu quả.

4.1.2. Kiểu dữ liệu Vector

Vector là một loại cấu trúc dữ liệu mạnh mẽ và linh hoạt, được cung cấp bởi thư viện chuẩn (`<vector>`). Nó thuộc nhóm cấu trúc dữ liệu tuyến tính và là một trong những lớp dữ liệu quan trọng nhất trong lập trình C++ nhờ tính năng động và hiệu quả của nó.

Đặc điểm chính của Vector:

- Kích Thước Động: Vector có thể thay đổi kích thước trong thời gian thực. Bạn có thể thêm hoặc xóa phần tử mà không cần phải xác định kích thước trước.
- Truy Cập Ngẫu Nhiên (Random Access): Bạn có thể truy cập bất kỳ phần tử nào của vector trực tiếp thông qua chỉ số, tương tự như trong mảng.
- Quản Lý Bộ Nhớ: Vector tự động quản lý bộ nhớ. Khi kích thước của vector thay đổi, bộ nhớ được cấp phát hoặc giải phóng tự động.
- Hiệu Suất Tốt: Thao tác thêm phần tử vào cuối vector (sử dụng `push_back`) là rất nhanh chóng. Tuy nhiên, việc chèn hoặc xóa phần tử ở giữa vector có thể tốn thời gian vì phải dịch chuyển các phần tử khác.

4.1.3. Cấu trúc Vector trong C++

Khai báo

- `vector<T> vec_name(n);`
- `vector<T> vec_name(n, value);`
- `T arr = {a1, a2, .., an};`
- `vector<T> vec_name(arr, arr + n);`
- `vector<T> vec_name(arr, arr + sizeof(arr)/sizeof(T));`

Các nhóm thao tác của Vector:

- Modifiers:
  - `assign(int size, int value);`
  - `pop_back();`
  - `insert(position, value);`
  - `erase(position);`
  - `erase(start-position, end-position);`
  - `emplace(position, element);`
  - `emplace_back();`
  - `emplace_back(value);`
  - `swap(vector);`
  - `clear();`
- Iterators:
  - `vector<int>::iterator vec_it;`
    - `begin()`
    - `end()`
  - `vector<int>::reverse_iterator vec_it;`
    - `rbegin()`
    - `rend()`
- Capacity:
  - `size();`
  - `max_size();`

- `capacity();`
- `resize(n);`
- `resize(int n, int value);`
- `empty();`
- `shrink_to_fit();`
- `reserve(n);`

- Element access
  - `at(position);`
  - `data()`
  - `front()`
  - `back()`
  - `[]`

4.1.4. Các ứng dụng của Vector

Ứng dụng trực tiếp:

Ứng dụng trực tiếp của `vector` đề cập đến việc sử dụng cấu trúc dữ liệu này để giải quyết các bài toán cụ thể hoặc thực hiện các chức năng nhất định. Những ứng dụng này làm nổi bật những lợi ích cơ bản của `vector`, chẳng hạn như khả năng mở rộng linh hoạt và thao tác dễ dàng.

`vector` có thể được sử dụng để lưu trữ và quản lý một tập hợp các đối tượng của cùng loại. Đây là một trong những ứng dụng cơ bản và phổ biến của `vector`.

Ứng dụng gián tiếp:

Ứng dụng gián tiếp của `vector` liên quan đến việc sử dụng `vector` như một phần của các cấu trúc dữ liệu hoặc thuật toán phức tạp hơn. Trong những trường hợp này, `vector` cung cấp một nền tảng vững chắc và hiệu quả cho việc triển khai các giải pháp phức tạp hơn.

`vector` có thể được sử dụng như một cấu trúc dữ liệu bổ trợ trong các thuật toán phức tạp. Ví dụ, bạn có thể sử dụng `vector` để lưu trữ các giá trị trung gian hoặc kết quả trong các thuật toán sắp xếp, tìm kiếm, hoặc phân tách.

`vector` có thể đóng vai trò là thành phần của các cấu trúc dữ liệu phức tạp hơn như ngăn xếp (stack), hàng đợi (queue), hoặc đồ thị (graph). Nhờ vào khả năng mở rộng và quản lý bộ nhớ tự động, `vector` thường được sử dụng để xây dựng các cấu trúc dữ liệu này.

`vector` cũng có thể được sử dụng trong các ứng dụng phức tạp như xử lý đồ họa, mô phỏng, hoặc phân tích dữ liệu.

4.1.5. Cài đặt Vector bằng mảng

Khởi tạo mảng:

- Một con trỏ để trỏ đến mảng động `v`. Mảng này sẽ được sử dụng để lưu trữ các phần tử của `vector`.
- Một biến để lưu kích thước hiện tại của mảng (`size`), và một biến để lưu sức chứa tối đa của mảng (`capacity`).

Cung cấp các phương thức cơ bản:

- Thêm phần tử (`push_back`): Kiểm tra xem mảng hiện tại có đủ sức chứa để thêm phần tử mới không, nếu không đủ, tăng kích thước của mảng. Sau đó thêm phần tử vào mảng và cập nhật kích thước.
- Truy cập phần tử (`operator[]`): Cung cấp phương thức để truy cập phần tử tại chỉ số nhất định, với kiểm tra lỗi nếu chỉ số nằm ngoài phạm vi của mảng.
- Truy cập phần tử đầu tiên và cuối cùng (`front, back`): Phương thức để truy cập phần tử đầu tiên và cuối cùng của vector. Đảm bảo kiểm tra nếu vector rỗng.
- Lấy số phần tử (`size`): Cung cấp phương thức để trả về số lượng phần tử hiện tại trong vector.
- Kiểm tra nếu vector rỗng (`empty`): Cung cấp phương thức để kiểm tra xem vector có chứa bất kỳ phần tử nào không.
- Xóa tất cả các phần tử (`clear`): Cung cấp phương thức để xóa tất cả các phần tử trong vector, thường bằng cách đặt kích thước về 0.

Code mẫu cài đặt Vector bằng mảng trong C++:

```
#include<iostream>
using namespace std;
#ifndef __vector__cpp__
#define __vector__cpp__

template <class T>
class vector_reverse_iterator{
	T *curr;
public:
	vector_reverse_iterator(T *c=0) {curr=c;}
	vector_reverse_iterator<T> &operator=(vector_reverse_iterator<T> it) {
		this->curr=it.curr;
		return *this;
	}
```

```
	vector_reverse_iterator<T> operator++()//++curr{
		curr--;
		return curr;
	}
	vector_reverse_iterator<T> operator++(int){
		vector_reverse_iterator<T> tmp=curr;
		curr--;
		return tmp;
	}
	T &operator*() {return *curr;}
	bool operator!=(vector_reverse_iterator<T> t) {return curr!=t.curr;}
};

template<class T>
class Vector{
private:
	int cap,num;
	T *buff;
public:
	Vector() {cap=num=0;buff=0;}
	Vector(int k,T x) {cap=num=k; buff=new T[k]; for(int i=0;i<k;i++) buff[i]=x;}
	~Vector() {if(buff) delete[]buff;}
	int capacity() {return cap;}
	int size() {return num;}
	bool empty() {return num==0;}
	void pop_back() {if(num>0) num--;}
	void extend(int newcap){
		if(newcap<cap) return;
		cap=newcap;
		T *temp=new T[cap];
		for(int i=0;i<num;i++) temp[i]=buff[i];
		if(buff) delete []buff;
		buff= temp;
	}
	T &back() {return buff[num-1];}
	void push_back(T x){
		if(num==cap) extend(cap*2+5);
		buff[num++]=x;
	}
	T &operator[](int k) {return buff[k];}
```

```
        void insert(int k,T x){
                if(cap==num) extend(cap*2+5);
                for(int i=num-1;i>=k;i--) buff[i+1]=buff[i];
                buff[k]=x;
                num++;
        }
        Vector &operator=(Vector<T> V){
                this->num=V.num;
                this->cap=V.cap;
                if(cap){
                        this->buff=new T[cap];
                        for(int i=0;i<num;i++) this->buff[i]=V.buff[i];
                }
                else this->buff=0;
                return *this;
        }
        typedef T *iterator;
        iterator begin() {return buff;}
        iterator end() {return buff+num;}
        typedef vector_reverse_iterator<T> reverse_iterator;
        reverse_iterator rbegin() {return reverse_iterator(buff+num-1);}
        reverse_iterator rend() {return reverse_iterator(buff-1);}
};
#endif
```

4.1.6. Phát triển mảng:

Khi thực hiện phép toán. Nếu mảng đầy sẽ dẫn đến xảy ra lỗi. Để có thể thêm phần tử đó vào ta phải mở rộng mảng.

Có hai chiến lược để có thể mở rộng mảng:

- Chiến lược phát triển theo hằng số: Tăng thêm kích thước mảng theo một hằng số c
- Chiến lược gấp đôi: Tăng gấp đôi số phần tử hiện có của mảng

Chiến lược phát triển theo hằng số:

Khi mảng cần mở rộng, kích thước của mảng mới được tăng thêm một số lượng cố định (c) phần tử.

Ưu điểm:

- Dễ dàng cài đặt: Cài đặt chiến lược này thường đơn giản hơn vì chỉ cần thêm một số phần tử cố định.

- Dự đoán kích thước: Có thể dễ dàng dự đoán kích thước của mảng mới sau mỗi lần mở rộng.

Nhược điểm:

- Hiệu suất kém: Thao tác mở rộng mảng có thể xảy ra thường xuyên hơn, dẫn đến nhiều thao tác sao chép hơn. Ví dụ, nếu bạn mở rộng mảng mỗi lần thêm 10 phần tử, và bạn liên tục thêm nhiều phần tử, bạn sẽ phải sao chép mảng nhiều lần.
- Tài nguyên bộ nhớ: Khi mảng mở rộng, bạn có thể cấp phát nhiều bộ nhớ hơn cần thiết, dẫn đến lãng phí bộ nhớ.

Chiến lược gấp đổi:

Khi mảng cần mở rộng, kích thước của mảng mới được gấp đôi so với kích thước hiện tại.

Ưu điểm:

- Hiệu suất tốt hơn: Chiến lược này giảm số lần mở rộng mảng cần thiết. Vì mảng được gấp đôi kích thước mỗi lần mở rộng, số lần mở rộng sẽ ít hơn nhiều so với chiến lược phát triển theo hằng số.
- Tối ưu hóa tài nguyên: Mảng sẽ có kích thước lớn hơn cần thiết ít hơn, dẫn đến ít lãng phí bộ nhớ hơn so với việc mở rộng theo hằng số.

Nhược điểm:

- Cài đặt phức tạp hơn: Cần phải tính toán kích thước mới của mảng mỗi lần mở rộng, điều này có thể phức tạp hơn một chút so với việc mở rộng theo hằng số.
- Sự lãng phí đôi lúc tạo mảng: Mặc dù có thể tiết kiệm bộ nhớ tổng thể, nhưng mỗi lần mở rộng có thể dẫn đến việc cấp phát một lượng bộ nhớ lớn hơn cần thiết, gây lãng phí bộ nhớ tạm thời.

Yếu tố	Chiến lược phát triển theo hằng số	Chiến lược gấp đôi
Tần số mở rộng	Mở rộng thường xuyên	Mở rộng ít hơn
Hiệu suất	Thấp do sao chép nhiều lần	Cao hơn do ít lần sao chép hơn
Lãng phí bộ nhớ	Cấp phát bộ nhớ lớn hơn cần thiết	Cấp phát gần đúng kích thước
Dễ cài đặt	Đơn giản hơn	Phức tạp hơn
Dự đoán kích thước	Dễ dàng dự đoán	Kích thước mới khó dự đoán

4.1.7. Iterator (Bộ lặp)

Iterator, hay còn gọi là bộ lặp, là một đối tượng trong lập trình cung cấp cách để duyệt qua các phần tử trong một cấu trúc dữ liệu, chẳng hạn như mảng, danh sách, hay vector, mà không cần biết chi tiết về cách dữ liệu được tổ chức bên trong.

Iterator cho phép người dùng truy cập và duyệt qua các phần tử của một cấu trúc dữ liệu theo cách tuần tự và nhất quán. Nó tách biệt việc duyệt qua các phần tử khỏi cấu trúc dữ liệu cụ thể, do đó hỗ trợ việc thay đổi cấu trúc dữ liệu mà không làm thay đổi mã duyệt qua.

Iterator thường có các thành phần sau:

- `begin()`: Trả về một iterator trỏ đến phần tử đầu tiên của cấu trúc dữ liệu.
- `end()`: Trả về một iterator trỏ đến vị trí ngay sau phần tử cuối cùng của cấu trúc dữ liệu (thường được sử dụng để xác định điểm dừng của vòng lặp).
- `operator*`: Truy cập giá trị của phần tử mà iterator đang trỏ đến.
- `operator++`: Di chuyển iterator đến phần tử tiếp theo trong cấu trúc dữ liệu.
- `operator!=`: So sánh hai iterator để kiểm tra xem chúng có trỏ đến cùng một vị trí hay không.

Ví dụ:
```
vector<int> V(7, 6);
vector<int>::iterator it;
for(it = V.begin(); it != V.end(); ++it){
	cout << *it << "\t";
}
```

Iterator được sử dụng để có thể chèn hoặc loại bỏ phần tử khỏi cấu trúc dữ liệu:

- Chèn thêm phần tử:
  - iterator insert (const_iterator position, const value_type& val)
  - Chúng ta cần tạo một ô mới có chỉ số r bằng cách đẩy `n - r` phần tử từ `V[r], ..., V[n - 1]` về sau 1 vị trí
  - Trong trường hợp xấu nhất (r = 0), phép toán thực hiện trong thời gian O(n)
- Loại bỏ phần tử:
  - Phép toán iterator erase (const_iterator position) chúng ta cần đẩy `n - r - 1` phần tử từ `V[r + 1], ..., V[n - 1]` về trước một vị trí
  - Trong trường hợp xấu nhất (r = 0), phép toán thực hiện trong thời gian O(n)

4.2. Danh sách liên kết (Linked List)

4.2.1. Mô hình cấu trúc danh sách liên kết

Danh sách liên kết (Linked List) là một cấu trúc dữ liệu động bao gồm một chuỗi các phần tử, trong đó mỗi phần tử chứa dữ liệu và một hoặc nhiều tham chiếu (liên kết) tới các phần tử khác. Điều này cho phép danh sách có thể mở rộng hoặc thu gọn một cách linh hoạt mà không cần cấp phát bộ nhớ cố định từ đầu.

Các loại danh sách liên kết:

- Danh sách liên kết đơn: Mỗi nút (node) trong danh sách chứa một phần tử dữ liệu và một con trỏ đến nút tiếp theo trong danh sách.
- Danh sách liên kết kép: Mỗi nút chứa một phần tử dữ liệu, một con trỏ tới nút tiếp theo và một con trỏ tới nút trước đó.

4.2.2. Cấu trúc danh sách trong thư viện STL C++

Trong thư viện Standard Template Library (STL) của C++, danh sách liên kết được cung cấp thông qua lớp `std::list`. Lớp này cung cấp các tính năng và thao tác cần thiết để làm việc với danh sách liên kết mà không cần phải tự cài đặt từ đầu.

Mô tả:

- `std::list` là một danh sách liên kết kép, trong đó mỗi phần tử chứa một con trỏ đến phần tử trước đó và một con trỏ đến phần tử kế tiếp.
- Cho phép truy cập và sửa đổi phần tử ở cả hai đầu của danh sách một cách hiệu quả.
- Hỗ trợ việc chèn (`insert`) và xóa (`erase`) các phần tử ở bất kỳ vị trí nào trong danh sách với độ phức tạp O(1), giả sử đã có con trỏ tới vị trí đó.

Tính năng chỉnh của danh sách liên kết:

- Xem phần tử
  - `front`
  - `back`
- Thêm phần tử
  - `push_back`
  - `push_front`
  - `insert`
- Bớt phần tử
  - `pop_back`
  - `pop_front`
  - `erase`
- Bộ lặp xuôi
  - `begin`
  - `end`
- Bộ lặp ngược
  - `rbegin`
  - `rend`
- Thao tác khác
  - `size`
  - `empty`

4.2.3. Danh sách liên kết đơn

Các nút (node) được cài đặt bao gồm:

- Phần tử lưu trữ trong nó
- Một liên kết đến nút kế tiếp

Để cài đặt một danh sách liên kết đơn, ta cần sử dụng môt con trỏ header trỏ vào node đầu danh sách và con trỏ trailer trỏ vào node cuối danh sách.

Cấu trúc của một nút (node) trong danh sách liên kết đơn:

- Các thuộc tính
  - `Element elem;`
  - `Node *next;`
- Các phương thức
  - `Node *getnext() //Trả lại địa chỉ của nút kế tiếp`
  - `Element getElem() //Trả lại giá trị phần tử lưu trữ trong nút`
  - `void setNext(Node *) //Gán địa chỉ cho thuộc tính next`
  - `void setElem(Element e) //Gán giá trị e cho thuộc tính elem`

Cấu trúc danh sách liên kết đơn:

- Các thuộc tính:
  - `Node *header`
  - `Node *trailer`
- Các phương thức chung:
  - `int size()`
  - `bool empty()`
- Các phương thức truy cập:
  - `front()`
  - `back()`
- Chèn và xóa:
  - `insert`
  - `erase`
- Các phương thức cập nhật:
  - `void push_front(T e)`
  - `void push_back(T e)`
  - `void pop_front()`
  - `void pop_back()`
- Bộ lặp xuôi:
  - `begin`
  - `end`
  - `=`
  - `!=`
  - `++`
  - `*`

4.2.4. Danh sách liên kết kép

Các nút (node) được cài đặt bao gồm:

- Phần tử lưu trữ trong nó
- Một liên kết đến nút trước nó
- Một liên kết đến nút kế tiếp

Để cài đặt một danh sách liên kết kép, ta cũng cần sử dụng môt con trỏ header trỏ vào node đầu danh sách và con trỏ trailer trỏ vào node cuối danh sách.

Cấu trúc của một nút (node) trong danh sách liên kết kép:

- Các thuộc tính
  - `Element elem;`
  - `Node *next, *pre;`
- Các phương thức
  - `Node *getnext() //Trả lại địa chỉ của nút kế tiếp`
  - `Node *getPre() //Trả lại địa chỉ của nút trước đó`
  - `Element getElem() //Trả lại địa chỉ của phần tử lưu trong nút`
  - `void setNext(Node *) //Đặt thuộc tính Next trỏ đến địa chỉ của phần tử là đối của phương thức`
  - `void setPre(Node *) //Đặt thuộc tính Prior trỏ đến địa chỉ của phần tử là đối của phương thức`
  - `void setElem(Element e) //Đặt phần tử e vào nút`

Cấu trúc danh sách liên kết kép:

- Các thuộc tính:
  - `Node *header`
  - `Node *trailer`
- Các phương thức chung:
  - `int size()`
  - `bool empty()`
- Các phương thức truy cập:
  - `front()`
  - `back()`
- Chèn và xóa:
  - `insert`
  - `erase`
- Các phương thức cập nhật:
  - `void push_front(T e)`
  - `void push_back(T e)`
  - `void pop_front()`
  - `void pop_back()`
- Bộ lặp xuôi:
  - `begin`
  - `end`
  - `=`
  - `!=`
  - `++`
  - `*`
- Bộ lặp ngược:
  - `rbegin`
  - `rend`
  - `=`
  - `!=`
  - `++`
  - `*`

Thuật toán chèn (insert):

```
insert (p,e): //Bổ sung phần tử e vào phần tử nút p

Tạo ra một nút mới q

q->setElement(e) //Đặt gia trị e vào nút q

q->setNext(p->getNext())//liên kết với phần tử sau nó

p->getNext()->setPrev(q)//Liên kết phần tử sau p với q

q->setPrev(p) //liên kết q với phần tử trước nó

p->setNext(q) //liên kết p với q

return q //trả lại vị trí của q
```

Thuật toán xóa (erase):

```
erase(p): //Xóa nút p
//kết nối phần tử trước p với phần tử sau p
p->getPre()->setNext(p->getNext())
//kết nối phần tử sau p với pần tử trước p
p->getNext()->setPre(p->getPre())
//bỏ kết nối p với phần tử trước và sau nó
p->setPre(NULL)
p->setNext(NULL)
delete p
```

4.2.5. So sánh mảng và danh sách liên kết:

Những đặc trưng của mảng và danh sách liên kết:

Mảng	Danh sách liên kết
• Bộ nhớ sử dụng lưu trữ phụ thuộc vào việc cài đặt chứ không phải số lượng thực sự cần lưu. • Mối quan hệ giữa phần tử đầu và các phần tử khác là rất ít • Các phần tử được sắp xếp cho phép tìm kiếm rất nhanh • Việc chèn và xóa phần tử đòi hỏi phải di chuyển các phần tử.	• Bộ nhớ sử dụng để lưu trữ tương ứng với số lượng các phần tử thực sự cần lưu tai bất kỳ thời điểm nào. • Sử dụng một con trỏ để lưu phần tử đầu, từ đó đi đến các phần tử khác. • Việc bổ sung và xóa bỏ các phần tử không phải di chuyển các phần tử • Truy nhập đến các phần tử chỉ có thể thực hiện được bằng cách đi dọc theo chuỗi mắt xích từ phần tử đầu. Vì vậy đối với danh sách liên kết đơn thì thời gian tìm kiếm một phần tử sẽ là O(n).

4.3. Cấu trúc dữ liệu ngăn xếp (Stack)

4.3.1. Định nghĩa ngăn xếp

Ngăn xếp (stack) là cách tổ chức lưu trữ các đối tượng dưới dạng một danh sách tuyến tính mà việc bổ sung đối tượng và lấy các đối tượng ra được thực hiện ở cùng một đầu của danh sách.

Ngăn xếp (stack) được gọi là danh sách kiểu LIFO (Last In, First Out), tức là phần tử được đưa vào cuối cùng sẽ được lấy ra đầu tiên.

4.3.2. Cấu trúc dữ liệu trừu tượng Stack

Cấu trúc dữ liệu Stack (Ngăn xếp) là một cấu trúc dữ liệu trừu tượng được sử dụng để lưu trữ và truy cập dữ liệu theo nguyên tắc LIFO (Last In, First Out), tức là phần tử được đưa vào cuối cùng sẽ được lấy ra đầu tiên.

Các thao tác cơ bản trong Stack:

- `push(Object o)`: bổ sung đối tượng o vào cuối Stack.
- `pop()`: xóa phần tử cuối cùng của Stack.
- `top()`: trả lại tham chiếu đến phần tử được bổ sung vào cuối cùng của Stack.
- `size()`: trả lại số phần tử hiện lưu trữ trong Stack.
- `empty()`: trả lại giá trị kiểu boolean để xác định Stack có lưu trữ phần tử nào hay không.

Chú ý: Với Stack rỗng thì phép toán `pop()` và `top()` không thể thực hiện được.

Một số ứng dụng của Stack:

- Ứng dụng trực tiếp của Stack:

Quay lại và hoàn tác: Trong các ứng dụng như trình soạn thảo văn bản và trình duyệt web, stack được sử dụng để lưu trữ các hành động đã thực hiện, cho phép người dùng quay lại trạng thái trước đó (Undo) hoặc thực hiện lại các hành động (Redo).

Duyệt cây và đồ thị: Các thuật toán duyệt cây như duyệt theo chiều sâu (Depth-First Search) và duyệt đồ thị sử dụng stack để lưu trữ các đỉnh hoặc các nút đang được duyệt.

Phân tích biểu thức: Trong việc phân tích biểu thức toán học hoặc ngôn ngữ lập trình, stack được sử dụng để xử lý các biểu thức hậu tố (postfix) hoặc tiền tố (prefix), cũng như để kiểm tra các dấu ngoặc mở và đóng.

Quản lý ngăn xếp hàm: Trong các ngôn ngữ lập trình, stack được sử dụng để quản lý các khung ngăn xếp (stack frames) trong quá trình gọi và trả về từ các hàm hoặc phương thức, giúp lưu trữ thông tin về các biến cục bộ và địa chỉ quay lại.

- Ứng dụng gián tiếp của Stack:

Tính toán đệ quy: Trong các thuật toán đệ quy, stack được sử dụng để lưu trữ trạng thái và thông tin về các cuộc gọi đệ quy. Mặc dù điều này thường được ẩn sau lớp trừu tượng của ngôn ngữ lập trình, nó vẫn là một phần quan trọng của cơ chế thực thi đệ quy.

Lưu trữ tạm thời: Trong một số hệ thống và thuật toán, stack có thể được sử dụng như một khu vực lưu trữ tạm thời để giữ dữ liệu hoặc trạng thái trong quá trình thực thi mà không yêu cầu cấu trúc lưu trữ lâu dài.

Tối ưu hóa bộ nhớ: Stack có thể giúp tối ưu hóa việc sử dụng bộ nhớ trong một số ứng dụng bằng cách cung cấp cách tổ chức dữ liệu hiệu quả cho các thao tác cần thực hiện nhanh chóng và đơn giản.

Hỗ trợ các cấu trúc dữ liệu khác: Một số cấu trúc dữ liệu phức tạp hơn, như danh sách liên kết hoặc bảng băm, có thể sử dụng stack như một phần của cơ chế hoạt động của chúng, giúp giải quyết các vấn đề phức tạp một cách hiệu quả hơn.

4.3.3. Cài đặt Stack bằng mảng

Code mẫu cài đặt Stack bằng mảng:

```
#include<bits/stdc++.h>
using namespace std;
#ifndef stack__cpp
#define stack__cpp
template <class T>
class Stack{
	int num, cap; //Lưu trữ số lượng phần tử hiện tại và sức chứa trong stack;
	T *elem; //Con trỏ đến mảng động chứa các phần tử của stack.
public:
	Stack() {num = cap = 0; elem = NULL;} //Constructor
	int size() {return num;}
	bool empty() {return num == 0;}
	T &top() {return elem[num-1];}
	void pop() {num--;}
	void push(T x){
		if(cap == num){
			cap = cap ? cap * 2 : 1;
			T *tem = new T[cap];
			for(int i = 0; i < num; i++) tem[i] = elem[i];
			if(elem) delete []elem;
			elem = tem;
		}
		elem[num++] = x;
	}
	void clear() {
		delete[] elem;
		elem = NULL; num = cap = 0;
	}
};
#endif
```

Trong đoạn code trên, chiến lược mở rộng gấp đôi đã được sử dụng để tăng kích thước của mảng trong Stack. Điều này giúp duy trì hiệu suất tối ưu cho các thao tác thêm phần tử vào stack bằng cách giảm số lần phải cấp phát lại bộ nhớ và sao chép dữ liệu. Khi mảng đầy, kích thước của nó được tăng gấp đôi, giúp giảm thiểu số lần cần phải mở rộng mảng trong tương lai gần. Điều này không chỉ cải thiện tốc độ hoạt động của stack mà còn giúp quản lý bộ nhớ hiệu quả hơn, tránh các thao tác cấp phát và giải phóng bộ nhớ tốn kém.

Bên cạnh đó, ta cũng có thể cài đặt Stack bằng danh sách liên kết đơn, giúp loại bỏ vấn đề liên quan đến việc phải mở rộng kích thước mảng khi stack trở nên đầy. Sử dụng danh sách liên kết đơn cho phép chúng ta thêm và loại bỏ các phần tử một cách linh hoạt mà không cần phải thay đổi kích thước cấu trúc lưu trữ. Mỗi phần tử được lưu trữ trong một nút của danh sách liên kết, với con trỏ đến nút tiếp theo, giúp việc thao tác với đỉnh stack trở nên đơn giản và hiệu quả. Phương pháp này cung cấp sự linh hoạt trong việc quản lý bộ nhớ và tránh các vấn đề liên quan đến giới hạn kích thước mảng. Nguồn tham khảo: https://ideone.com/s6RpaH

4.3.4. Ký pháp Ba Lan – Ứng dụng Stack tính giá trị biểu thức

Trong toán học, các biểu thức thường được viết theo ký pháp trung tố, nơi các toán tử nằm giữa các toán hạng. Ví dụ, biểu thức `a*(b+c)-(d*a)` được viết với các toán tử `*` và `-` đặt giữa các toán hạng như `a`, `b`, `c`, và `d`. Tuy nhiên, nhà toán học người Ba Lan Jan Łukasiewicz đã đề xuất hai dạng ký pháp khác để biểu diễn các biểu thức toán học: ký pháp tiền tố (prefix) và ký pháp hậu tố (postfix).

Ký pháp tiền tố (Prefix Notation) là cách viết mà các toán tử được đặt trước các toán hạng. Ví dụ, biểu thức `a*(b+c)-(d*a)` trong ký pháp tiền tố sẽ được viết là `-*a+bc*da`. Ở đây, toán tử `-` đứng trước, tiếp theo là `*a+bc` và `*da`, thể hiện rõ thứ tự các phép toán mà không cần dấu ngoặc.

Ký pháp hậu tố (Postfix Notation), ngược lại, đặt các toán tử sau các toán hạng. Ví dụ, cùng biểu thức `a*(b+c)-(d*a)` trong ký pháp hậu tố sẽ được viết là `abc+*da*-`. Ở dạng này, các phép toán được thực hiện ngay sau khi các toán hạng của chúng được cung cấp, cũng loại bỏ nhu cầu về dấu ngoặc đơn.

Cả hai dạng ký pháp tiền tố và hậu tố đều được gọi chung là ký pháp Ba Lan (Polish Notation). Các biểu thức viết theo ký pháp Ba Lan có một số ưu điểm quan trọng. Thứ nhất, chúng không sử dụng các dấu ngoặc đơn, giúp tránh được sự phức tạp và nhầm lẫn khi xác định thứ tự ưu tiên các phép toán. Thứ hai, chúng dễ dàng hơn trong việc lập trình và tính toán giá trị của các biểu thức, do không cần phải xử lý các ưu tiên toán tử và ngoặc đơn. Điều này làm cho chúng đặc biệt hữu ích trong các hệ thống máy tính và ngôn ngữ lập trình.

Dưới đây là thuật toán chuyển đổi biểu thức dạng trung tố về dạng ký pháp hậu tố:

- Khởi tạo: Tạo hai stack: `Opr` để lưu các toán tử và `BLExp` để lưu các phần của biểu thức hậu tố.
- Đọc biểu thức: Đọc lần lượt từng ký tự từ trái qua phải của biểu thức trung tố.
- Xử lý các ký tự:
  - Nếu gặp dấu `(` : PUSH dấu `(` vào stack `Opr`.
  - Nếu gặp toán hạng (chữ cái hoặc số): PUSH toán hạng đó vào stack `BLExp`.
  - Nếu gặp dấu `)`:
    - POP các toán tử từ `Opr` và PUSH vào `BLExp` cho đến khi gặp dấu `(`.
    - POP dấu `(` ra khỏi `Opr` và bỏ qua nó.
  - Nếu gặp toán tử (như `+`, `-`, `*`, `/`):
    - Nếu `Opr` rỗng: PUSH toán tử vào `Opr`.
    - Nếu không rỗng:
      - Nếu toán tử ở đỉnh `Opr` có mức ưu tiên cao hơn hoặc bằng toán tử hiện tại, POP các toán tử từ `Opr` và PUSH vào `BLExp` cho đến khi gặp toán tử có mức ưu tiên thấp hơn hoặc `Opr` rỗng. Sau đó, PUSH toán tử hiện tại vào `Opr`.
      - Nếu toán tử ở đỉnh `Opr` có mức ưu tiên thấp hơn, chỉ PUSH toán tử hiện tại vào `Opr`.
- Hoàn tất: Khi đã đọc hết biểu thức, POP tất cả các toán tử còn lại trong `Opr` và PUSH vào `BLExp`.
- Kết quả: `BLExp` hiện tại chứa biểu thức dạng hậu tố.

Chú ý: Thứ tự ưu tiên giảm dần: (,), /, *, -, +.

Ví dụ: Biểu thức trung tố: `a*(b+c)-(d*a)`

- Bước 1: Đọc `a`, PUSH vào `BLExp`.
- Bước 2: Gặp `*`, PUSH vào `Opr`.
- Bước 3: Gặp `(`, PUSH vào `Opr`.
- Bước 4: Gặp `b`, PUSH vào `BLExp`.
- Bước 5: Gặp `+`, PUSH vào `Opr`.
- Bước 6: Gặp `c`, PUSH vào `BLExp`.
- Bước 7: Gặp `)`, POP `+` từ `Opr` vào `BLExp`, sau đó POP `(` và bỏ qua.
- Bước 8: Gặp `-`, xử lý tương tự cho đến hết.

Kết quả cuối cùng trong `BLExp`: `abc+*da*-`

Thuật toán tính giá trị biểu thức dạng hậu tố:

- Khởi tạo:
  - Sử dụng stack `BLExp` chứa biểu thức hậu tố.
  - Tạo một stack phụ `T` để lưu trữ các toán hạng và kết quả trung gian.
- Xử lý biểu thức: Thực hiện POP từng phần tử từ `BLExp` cho đến khi `BLExp` rỗng.
- Xử lý các phần tử:
  - Nếu phần tử là toán hạng:
    - PUSH toán hạng đó vào stack `T`.
  - Nếu phần tử là toán tử:
    - POP hai phần tử đầu từ stack `T` (lần lượt là `operand2` và `operand1`).
    - Thực hiện phép toán với hai toán hạng này, sử dụng toán tử vừa đọc: Ví dụ, với toán tử `+`, thực hiện phép tính `operand1 + operand2`.
    - PUSH kết quả của phép toán trở lại vào stack `T`.
- Hoàn tất: Khi `BLExp` rỗng, giá trị cuối cùng còn lại trong stack `T` chính là giá trị của biểu thức.

4.4. Cấu trúc dữ liệu hàng đợi (Queue)

4.4.1. Định nghĩa danh sách kiểu hàng đợi

Queue là một cấu trúc dữ liệu trừu tượng, tổ chức lưu trữ các đối tượng dưới dạng một danh sách tuyến tính mà việc bổ sung đối tượng được thực hiện ở đầu này danh sách và việc lấy đối tượng ra được thực hiện đầu kia của danh sách (FIFO – first in first out).

Queue còn được gọi là danh sách kiểu FIFO (First In First Out - vào trước ra trước)

4.4.2. Cấu trúc dữ liệu trừu tượng Queue

Cấu trúc dữ liệu Queue (Hàng đợi) là một cấu trúc dữ liệu trừu tượng được sử dụng để lưu trữ và truy cập dữ liệu theo nguyên tắc FIFO (First In, First Out), tức là phần tử được đưa vào đầu tiên sẽ được lấy ra đầu tiên.

Các thao tác cơ bản trong Queue:

- `push(Object o)`: bổ sung đối tượng o vào cuối Queue.
- `pop()`: xóa phần tử đầu tiên của Queue.
- `front()`: trả lại phần tử đầu Queue nhưng không xóa
- `back()`: trả lại phần tử cuối Queue nhưng không xóa
- `size()`: trả lại số phần tử hiện lưu trữ trong Queue.
- `empty()`: trả lại giá trị kiểu boolean để xác định Queue có lưu trữ phần tử nào hay không.

Chú ý: Queue rỗng thì phép toán `pop()`, `front()` và `back()` ***không thể thực hiện được.***

Một số ứng dụng của hàng đợi:

- Ứng dụng trực tiếp của Queue

Danh Sách Hàng Đợi: Quản lý lượt chờ của khách hàng tại quầy dịch vụ, trung tâm hỗ trợ, hoặc các tác vụ đợi xử lý trong hệ thống điều hành.

Truy Cập Nguồn Dùng Chung: Quản lý việc truy cập đến các tài nguyên chung như máy in hoặc máy chủ, đảm bảo thứ tự xử lý và tránh xung đột.

Đa Lập Trình: Hệ điều hành sử dụng hàng đợi để quản lý và sắp xếp các tiến trình đang chờ tài nguyên CPU, tối ưu hóa sử dụng CPU.

- Ứng dụng gián tiếp của Queue

Hỗ Trợ Thuật Toán: Sử dụng trong thuật toán tìm kiếm theo chiều rộng (BFS) và các hệ thống xử lý ngôn ngữ tự nhiên, quản lý nhiệm vụ và phân phối công việc.

Thành Phần Cấu Trúc Dữ Liệu: Làm nền tảng cho các cấu trúc dữ liệu phức tạp hơn như hàng đợi ưu tiên, hàng đợi vòng, và hàng đợi hai đầu, cung cấp các tính năng đặc biệt cho các ứng dụng cụ thể.

4.4.3. Cài đặt Queue bằng mảng

Code mẫu cài đặt Queue bằng mảng:

```
#include<bits/stdc++.h>
using namespace std;
#ifndef queue__cpp
#define queue__cpp
template <class T>
class Queue{
	int num, cap; //Lưu trữ số lượng phần tử hiện tại và sức chứa trong stack;
	int F, L; //Lưu trữ vị trí của phần tử front và back
	T *elem; //Con trỏ đến mảng động chứa các phần tử của stack.
public:
	Queue() {num = F = L = cap = 0; elem = NULL;}
	~Queue() {if(elem) delete[] elem;}
	int size() {return num;}
	bool empty() {return num == 0;}
	T &front() {return elem[F];}
	T &back() {return L == 0 ? elem[cap-1] : elem[L-1];}
```

```
        void pop() {F = (F+1) % cap; num--;}
        void push(T x){
            if(num == cap){
                cap = cap ? cap * 2 : 1;
                T *tem = new T[cap];
                for(int i = 0,j = F; j < F + num; j++, i++) tem[i] = elem[j % num];
                if(elem) delete[] elem;
                elem = tem;
                F = 0; L = num;
            }
            elem[L] = x ; L = (L + 1) % cap;
            num++;
        }
};
#endif
```

Trong đoạn code trên, chiến lược mở rộng gấp đôi đã được sử dụng để tăng kích thước của mảng trong Queue. Điều này giúp duy trì hiệu suất tối ưu cho các thao tác thêm phần tử vào Queue bằng cách giảm số lần phải cấp phát lại bộ nhớ và sao chép dữ liệu. Khi mảng đầy, kích thước của nó được tăng gấp đôi, giúp giảm thiểu số lần cần phải mở rộng mảng trong tương lai gần. Điều này không chỉ cải thiện tốc độ hoạt động của Queue mà còn giúp quản lý bộ nhớ hiệu quả hơn, tránh các thao tác cấp phát và giải phóng bộ nhớ tốn kém.

Bên cạnh đó ta cũng có thể cài đặt Queue bằng danh sách liên kết đơn. Nguồn tham khảo: https://ideone.com/4fIzIm

Chương V. Cấu trúc dữ liệu cây

5.1. Cấu trúc dữ liệu cây tổng quát (Tree)

5.1.1. Cây tổng quát

Cây là gì?

Cây (tree) là một cấu trúc dữ liệu bao gồm một tập hợp các nút (nodes), với mối quan hệ cha-con (parent-child) giữa các nút. Trong cây, có một nút đặc biệt gọi là nút gốc (root), là nút duy nhất không có cha. Từ nút gốc, các nhánh dẫn đến các nút con, tạo nên một cấu trúc phân cấp.

Các ứng dụng:

- Tổ chức biểu đồ
- Hệ thống file
- Các môi trường lập trình …

Một số khái niệm:

- Nút Gốc (Root Node): Là nút đầu tiên và duy nhất không có cha. Mọi nút khác trong cây có thể được truy cập từ nút gốc qua các cạnh (edges).
- Nút Con (Child Node): Là nút được liên kết trực tiếp với một nút khác, được gọi là nút cha (parent node). Mỗi nút có thể có nhiều nút con, nhưng mỗi nút chỉ có một nút cha.
- Nút Lá (Leaf Node): Là nút không có nút con nào, thường nằm ở các vị trí cuối cùng của cây.
- Cạnh (Edge): Là liên kết giữa hai nút trong cây, đại diện cho mối quan hệ cha-con giữa các nút đó.
- Độ sâu (Depth): Độ sâu của một nút là khoảng cách từ nút đó đến nút gốc
  - Nút gốc luôn có độ sâu là 0, vì không có cạnh nào từ nút gốc đến chính nó.
  - Độ sâu của các nút con được xác định bằng cách thêm 1 vào độ sâu của nút cha. Ví dụ, nếu nút cha có độ sâu là 2, thì các nút con của nó sẽ có độ sâu là 3.
- Chiều Cao (Height): Chiều cao của cây là số cạnh dài nhất từ nút gốc đến nút lá.
- Cây con (Subtree): Cây bao gồm một số nút của một cây ban đầu.

Cấu trúc dữ liệu cây:

Cấu trúc dữ liệu cây là một cấu trúc dữ liệu phi tuyến, trừu tượng và phân cấp, trong đó các nút (nodes) có quan hệ cha con với nhau. Cấu trúc cây bao gồm:

- Một nút gốc là nút không có cha.
- Các cây con xuất phát từ nút gốc. Mỗi nút trong cây có một đường duy nhất dẫn về nút gốc, do mỗi nút chỉ có duy nhất một cha.

Các thao tác có trong cấu trúc dữ liệu cây:

- Quản lý nút thông qua địa chỉ: Chúng ta quản lý và thao tác các nút trong cây thông qua địa chỉ của chúng, giúp dễ dàng truy cập và thao tác trên các nút.
- Phương thức chung:
  - `int size()`: Trả về số lượng nút hiện có trong cây.
  - `int isEmpty()`: Kiểm tra xem cây có rỗng không, trả về 1 nếu cây rỗng và 0 nếu cây không rỗng.
- Các phương pháp duyệt cây:
  - `void preorder(Node*)`: Duyệt tiền thứ tự (Preorder Traversal).
  - `void inorder(Node*)`: Duyệt trung thứ tự (Inorder Traversal).

- `void postorder(Node*)`: Duyệt hậu thứ tự (Postorder Traversal).

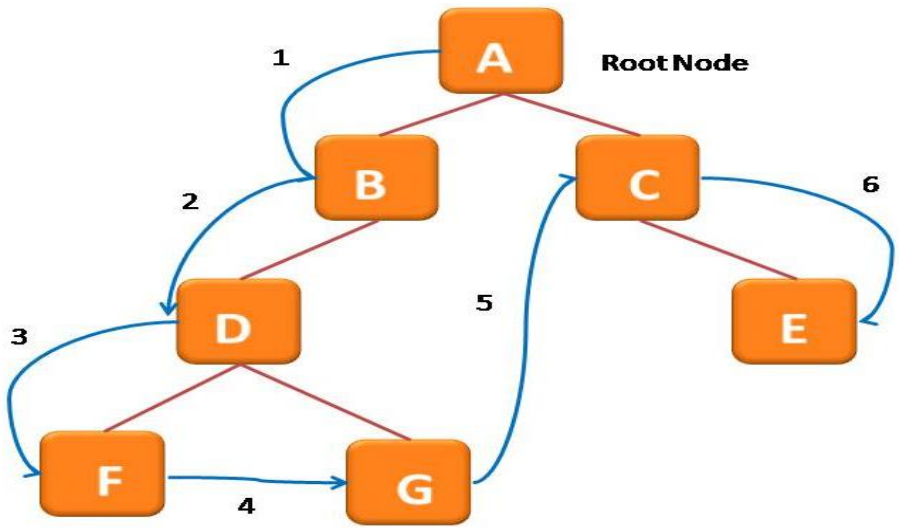
- Các phương thức truy cập:
  - `Node* root()`: Trả về địa chỉ của nút gốc, là điểm khởi đầu để truy cập các phần tử trong cây.
- Các phương thức truy vấn:
  - `int isInternal(Node*)`: Kiểm tra xem nút có phải là nút nội bộ không (có ít nhất một nút con).
  - `int isExternal(Node*)`: Kiểm tra xem nút có phải là nút lá không (không có nút con).
  - `int isRoot(Node*)`: Kiểm tra xem nút có phải là nút gốc của cây không.
- Các phương thức cập nhật:
  - `void insert(Node* parent, Element e)`: Thêm một phần tử mới e vào cây dưới một nút cha xác định parent.
  - `void remove(Node*)`: Xóa một nút khỏi cây.

5.1.2. Các thứ tự duyệt cây

Duyệt cây theo tiền thứ tự (Preorder Traversal): nút cha được thăm trước sau đó thăm các nút con, cháu, …

Algorithm preOrder(v):

```
if v != null then
      visit(v) // Thăm nút hiện tại v
      for each child w of v do
            preOrder(w) // Đệ quy duyệt cây con với gốc là w
```



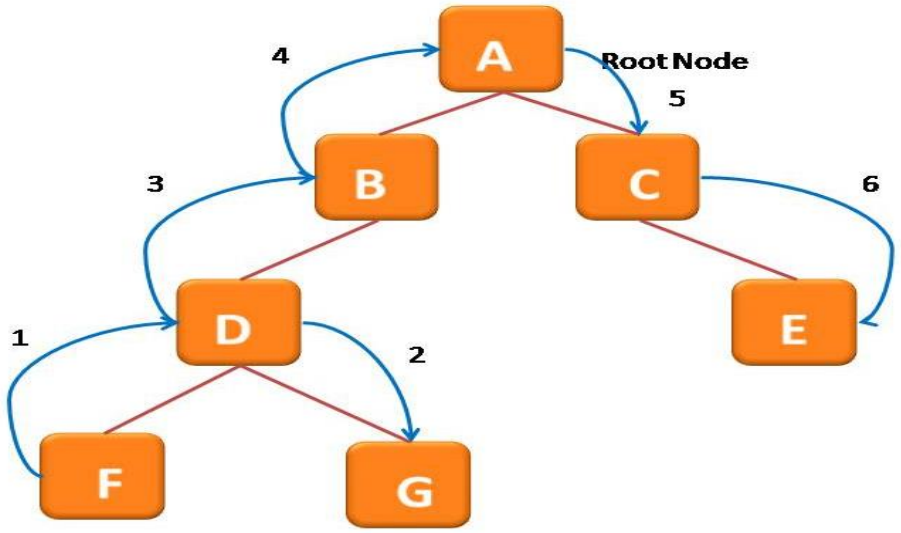
Mô hình duyệt cây theo tiền thứ tự

Duyệt cây theo trung thứ tự (Inorder Traversal): nút con được thăm trước sau đó thăm nút cha

Ứng dụng: Tính toán không gian sử dụng bởi các files và các thư mục con.

Algorithm inOrder(v)

```
if v != null then
      w = leftmost child of v // Lấy nút con trái nhất của v
      inOrder(w) // Đệ quy duyệt cây con trái
      visit(v) // Thăm nút hiện tại v
      for each child w1 of v do
            if w1 != w then
                  inOrder(w1) // Đệ quy duyệt các cây con khác
```



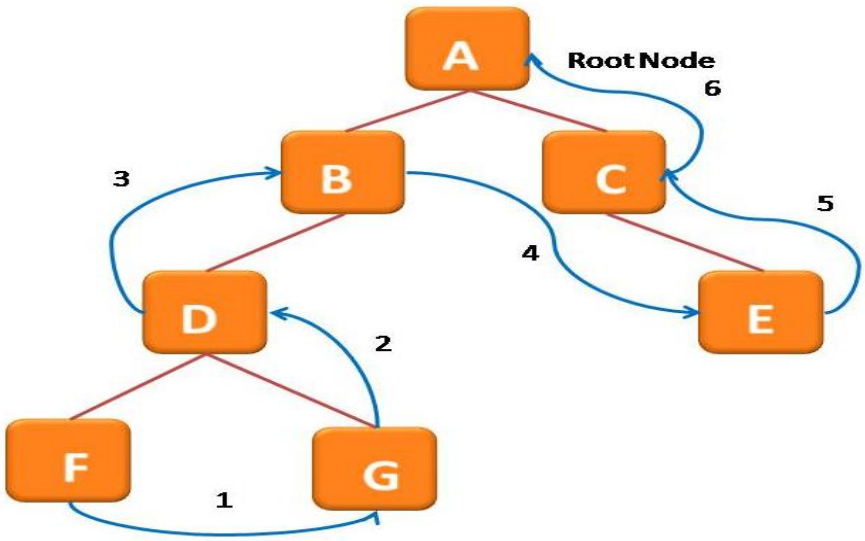
Mô hình duyệt cây theo trung thứ tự

Duyệt cây theo hậu thứ tự: nút con được thăm trước sau đó thăm nút cha

Ứng dụng: Tính toán không gian sử dụng bởi các files và các thư mục con

Algorithm postOrder(v)

```
if v != null then
      for each child w of v do
            postOrder(w) // Đệ quy duyệt các cây con của v
      visit(v) // Thăm nút hiện tại v sau khi đã thăm tất cả các nút con
```



Mô hình duyệt cây theo hậu thứ tự

5.1.3. Cấu trúc liên kết cây tổng quát

Mỗi nút là một đối tượng, đang lưu trữ:

- Phần tử (Element)
- Nút cha (Parent node)
- Lưu dãy địa chỉ của các nút con

Mỗi nút thể hiện một vị trí trong cấu trúc dữ liệu cây tổng quát.

Cấu trúc node của cây tổng quát:

- Thuộc tính:
  - `Object elem`: Lưu trữ dữ liệu của nút.
  - `TreeNode* parent`: Con trỏ đến nút cha của nút hiện tại.
  - `std::vector<TreeNode*> children`: Danh sách các nút con của nút hiện tại, sử dụng `std::vector` để quản lý động số lượng con.
- Phương thức:
  - `TreeNode* getParent()`: Trả về con trỏ đến nút cha của nút hiện tại.
  - `void setParent(TreeNode* p)`: Thiết lập nút cha của nút hiện tại với con trỏ `p`.
  - `TreeNode* getChild(int i)`: Trả về con trỏ đến nút con tại chỉ số `i`. Nếu chỉ số không hợp lệ, trả về `nullptr`.
  - `void insertChild(Object elem)`: Tạo một nút con mới với dữ liệu `elem`, thiết lập nút cha của nút con là nút hiện tại, và thêm nó vào danh sách các con.
  - `std::vector<TreeNode*> getChildren()`: Trả về danh sách các nút con của nút hiện tại.

- `Object getElem()`: Trả về dữ liệu của nút hiện tại.
- `void setElem(Object o)`: Thiết lập dữ liệu của nút hiện tại với giá trị `o`.

Cấu trúc Cây tổng quát

- Thuộc tính:
  - `TreeNode* root`: Con trỏ đến nút gốc của cây.
- Phương thức Truy cập:
  - `TreeNode* root()`: Trả về con trỏ đến nút gốc của cây.
- Các phương thức:
  - `int size()`: Trả về số lượng nút trong cây.
  - `bool isEmpty()`: Kiểm tra xem cây có rỗng không.
  - `bool isInternal(TreeNode* node)`: Kiểm tra nút node có phải là nút nội bộ không.
  - `bool isExternal(TreeNode* node)`: Kiểm tra xem nút node có phải là nút lá không.
  - `bool isRoot(TreeNode* node)`: Kiểm tra xem nút node có phải là nút gốc không.
  - `void preOrder(TreeNode* node, void (*visit)(TreeNode*))`: Duyệt cây theo thứ tự trước (preorder). Thực hiện hàm visit trên mỗi nút node theo thứ tự: thăm nút hiện tại, sau đó các nút con.
  - `void inOrder(TreeNode* node, void (*visit)(TreeNode*))`: Duyệt cây theo thứ tự giữa (inorder). Thực hiện hàm visit trên mỗi nút node theo thứ tự: thăm cây con trái, nút hiện tại, và cây con phải.
  - `void postOrder(TreeNode* node, void (*visit)(TreeNode*))`: Duyệt cây theo thứ tự sau (postorder). Thực hiện hàm visit trên mỗi nút node theo thứ tự: thăm các nút con trước, sau đó nút hiện tại.
  - `void insert(TreeNode* parent, Element elem)`: Thêm một nút mới với dữ liệu elem dưới nút cha parent.
  - `void remove(TreeNode* node)`: Xóa nút node khỏi cây.

5.2. Cấu trúc dữ liệu cây nhị phân (Binary Tree)

5.2.1. Khái quát về cây nhị phân

Cây nhị phân là một loại cấu trúc dữ liệu cây với các đặc điểm sau:

- Mỗi nút có tối đa hai nút con: Một nút trong cây nhị phân không có nhiều hơn hai nút con.
- Thứ tự của các nút con: Các nút con của mỗi nút được phân loại thành con trái và con phải. Cặp nút con của một nút có thứ tự cố định, với nút con trái đứng trước nút con phải.

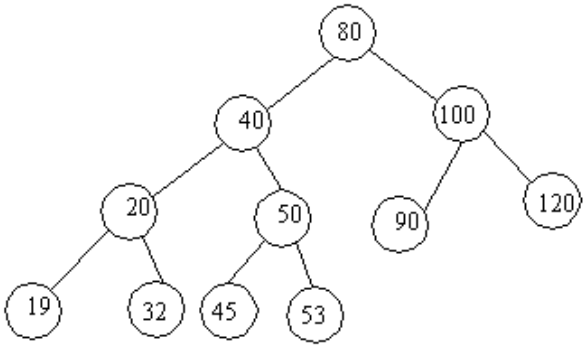
Định nghĩa cây nhị phân bằng đệ quy: Một cây nhị phân có thể là một cây chỉ có một nút (nút gốc) hoặc là một cây mà nút gốc của nó có một cặp nút con (con trái và con phải), mỗi nút con là gốc của một cây nhị phân khác.

Ứng dụng:

- Biểu diễn các biểu thức toán học.
- Quá trình quyết định.
- Tìm kiếm.

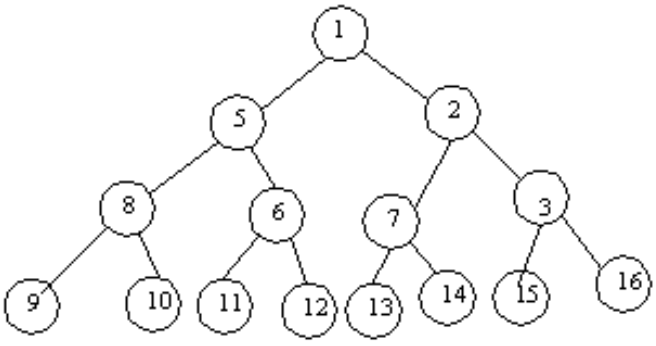
5.2.2. Một số định nghĩa của cây nhị phân

Cây nhị phân hoàn chỉnh là một cây nhị phân mà mọi mức của cây đều được lấp đầy hoàn toàn, ngoại trừ mức cuối cùng, nơi các nút được lấp đầy từ trái sang phải mà không có khoảng trống giữa các nút. Trong cây nhị phân hoàn chỉnh, tất cả các nút ở mức trước mức cuối cùng đều có hai nút con, và các nút ở mức cuối cùng được đặt gần nhau từ trái sang phải.



Ví dụ về cây nhị phân hoàn chỉnh

Cây nhị phân đầy đủ là một cây nhị phân trong đó mỗi nút nội bộ có chính xác hai nút con và tất cả các nút lá nằm ở cùng một mức. Trong cây nhị phân đầy đủ, tất cả các mức của cây đều được lấp đầy hoàn toàn ngoại trừ mức cuối cùng, nơi mà tất cả các nút lá nằm ở cùng một mức.



Ví dụ về cây nhị phân đầy đủ

5.2.3. Cấu trúc liên kết cho cây nhị phân

Cấu trúc dữ liệu cây nhị phân là một sự mở rộng của cây, kế thừa các phương thức của cấu trúc cây cơ bản và bổ sung thêm các phương thức đặc thù cho cây nhị phân.

Một nút là một đối tượng, đang lưu trữ:

- Phần tử (Element)
- Nút cha (Parent node)
- Nút con trái
- Nút con phải

Mỗi nút thể hiện một ví trí trong cấu trúc dữ liệu cây.

Cấu trúc một node của cây nhị phân:

- Thuộc tính:
  - `Object elem`: Lưu trữ dữ liệu của nút.
  - `BTreeNode* parent`: Con trỏ đến nút cha của nút hiện tại.
  - `BTreeNode *Left`: Con trỏ đến nút con trái của nút hiện tại.
  - `BTreeNode *Right`: Con trỏ đến nút con phải của nút hiện tại.
- Phương thức:
  - `BTreeNode* getParent()`: Trả về con trỏ đến nút cha của nút hiện tại.
  - `void setParent(BTreeNode* p)`: Thiết lập nút cha của nút hiện tại với con trỏ `p`.
  - `BTreeNode *getLeft()`: Trả về con trỏ đến nút con trái của nút hiện tại.
  - `BTreeNode *getRight()`: Trả về con trỏ đến nút con phải của nút hiện tại.
  - `void setLeft(BTreeNode *left)`: Thiết lập con trỏ đến nút con trái của nút hiện tại.
  - `void setRight(BTreeNode *right)`: Thiết lập con trỏ đến nút con phải nút hiện tại.
  - `void setParent(BTreeNode *parent)`: Thiết lập con trỏ đến nút cha của nút hiện tại.
  - `bool hasLeft()`: Kiểm tra xem nút hiện tại có con trái không.
  - `bool hasRight()`: Kiểm tra xem nút hiện tại có con phải không.
  - `Object getElem()`: Trả về dữ liệu của nút hiện tại.
  - `void setElem(Object o)`: Thiết lập dữ liệu của nút hiện tại với giá trị `o`.

Cấu trúc dữ liệu cây nhị phân:

- Thuộc Tính:
  - `BTreeNode *root`: Con trỏ đến nút gốc của cây nhị phân.
- Các Phương Thức Truy Cập:
  - `BTreeNode *getRoot()`: Trả về con trỏ đến nút gốc của cây.

- Các Phương Thức:
  - `int size()`: Trả về số lượng nút trong cây.
  - `bool isEmpty()`: Kiểm tra xem cây có rỗng không.
  - `bool isInternal(BTreeNode *node)`: Kiểm tra nút node có phải là nút nội bộ không.
  - `bool isExternal(BTreeNode *node)`: Kiểm tra nút node có phải là nút lá không.
  - `bool isRoot(BTreeNode *node)`: Kiểm tra nút node có phải nút gốc của cây không.
  - `void preOrder(BTreeNode *node, void (*visit)(BTreeNode *))`: Duyệt cây theo thứ tự tiền tố (pre-order).
  - `void inOrder(BTreeNode *node, void (*visit)(BTreeNode *))`: Duyệt cây theo thứ tự trung tố (in-order).
  - `void postOrder(BTreeNode *node, void (*visit)(BTreeNode *))`: Duyệt cây theo thứ tự hậu tố (post-order).
  - `BTreeNode* insert(BTreeNode *parent, element elem)`: Thêm một nút mới với giá trị elem dưới nút parent.
  - `void remove(BTreeNode *node)`: Xóa nút node khỏi cây.

5.3. Cây tìm kiếm nhị phân (Binary Search Tree)

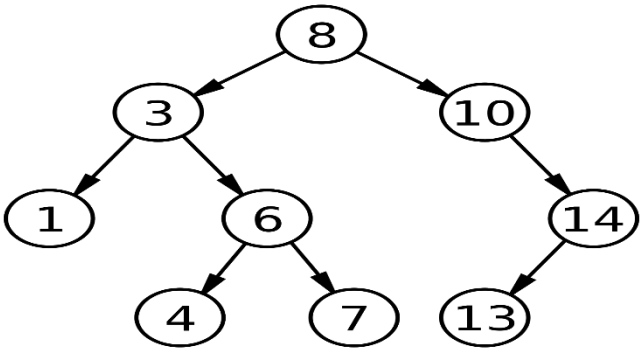
5.3.1. Khái niệm và cấu trúc cây tìm kiếm nhị phân

Cây tìm kiếm nhị phân (BST) là một loại cây nhị phân với các tính chất đặc biệt giúp thực hiện các thao tác tìm kiếm, chèn và xóa một cách hiệu quả. Cụ thể:

Tính Chất: Trong một cây tìm kiếm nhị phân, mỗi nút có một giá trị, và giá trị của mọi nút con trái của nút hiện tại nhỏ hơn giá trị của nút hiện tại, trong khi giá trị của mọi nút con phải của nút hiện tại lớn hơn giá trị của nút hiện tại. Điều này đảm bảo rằng các giá trị trong cây được sắp xếp theo thứ tự, tạo điều kiện thuận lợi cho việc tìm kiếm và thao tác trên cây.

Cấu Trúc: Mỗi nút trong cây tìm kiếm nhị phân có tối đa hai nút con: một nút con trái và một nút con phải. Cây tìm kiếm nhị phân có thể không hoàn chỉnh hoặc không cân bằng, nhưng vẫn duy trì tính chất của cây tìm kiếm.

Ứng Dụng: Cây tìm kiếm nhị phân thường được sử dụng trong các thuật toán tìm kiếm, sắp xếp, và các cấu trúc dữ liệu như bộ nhớ đệm (cache) và chỉ mục cơ sở dữ liệu.



Ví dụ về cây tìm kiếm nhị phân

Cấu trúc node của cây tìm kiếm nhị phân:

- Thuộc Tính:
  - `Keys key`: Tập các giá trị được sắp xếp có thứ tự.
  - `T elem`: Dữ liệu hoặc giá trị của nút.
  - `Node *Parent`: Con trỏ đến nút cha của nút hiện tại.
  - `Node *Left`: Con trỏ đến nút con trái của nút hiện tại.
  - `Node *Right`: Con trỏ đến nút con phải của nút hiện tại.
- Phương Thức:
  - `Node *getParent()`: Trả về con trỏ đến nút cha của nút hiện tại.
  - `Node *getLeft()`: Trả về con trỏ đến nút con trái của nút hiện tại.
  - `Node *getRight()`: Trả về con trỏ đến nút con phải của nút hiện tại.
  - `void setLeft(Node *left)`: Thiết lập con trỏ đến nút con trái của nút hiện tại.
  - `void setRight(Node *right)`: Thiết lập con trỏ đến nút con phải của nút hiện tại.
  - `void setParent(Node *parent)`: Thiết lập con trỏ đến nút cha của nút hiện tại.
  - `bool hasLeft()`: Kiểm tra xem nút hiện tại có con trái không.
  - `bool hasRight()`: Kiểm tra xem nút hiện tại có con phải không.
  - `T getElem()`: Trả về giá trị hoặc dữ liệu của nút hiện tại.
  - `void setElem(T o)`: Thiết lập giá trị hoặc dữ liệu của nút hiện tại.
  - `void setKey(Keys k)`: Thiết lập khóa (key) của nút hiện tại.
  - `Keys getKey()`: Trả về khóa (key) của nút hiện tại.

Cấu trúc dữ liệu cây tìm kiếm nhị phân:

- Thuộc Tính:
  - `Node<Keys, T> *root`: Con trỏ đến nút gốc của cây tìm kiếm nhị phân.
- Phương Thức Truy Cập:
  - `Node<Keys, T> *root()`: Trả về con trỏ đến nút gốc của cây.
- Các Phương Thức:
  - `int size()`: Trả về số lượng nút trong cây.
  - `bool isEmpty()`: Kiểm tra xem cây có rỗng không.
  - `bool isInternal(Node<Keys, T> *node)`: Kiểm tra nút node có là nút nội bộ không.
  - `bool isExternal(Node<Keys, T> *node)`: Kiểm tra nút node có phải là nút lá không.
  - `bool isRoot(Node<Keys, T> *node)`: Kiểm tra nút node có phải là nút gốc không.
  - `void preOrder(Node<Keys, T> *node)`: Duyệt cây theo thứ tự tiền tố (pre-order), bắt đầu từ nút node.

- `void inOrder(Node<Keys, T> *node)`: Duyệt cây theo thứ tự trung tố (in-order), bắt đầu từ nút node.
- `void postOrder(Node<Keys, T> *node)`: Duyệt cây theo thứ tự hậu tố (post-order), bắt đầu từ nút node.
- `Node<Keys, T> *search(Keys key, Node<Keys, T> *node)`: Tìm kiếm nút có khóa key trong cây bắt đầu từ nút node.
- `Node<Keys, T> *insert(Keys key, T value)`: Chèn một nút mới với khóa key và giá trị value vào cây.
- `void remove(Keys key)`: Xóa nút có khóa key khỏi cây.

5.3.2. Thuật toán tìm kiếm

Giả mã của thuật toán tìm kiếm trong cây tìm kiếm nhị phân:

Function search(k, v):

```
if v is NULL
      return NULL // Không tìm thấy khóa k trong cây
else if k < v.getKey()
      return search(k, v.getLeft()) // Tìm kiếm trong cây con trái
else if k == v.getKey()
      return v  // Khóa k được tìm thấy tại nút v
else  // k > v.getKey()
      return search(k, v.getRight()) // Tìm kiếm trong cây con phải
```

Giải thích:

- Kiểm Tra Nút Rỗng: Nếu nút `v` là `NULL`, nghĩa là cây rỗng hoặc đã tìm đến nút không tồn tại. Do đó, trả về `NULL`, báo hiệu rằng khóa `k` không có trong cây.
- So Sánh Khóa:
  - Nếu `k` nhỏ hơn khóa của nút hiện tại `(v.getKey())`: Tìm kiếm tiếp tục trong cây con bên trái của nút hiện tại. Gọi đệ quy hàm `search` với nút con trái của `v`.
  - Nếu `k` bằng khóa của nút hiện tại `(v.getKey())`: Đã tìm thấy khóa `k` tại nút hiện tại. Trả về nút `v` chứa khóa k.
  - Nếu `k` lớn hơn khóa của nút hiện tại `(v.getKey())`: Tìm kiếm tiếp tục trong cây con bên phải của nút hiện tại. Gọi đệ quy hàm search với nút con phải của `v`.

Phân tích thời gian chạy của thuật toán tìm kiếm:

Là thuật toán đệ quy: Thuật toán tìm kiếm trong cây tìm kiếm nhị phân sử dụng đệ quy để duyệt cây. Mỗi lần gọi đệ quy thực hiện một số phép toán cơ bản, chẳng hạn như so sánh khóa và di chuyển đến nút con trái hoặc phải.

Thời gian cho một lần gọi đệ quy: Mỗi lần gọi đệ quy thực hiện các phép toán cơ bản không đổi và cần thời gian là O(1).

Số lần gọi đệ quy: Thuật toán gọi đệ quy dọc theo các nút từ nút gốc đến nút mà khóa tìm kiếm có thể nằm. Mỗi lần gọi đệ quy làm giảm chiều cao cây một mức.

Chiều cao của cây: Trong cây tìm kiếm nhị phân, số lần gọi đệ quy cần thực hiện tối đa không vượt quá chiều cao h của cây.

- Trường hợp tốt nhất: Cây cân bằng, chiều cao h của cây là O(log n), do đó thời gian chạy là O(log n).
- Trường hợp xấu nhất: Cây không cân bằng (danh sách liên kết), chiều cao h của cây là O(n), do đó thời gian chạy là O(n).

Kết luận: Thời gian chạy của thuật toán tìm kiếm trong cây tìm kiếm nhị phân là O(h), trong đó h là chiều cao của cây. Thời gian chạy có thể là O(log n) trong trường hợp tốt nhất và O(n) trong trường hợp xấu nhất.

5.3.3. Phân tích một số phương thức có trong cây tìm kiếm nhị phân

Chèn nút vào cây tìm kiếm nhị phân:

Khi chèn một nút mới vào cây tìm kiếm nhị phân, cần duy trì cấu trúc của cây để nó vẫn thỏa mãn các điều kiện của một cây tìm kiếm nhị phân, tức là:

- Mỗi nút con bên trái có giá trị nhỏ hơn nút cha.
- Mỗi nút con bên phải có giá trị lớn hơn nút cha.

Giả mã của thuật toán chèn thêm nút vào cây:

Function insert(key, value, node) :

```
if node is NULL
      return new Node(key, value) //Tạo nút mới nếu vị trí chèn rỗng
if key < node.getKey()
      node.setLeft(insert(key, value, node.getLeft())) //Cây con trái
else
      node.setRight(insert(key, value, node.getRight())) //Cây con phải
return node // Trả về nút hiện tại sau khi chèn
```

Giải thích:

- Kiểm tra nút rỗng: Nếu node là NULL, nghĩa là đã tìm đến vị trí chèn thích hợp. Tạo một nút mới với khóa key và giá trị value, và trả về nút mới.
- So sánh khóa:
  - Nếu khóa `key` nhỏ hơn khóa của nút hiện tại (`node.getKey()`), tiếp tục chèn vào cây con bên trái (`node.getLeft()`).
  - Nếu khóa `key` lớn hơn hoặc bằng khóa của nút hiện tại, tiếp tục chèn vào cây con bên phải (`node.getRight()`).
- Đệ quy: Hàm insert được gọi đệ quy để tìm vị trí chèn thích hợp. Sau khi chèn, trả về nút hiện tại (node), để cập nhật con trỏ bên trái hoặc bên phải của nút cha.

Thời gian chạy:

- Trường hợp tốt nhất: Cây cân bằng, thời gian chạy là O(log n), n là số nút trong cây.
- Trường hợp xấu nhất: Cây không cân bằng (danh sách liên kết), thời gian chạy là O(n).

Xóa nút trên cây tìm kiếm nhị phân:

Khi xóa một nút từ cây tìm kiếm nhị phân, cần đảm bảo rằng cây vẫn duy trì cấu trúc của cây tìm kiếm nhị phân. Có ba trường hợp chính khi xóa một nút:

- Nút cần xóa là nút lá (không có con): Đơn giản chỉ cần xóa nút đó.
- Nút cần xóa có một con: Thay thế nút đó bằng nút con của nó.
- Nút cần xóa có hai con: Tìm nút kế tiếp (hoặc nút kế trước) để thay thế nút cần xóa, sau đó xóa nút kế tiếp (hoặc kế trước).

Giả mã của thuật toán xóa nút trên cây:

Function delete(key, node):

```
if node is NULL
      return NULL  // Nút không tồn tại
if key < node.getKey()
      node.setLeft(delete(key, node.getLeft()))  // Xóa trong cây con trái
else if key > node.getKey()
      node.setRight(delete(key, node.getRight())) //Xóa trong cây con phải
else
      // Nút cần xóa được tìm thấy
      if node.getLeft() is NULL
```

```
            return node.getRight()  // Thay thế bằng nút con phải
        else if node.getRight() is NULL
            return node.getLeft()  // Thay thế bằng nút con trái
        else
            // Nút có hai con
            successor = findMin(node.getRight())
            // Tìm nút nhỏ nhất trong cây con phải
            node.setKey(successor.getKey())
            // Thay khóa của nút cần xóa bằng khóa của nút kế tiếp
            node.setRight(delete(successor.getKey(), node.getRight()))
            // Xóa nút kế tiếp
    return node
```

Function findMin(node):

```
while node.getLeft() is not NULL
    node = node.getLeft()
return node
```

Giải thích:

- Tìm nút để xóa:
  - Nếu khóa cần xóa nhỏ hơn khóa của nút hiện tại, tiếp tục xóa trong cây con trái.
  - Nếu khóa cần xóa lớn hơn khóa của nút hiện tại, tiếp tục xóa trong cây con phải.
  - Nếu khóa cần xóa bằng khóa của nút hiện tại, nút cần xóa đã được tìm thấy.
- Xử lý các trường hợp xóa:
  - Nút là nút lá (không có con): Trả về `NULL`, xóa nút hiện tại.
  - Nút có một con: Thay thế nút hiện tại bằng nút con của nó (trái hoặc phải).
  - Nút có hai con: Tìm nút kế tiếp trong cây con phải (nút nhỏ nhất trong cây con phải), thay thế khóa của nút cần xóa với khóa của nút kế tiếp, sau đó xóa nút kế tiếp.
- Cập Nhật Cây: Sau khi xóa, trả về nút hiện tại để cập nhật cấu trúc cây.

Thời Gian Chạy

- Trường hợp tốt nhất: Cây cân bằng, thời gian chạy là O(log n).
- Trường hợp xấu nhất: Cây không cân bằng (danh sách liên kết), thời gian chạy là O(n).

Chương VI. Các thuật toán sắp xếp

6.1. Các thuật toán sắp xếp đơn giản

6.1.1. Thuật toán sắp xếp chọn (Selection Sort)

Thuật toán sắp xếp đơn giản nhất được đề cập đến là thuật toán sắp xếp chọn. Thuật toán thực hiện sắp xếp dãy các đối tượng bằng cách lặp lại việc tìm kiếm phần tử có giá trị nhỏ nhất từ thành phần chưa được sắp xếp trong mảng và đặt nó vào vị trí đầu tiên của dãy. Trên dãy các đối tượng ban đầu, thuật toán luôn duy trì hai dãy con: dãy con đã được sắp xếp là các phần tử bên trái của dãy và dãy con chưa được sắp xếp là các phần tử bên phải của dãy. Quá trình lặp sẽ kết thúc khi dãy con chưa được sắp xếp chỉ còn lại đúng một phần tử.

Giả mã của thuật toán sắp xếp chọn:

```
Function selectionSort(array)
    n = length(array)
    for i from 0 to n-1
        minIndex = i
        for j from i+1 to n
            if array[j] < array[minIndex]
                minIndex = j
        // Hoán đổi giá trị array[i] và array[minIndex]
        swap(array[i], array[minIndex])
```

Giải thích:

- Khởi tạo: `i` chạy từ đầu đến cuối của mảng, xác định vị trí phần tử hiện tại cần sắp xếp.
- Tìm phần tử nhỏ nhất:
  - Khởi tạo `minIndex` là chỉ số của phần tử đầu tiên chưa được sắp xếp.
  - Duyệt qua phần còn lại của mảng (từ `i+1` đến cuối), tìm phần tử nhỏ nhất. Cập nhật `minIndex` nếu tìm thấy phần tử nhỏ hơn.
- Hoán đổi: Hoán đổi phần tử ở vị trí `i` với phần tử nhỏ nhất tìm được (`minIndex`).
- Lặp lại: Lặp lại cho đến khi tất cả các phần tử được sắp xếp.

Thời gian chạy: Thời gian chạy là O(n^2) trong mọi trường hợp (tốt nhất, trung bình, xấu nhất) vì thuật toán phải tìm phần tử nhỏ nhất trong phần chưa sắp xếp và thực hiện hoán đổi.

6.1.2. Thuật toán sắp xếp chèn (Insertion Sort)

Thuật toán sắp xếp chèn là một thuật toán sắp xếp đơn giản và trực quan. Nó hoạt động bằng cách chia mảng thành hai phần: phần đã được sắp xếp và phần chưa được sắp xếp. Thuật toán lặp qua từng phần tử trong phần chưa được sắp xếp và chèn nó vào vị trí đúng trong phần đã được sắp xếp.

Giả mã của thuật toán sắp xếp chèn:

```
Function insertionSort(array)
    n = length(array)
    for i from 1 to n-1
        key = array[i]
        j = i - 1
        // Di chuyển các phần tử lớn hơn key sang phải
        while j >= 0 and array[j] > key
            array[j + 1] = array[j]
            j = j - 1
        // Chèn key vào vị trí đúng
        array[j + 1] = key
```

Giải thích:

- Khởi tạo: `i` bắt đầu từ `1` đến `n-1`, vì phần tử đầu tiên được coi là đã được sắp xếp sẵn.
- Lưu trữ phần tử đang xét: `key` lưu trữ giá trị của phần tử tại chỉ số `i` mà chúng ta muốn chèn vào vị trí đúng trong phần đã được sắp xếp.
- Di chuyển các phần tử: Duyệt từ phải sang trái (từ `j = i-1` đến `0`), di chuyển tất cả các phần tử lớn hơn `key` sang phải để tạo không gian cho `key`.
- Chèn phần tử: Đặt `key` vào vị trí đúng (tức là `array[j + 1]`).

Thời gian chạy:

- Thời gian chạy là O(n^2) trong trường hợp xấu nhất và trung bình, do thuật toán phải so sánh và di chuyển các phần tử trong phần đã sắp xếp.
- Thời gian chạy là O(n) trong trường hợp tốt nhất khi mảng đã được sắp xếp sẵn.

Ứng dụng: Sắp xếp chèn được sử dụng khi số lượng phần tử nhỏ hoặc khi dữ liệu gần như đã được sắp xếp sẵn. Nó dễ triển khai và có thể hoạt động hiệu quả trong các tình huống này.

6.1.3. Thuật toán sắp xếp nổi bọt (Bubble Sort)

Sắp xếp nổi bọt là một trong những thuật toán sắp xếp đơn giản và cơ bản nhất. Nó hoạt động bằng cách liên tục duyệt qua danh sách, so sánh các phần tử kề nhau và hoán đổi chúng nếu chúng nằm sai thứ tự. Quá trình này được lặp lại cho đến khi danh sách được sắp xếp hoàn toàn.

Giả mã của thuật toán sắp xếp nổi bọt:

```
Function bubbleSort(array)
    n = length(array)
    for i from 0 to n-1
        for j from 0 to n-i-1
            if array[j] > array[j+1]
                // Hoán đổi array[j] và array[j+1]
                swap(array[j], array[j+1])
```

Giải thích:

- Khởi tạo: `i` là biến đếm của vòng lặp ngoài, đại diện cho số lần duyệt qua mảng cần thực hiện. Vòng lặp này chạy từ `0` đến `n-1`.
- So sánh và hoán đổi:
  - Vòng lặp bên trong chạy từ `0` đến `n-i-1` và so sánh các phần tử kề nhau.
  - Nếu `array[j] > array[j+1]`, hai phần tử này được hoán đổi vị trí để đặt phần tử lớn hơn về phía sau.
- Tối ưu hóa: Mỗi lần kết thúc một lần lặp của vòng lặp ngoài, phần tử lớn nhất trong phần chưa sắp xếp được đẩy về cuối danh sách, vì vậy vòng lặp bên trong giảm đi một lần duyệt.

Thời gian chạy:

- Thời gian chạy là O(n^2) trong cả trường hợp tốt nhất, trung bình và xấu nhất, do thuật toán phải kiểm tra tất cả các cặp phần tử.
- Bubble Sort không phải là thuật toán hiệu quả cho các tập dữ liệu lớn, nhưng nó dễ hiểu và dễ triển khai.

Ứng dụng:

- Bubble Sort thường được sử dụng trong giáo dục để giới thiệu các khái niệm cơ bản về thuật toán sắp xếp.
- Có thể hiệu quả cho các tập dữ liệu nhỏ hoặc khi dữ liệu gần như đã được sắp xếp sẵn.

6.2. Thuật toán sắp xếp nhanh (Quick Sort)

Thuật toán Quick Sort là một phương pháp sắp xếp sử dụng mô hình chia để trị (Divide and Conquer). Cơ chế hoạt động của thuật toán xoay quanh một phần tử đặc biệt gọi là "chốt" (pivot). Việc chọn vị trí của phần tử chốt trong dãy quyết định các phiên bản khác nhau của Quick Sort, bao gồm:

- Chọn phần tử đầu tiên trong dãy làm chốt.
- Chọn phần tử cuối cùng trong dãy làm chốt.
- Chọn phần tử ở giữa dãy làm chốt.
- Chọn ngẫu nhiên một phần tử trong dãy làm chốt.

Thành phần quan trọng của thuật toán Quick Sort là thủ tục phân đoạn (Partition). Thủ tục này có hai nhiệm vụ chính:

- Xác định vị trí chính xác của phần tử chốt trong dãy nếu dãy được sắp xếp.
- Chia dãy ban đầu thành hai dãy con: dãy con phía trước phần tử chốt chứa các phần tử nhỏ hơn hoặc bằng chốt, và dãy con phía sau chốt chứa các phần tử lớn hơn chốt.

Quá trình này giúp đảm bảo rằng mỗi phần tử chốt sẽ được đặt đúng vị trí cuối cùng của nó trong dãy đã sắp xếp, và sau đó thuật toán tiếp tục áp dụng Quick Sort đệ quy lên hai dãy con để hoàn tất quá trình sắp xếp.

Giả mã thuật toán Partition chọn phần tử cuối làm chốt và thuật toán Quick Sort tương ứng:

Function quickSort(array, low, high)

```
if low < high
      // p là chỉ số phân chia
      p = partition(array, low, high)
      quickSort(array, low, p - 1)  //Sắp xếp các phần tử trước chốt
      quickSort(array, p + 1, high) //Sắp xếp các phần tử sau chốt
```

Function partition(array, low, high)

```
pivot = array[high]  // Chọn phần tử cuối làm chốt
i = low - 1  // Chỉ số của phần tử nhỏ hơn
for j from low to high - 1
      if array[j] <= pivot
            i = i + 1
            swap(array[i], array[j])
swap(array[i + 1], array[high])
return i + 1
```

Giải thích:

- Chọn chốt: `pivot` là phần tử mà các phần tử khác trong mảng sẽ được sắp xếp xung quanh. Trong trường hợp này, chốt được chọn là phần tử cuối của mảng.
- Phân chia: `partition` là bước chính của thuật toán, nơi mảng được phân chia dựa trên chốt. Các phần tử nhỏ hơn chốt được di chuyển về phía trước, và các phần tử lớn hơn chốt được di chuyển về phía sau. Phần tử chốt được đặt ở vị trí chính xác của nó trong mảng đã được sắp xếp.
- Sắp xếp đệ quy: Sau khi mảng được phân chia, thuật toán `quickSort` được gọi đệ quy trên hai mảng con: một mảng chứa các phần tử nhỏ hơn chốt và mảng kia chứa các phần tử lớn hơn chốt.

Thời gian chạy:

- Trung bình: O(n log n), do tính chất chia đôi mảng của thuật toán.
- Tốt nhất: O(n log n), xảy ra khi chốt phân chia mảng đều.
- Xấu nhất: O(n^2), xảy ra khi chốt là phần tử lớn nhất hoặc nhỏ nhất, dẫn đến phân chia không đều (có thể cải thiện bằng cách chọn chốt thông minh hơn).

Ứng dụng:

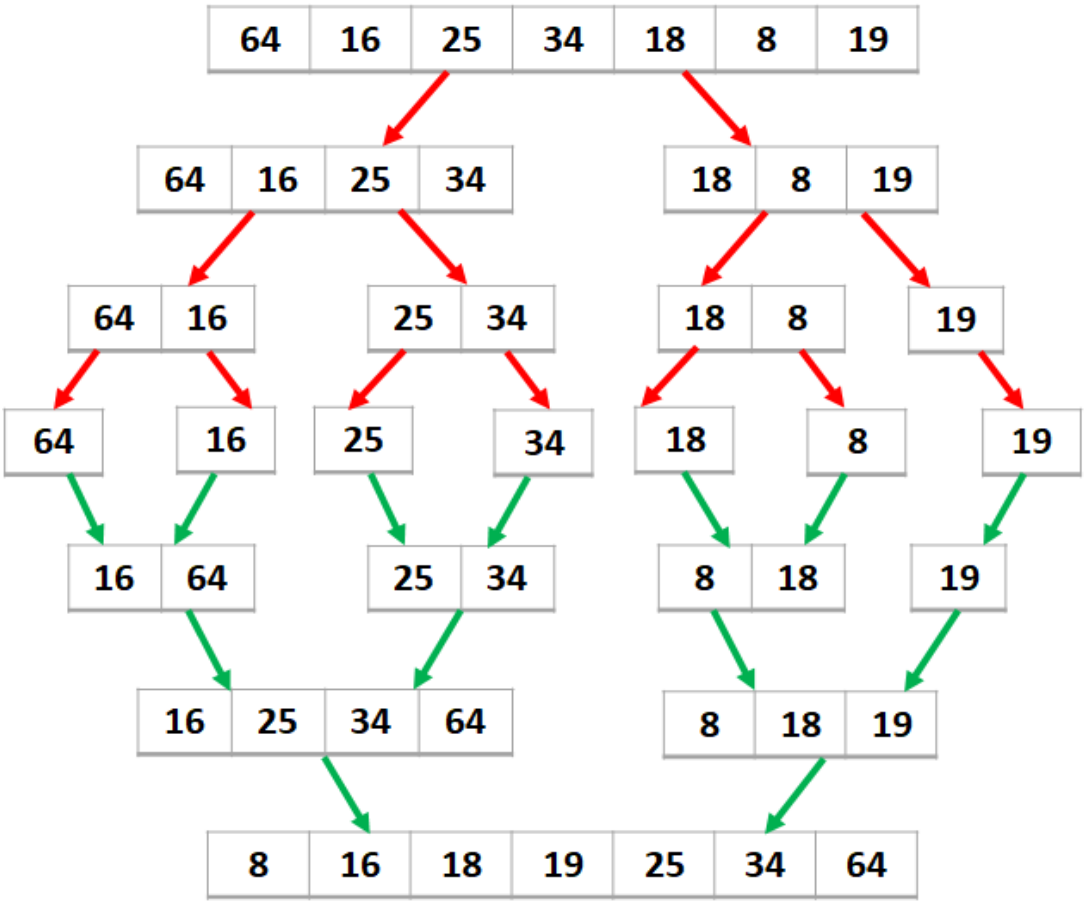
- Quick Sort thường được sử dụng khi cần sắp xếp các tập dữ liệu lớn, do hiệu quả trung bình của nó tốt hơn nhiều so với các thuật toán sắp xếp cơ bản như Bubble Sort hoặc Insertion Sort.
- Nó được ưa chuộng trong nhiều ứng dụng vì hiệu quả và sự đơn giản của nó trong triển khai.

6.3. Thuật toán sắp xếp trộn (Merge Sort)

Giống như Quick Sort, Merge Sort cũng được xây dựng theo mô hình chia để trị (Divide and Conquer). Thuật toán này chia mảng cần sắp xếp thành hai phần bằng nhau. Sau đó, nó gọi đệ quy để sắp xếp từng phần riêng biệt và cuối cùng hợp nhất hai phần đã sắp xếp lại với nhau. Quy trình thực hiện Merge Sort bao gồm bốn bước chính:

- Tìm điểm giữa của mảng và chia mảng thành hai phần.
- Thực hiện Merge Sort cho phần thứ nhất.
- Thực hiện Merge Sort cho phần thứ hai.

- Hợp nhất hai phần đã được sắp xếp lại với nhau.



Hình ảnh minh họa thuật toán Merge Sort

Mấu chốt của thuật toán Merge Sort là việc xây dựng một thủ tục hợp nhất (Merge). Thủ tục Merge có nhiệm vụ hợp nhất hai mảng đã được sắp xếp để tạo thành một mảng mới cũng được sắp xếp. Vấn đề hợp nhất có thể được phát biểu như sau:

Bài toán hợp nhất (Merge): Cho hai nửa của một mảng `Arr[1,..,m]` và `Arr[m+1,..,r]` đã được sắp xếp. Nhiệm vụ là hợp nhất hai nửa này thành mảng `Arr[1, 2,..,r]` cũng được sắp xếp.

Giả mã của thuật toán Merge và thuật toán Merge Sort:

Function mergeSort(array, left, right)

```
if left < right
      mid = (left + right) / 2
      mergeSort(array, left, mid)
      mergeSort(array, mid + 1, right)
      merge(array, left, mid, right)
```

Function merge(array, left, mid, right)

```
n1 = mid - left + 1

n2 = right - mid

LeftArray = new array[n1]

RightArray = new array[n2]

for i from 0 to n1 - 1

      LeftArray[i] = array[left + i]

for j from 0 to n2 - 1

      RightArray[j] = array[mid + 1 + j]

i = 0

j = 0

k = left

while i < n1 and j < n2

      if LeftArray[i] <= RightArray[j]

            array[k] = LeftArray[i]

            i = i + 1

      else

            array[k] = RightArray[j]

            j = j + 1

      k = k + 1

while i < n1

      array[k] = LeftArray[i]

      i = i + 1

      k = k + 1

while j < n2

      array[k] = RightArray[j]

      j = j + 1

      k = k + 1
```

Giải thích:

- Chia để trị: Thuật toán `mergeSort` chia mảng thành hai nửa bằng nhau bằng cách tìm chỉ số giữa (`mid`). Sau đó, nó gọi đệ quy `mergeSort` cho từng nửa của mảng.
- Hợp nhất: Sau khi các mảng con được sắp xếp, chúng được hợp nhất lại bằng cách sử dụng hàm `merge`. Hàm này tạo ra hai mảng tạm thời để lưu trữ các giá trị từ hai nửa của mảng, sau đó so sánh các phần tử từ mảng tạm thời và sao chép chúng lại vào mảng chính theo thứ tự đã sắp xếp.
- Tổng hợp: Quy trình chia và hợp nhất tiếp tục cho đến khi toàn bộ mảng được sắp xếp.

Thời gian chạy:

- Tốt nhất, trung bình, xấu nhất: O(n log n) do mảng được chia liên tục thành hai phần và mỗi phần phải được hợp nhất lại.

Ứng dụng:

- Merge Sort là lựa chọn tốt khi cần sắp xếp các dữ liệu lớn do tính ổn định và hiệu quả của nó, đặc biệt khi dữ liệu không thể được lưu trữ trong bộ nhớ và phải xử lý từ đĩa cứng hoặc qua mạng.

6.4. Thuật toán sắp xếp vun đống (Heap Sort)

Thuật toán Heap Sort là một kỹ thuật sắp xếp dựa trên cấu trúc dữ liệu heap. Để sắp xếp mảng theo thứ tự tăng dần, thuật toán sử dụng Max Heap, trong khi để sắp xếp theo thứ tự giảm dần, thuật toán sử dụng Min Heap. Do cấu trúc heap là một cây nhị phân đầy đủ, nó có thể được biểu diễn hiệu quả bằng mảng, trong đó phần tử tại chỉ số i có con trái tại chỉ số 2i + 1 và con phải tại chỉ số 2i + 2.

Tư tưởng của Heap Sort tương tự như Selection Sort, trong đó phần tử lớn nhất được chọn và đặt vào vị trí cuối cùng. Tuy nhiên, khác với Selection Sort, trong Heap Sort, phần tử lớn nhất luôn nằm ở đầu heap và các phần tử con luôn nhỏ hơn hoặc bằng phần tử gốc.

Các bước chính của thuật toán Heap Sort bao gồm:

- Xây dựng Max Heap: Tạo Max Heap từ mảng ban đầu. Ví dụ, với mảng A[] = {9, 7, 12, 8, 6, 5}, Max Heap được xây dựng thành A[] = {12, 8, 9, 7, 6, 5}.
- Đổi chỗ phần tử lớn nhất: Sau khi xây dựng Max Heap, phần tử lớn nhất nằm ở đầu mảng. Ta hoán đổi phần tử này với phần tử cuối cùng, nhận được dãy A[] = {5, 8, 9, 7, 6, 12}.
- Lặp lại quá trình: Xây dựng lại Max Heap cho n-1 phần tử đầu tiên và lặp lại quá trình cho đến khi chỉ còn lại một phần tử trong heap.

Giả mã của thuật toán Heapify và thuật toán Heap Sort:

Function heapSort(array)

```
n = length(array)
// Xây dựng Max Heap từ mảng
for i from n // 2 - 1 down to 0
      heapify(array, n, i)
// Trích xuất từng phần tử từ heap
for i from n - 1 to 1
      swap(array[0], array[i])
      heapify(array, i, 0)
```

Function heapify(array, n, i)

```
largest = i        // Khởi tạo largest là root
left = 2 * i + 1   // Chỉ số của con trái
right = 2 * i + 2  // Chỉ số của con phải
// Nếu con trái lớn hơn root
if left < n and array[left] > array[largest]
      largest = left
// Nếu con phải lớn hơn largest
if right < n and array[right] > array[largest]
      largest = right
// Nếu largest không phải là root
if largest != i
      swap(array[i], array[largest])
      heapify(array, n, largest)
```

Giải thích:

- Xây dựng Max Heap:
  - Vòng lặp từ `n // 2 - 1` đến `0`: Xử lý tất cả các nút không phải lá để đảm bảo toàn bộ mảng trở thành Max Heap.
  - Hàm `heapify(array, n, i)`: Đảm bảo rằng phần tử tại chỉ số i và các con của nó tuân theo thuộc tính Max Heap.
- Trích xuất phần tử lớn nhất:

- Đổi chỗ phần tử gốc với phần tử cuối cùng của dãy.
- Giảm kích thước Heap và áp dụng `heapify` để khôi phục thuộc tính Max Heap.

- Lặp lại quá trình trích xuất và xây dựng Heap cho đến khi Heap chỉ còn lại một phần tử.

Phân tích thời gian chạy:

- Xây dựng Max Heap: Thời gian chạy của bước này là O(n). Mặc dù có vẻ như mỗi gọi `heapify` tốn O(log n) thời gian, nhưng do cách gọi `heapify`, tổng chi phí cho bước này là O(n).
- Trích xuất phần tử và xây dựng lại Heap: Mỗi phần tử được trích xuất và `heapify` được gọi O(log n) lần cho n phần tử. Do đó, thời gian chạy của bước này là O(n log n).

Tổng thời gian chạy của thuật toán Heap-Sort là O(n log n) cho cả hai bước.

Ứng dụng:

- Sắp xếp: Heap-Sort là một thuật toán sắp xếp không ổn định với thời gian chạy tối ưu.
- Tạo Heap: Được sử dụng trong các thuật toán cần cấu trúc Heap, như thuật toán Dijkstra cho tìm kiếm đường đi ngắn nhất.
- Lên lịch: Dùng trong các hệ thống thời gian thực để quản lý tác vụ dựa trên độ ưu tiên.

Chương VII. Các thuật toán tìm kiếm

7.1. Thuật toán tìm kiếm tuyến tính (Sequential Search)

Tìm kiếm tuyến tính hay tìm kiếm tuần tự là một phương pháp đơn giản và trực tiếp để tìm kiếm một giá trị trong một danh sách hoặc mảng. Thuật toán này hoạt động bằng cách duyệt qua từng phần tử của danh sách cho đến khi tìm thấy phần tử cần tìm hoặc kết thúc danh sách.

Tìm kiếm tuần tự là một thuật toán tìm kiếm cơ bản và dễ hiểu. Nó được sử dụng khi bạn có một danh sách không được sắp xếp hoặc không thể sắp xếp. Thuật toán này có thể áp dụng cho cả mảng và danh sách liên kết.

Giả mã của thuật toán tìm kiếm tuyến tính:

Function sequentialSearch(array, key)

```
for i from 0 to length(array) - 1
      if array[i] == key
            return i  // Trả về chỉ số của phần tử nếu tìm thấy
return -1  // Trả về -1 nếu không tìm thấy phần tử
```

Giải thích:

- Duyệt qua từng phần tử: Thuật toán bắt đầu từ phần tử đầu tiên trong danh sách và so sánh từng phần tử với giá trị cần tìm.
- So sánh: Nếu phần tử hiện tại trùng với giá trị cần tìm, thuật toán trả về chỉ số của phần tử đó.
- Kết thúc: Nếu không tìm thấy phần tử trong danh sách, thuật toán trả về -1 hoặc một giá trị đại diện cho việc không tìm thấy.

Thời gian chạy:

- Trung bình: O(n), vì trong trường hợp trung bình, giá trị cần tìm có thể nằm ở giữa danh sách.
- Xấu nhất: O(n), nếu giá trị cần tìm nằm ở cuối danh sách hoặc không có trong danh sách.
- Tốt nhất: O(1), nếu giá trị cần tìm là phần tử đầu tiên trong danh sách.

Ứng dụng:

- Danh sách không sắp xếp: Tìm kiếm tuần tự là lựa chọn tốt khi làm việc với danh sách không được sắp xếp hoặc khi không thể sắp xếp.
- Danh sách nhỏ: Đối với danh sách có kích thước nhỏ, tìm kiếm tuần tự có thể hiệu quả và dễ thực hiện.
- Danh sách liên kết: Có thể áp dụng cho danh sách liên kết đơn hoặc đôi, nơi các phần tử không liên tiếp trong bộ nhớ.

7.2. Thuật toán tìm kiếm nhị phân (Binary Search)

Tìm kiếm nhị phân là một thuật toán tìm kiếm hiệu quả cho các danh sách hoặc mảng đã được sắp xếp. Thuật toán này sử dụng phương pháp chia để trị để tìm kiếm một giá trị mục tiêu bằng cách liên tục chia đôi phần tử của mảng cho đến khi tìm thấy giá trị mục tiêu hoặc xác định rằng giá trị đó không có trong mảng.

Tìm kiếm nhị phân hoạt động trên mảng đã được sắp xếp theo thứ tự tăng dần hoặc giảm dần. Thuật toán này so sánh giá trị mục tiêu với phần tử giữa của mảng và loại bỏ một nửa của mảng trong mỗi bước, làm giảm kích thước bài toán một cách nhanh chóng.

Giả mã của thuật toán tìm kiếm nhị phân:

```
Function binarySearch(array, key)
    low = 0
    high = length(array) - 1
    while low <= high
        mid = (low + high) / 2
        if array[mid] == key
            return mid  // Trả về chỉ số của phần tử nếu tìm thấy
        else if array[mid] < key
            low = mid + 1  // Tìm kiếm nửa bên phải
        else
            high = mid - 1  // Tìm kiếm nửa bên trái
    return -1  // Trả về -1 nếu không tìm thấy phần tử
```

Giải thích:

- Khởi tạo: Bắt đầu bằng cách thiết lập hai chỉ số `low` và `high` cho mảng, đại diện cho các giới hạn của mảng cần tìm kiếm.
- Lặp và so sánh: Tính toán chỉ số giữa `mid` và so sánh phần tử ở chỉ số giữa với giá trị mục tiêu `key`.
  - Nếu phần tử giữa bằng `key`, trả về chỉ số giữa.
  - Nếu phần tử giữa nhỏ hơn `key`, điều chỉnh chỉ số `low` để tìm kiếm nửa bên phải.
  - Nếu phần tử giữa lớn hơn `key`, điều chỉnh chỉ số `high` để tìm kiếm nửa bên trái.
- Kết thúc: Lặp lại cho đến khi `low` vượt qua `high`. Nếu không tìm thấy giá trị mục tiêu, trả về `-1`.

Thời gian chạy:

- Trung bình: O(log n), vì mỗi lần so sánh giảm số lượng phần tử cần kiểm tra xuống một nửa.
- Xấu nhất: O(log n), do tính chất phân chia của thuật toán.
- Tốt nhất: O(1), nếu giá trị mục tiêu là phần tử giữa trong lần kiểm tra đầu tiên.

Ứng dụng:

- Mảng đã sắp xếp: Tìm kiếm nhị phân là lý tưởng cho các mảng đã được sắp xếp theo thứ tự tăng dần hoặc giảm dần.

- Tìm kiếm nhanh: Được sử dụng khi cần tìm kiếm nhanh trong các danh sách lớn mà không thể sử dụng tìm kiếm tuần tự.
- Cấu trúc dữ liệu sắp xếp: Thường được sử dụng trong các cấu trúc dữ liệu như cây tìm kiếm nhị phân (Binary Search Tree) và bảng băm có duy trì thứ tự.

7.3. Tìm kiếm trên bảng băm (Hash Table)

7.3.1. Hàm băm (Hash Function) và bảng băm (Hash Table)

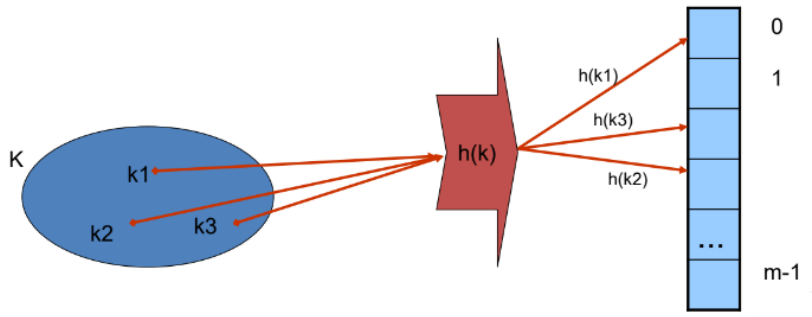
Xây dựng hàm băm:

Cấu trúc hàm băm: Hàm băm được ký hiệu là h và có dạng: h: K→{0, 1, …, m−1}

- h là hàm băm.
- K là tập giá trị khóa.
- {0, 1, …, m−1} là bảng địa chỉ, trong đó m là kích thước của bảng.

Yêu cầu khi xây dựng hàm băm:

- Hàm phải phân phối đều các địa chỉ trong bảng.
- Hàm băm phải được tính toán đơn giản để hiệu quả và tốc độ cao.



Hình ảnh minh họa hàm băm

Một số phương pháp xây dựng hàm băm:

Phương pháp chia:

- Công thức: h(k) = k % m
- Để tính địa chỉ, lấy giá trị khóa chia cho kích thước của bảng, và địa chỉ là phần dư của phép chia.
- Yêu cầu: Hàm h phải phân phối các đối tượng đều trên bảng. Thông thường, chọn m là số nguyên tố nhỏ hơn gần với các giá trị như 10, 100, 1000, ...

Ví dụ:

- Nạp các giá trị khóa: 89, 2, 13, 43, 65, 23, 54, 49, 32
- Sử dụng hàm băm: h(k) = k % 7

Phương pháp nhân:

- Giá trị khóa được nhân với một hệ số, sau đó lấy các chữ số ở giữa của kết quả để làm địa chỉ băm.

Ví dụ: Nếu k = 1234, ta nhân với hệ số và lấy các chữ số ở giữa của kết quả như 5678. Địa chỉ băm có thể là 567 hoặc 678.

Phương pháp phân đoạn: Giá trị khóa được chia thành nhiều đoạn bằng nhau. Sử dụng hai kỹ thuật phân đoạn:

- Tách (Segmentation): Tách các đoạn ra, sắp xếp chúng thành hàng, căn lề trái hoặc phải.
- Gấp (Folding): Gấp các đoạn lại theo đường biên tương tự như gấp giấy, các giá trị trong cùng một đoạn được cộng lại.

Bảng băm (Hash table):

Khái niệm: Bảng băm là một cấu trúc dữ liệu dựa trên mảng dùng để lưu trữ các phần tử dưới dạng các cặp Khóa - Giá trị (key - value).

Các thành phần cấu thành bảng băm:

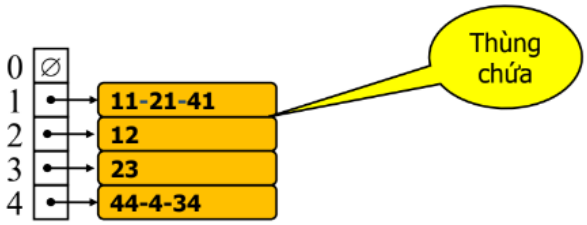
- Mảng chứa: Một mảng, trong đó mỗi phần tử của mảng có thể chứa một danh sách (hoặc một cấu trúc dữ liệu khác) để quản lý các phần tử có cùng chỉ mục.
- Danh sách phần tử: Mỗi phần tử trong mảng là một danh sách (hoặc danh sách liên kết) lưu trữ các phần tử có khóa ánh xạ đến cùng một chỉ mục do hàm băm cung cấp.
- Hàm băm (Hash Function).

Ví dụ:

Giả sử bạn có bảng băm với kích thước m = 5 và hàm băm h(k) = k % 5. Các giá trị khóa và chỉ mục của chúng sẽ như sau:

- Khóa 11: h(11) = 11 % 5 = 1
- Khóa 21: h(21) = 21 % 5 = 1
- Khóa 44: h(44) = 44 % 5 = 4
- Khóa 23: h(23) = 23 % 5 = 3

- Khóa 41: h(41) = 41 % 5 = 1
- Khóa 4: h(4) = 4 % 5 = 4
- Khóa 34: h(34) = 34 % 5 = 4



Hình ảnh minh họa bảng băm sau khi chèn

7.3.1. Tìm kiếm trong bảng băm

Tìm kiếm trong bảng băm là một thuật toán tìm kiếm hiệu quả cho các cấu trúc dữ liệu bảng băm, nơi dữ liệu được lưu trữ dưới dạng các cặp Khóa - Giá trị (key - value). Thuật toán tìm kiếm sử dụng hàm băm để ánh xạ khóa đến chỉ mục của mảng và truy xuất giá trị mục tiêu một cách nhanh chóng.

Giả mã của thuật toán tìm kiếm trong bảng băm:

Function hashTableSearch(hashTable, key)

```
index = hashFunction(key)  // Tính toán chỉ mục từ hàm băm
if hashTable[index] is not empty
      // Duyệt qua danh sách liên kết hoặc kiểm tra các ô tiếp theo
      for each element in hashTable[index]
            if element.key == key
                  // Trả về giá trị tương ứng với khóa nếu tìm thấy
                  return element.value
return null  // Trả về null nếu không tìm thấy phần tử
```

Giải thích:

- Khởi tạo: Tính toán chỉ mục của mảng bằng cách sử dụng hàm băm `hashFunction` trên khóa `key`.
- Truy cập danh sách hoặc ô:
  - Nếu tại chỉ mục tính toán không rỗng, duyệt qua danh sách liên kết (chaining) hoặc kiểm tra các ô tiếp theo (open addressing) để tìm phần tử với khóa `key`.

- Nếu phần tử với khóa `key` được tìm thấy, trả về giá trị tương ứng.
- Nếu không tìm thấy phần tử, trả về `null`.

Thời gian chạy:

- Trung bình: O(1 + α): Trong trường hợp tốt nhất và trung bình, thời gian tìm kiếm gần O(1) khi không có xung đột hoặc xung đột được xử lý tốt. `α` là hệ số tải, số phần tử trên mỗi ô của bảng.
- Xấu nhất: O(n): Khi tất cả các phần tử bị ánh xạ đến cùng một chỉ mục trong trường hợp xấu nhất (chaining), hoặc khi bảng gần đầy và phải kiểm tra nhiều ô liên tiếp (open addressing), thời gian tìm kiếm có thể là O(n), với `n` là tổng số phần tử trong bảng.
- Tốt nhất: O(1): Nếu phần tử với khóa `key` nằm ở chỉ mục được tính toán và không có xung đột, thời gian tìm kiếm có thể là O(1).

Ứng dụng:

- Lưu trữ dữ liệu nhanh: Bảng băm là lý tưởng cho việc lưu trữ và truy xuất dữ liệu nhanh chóng khi khóa có thể được ánh xạ một cách hiệu quả.
- Tìm kiếm nhanh: Được sử dụng khi cần thực hiện tìm kiếm nhanh trong các danh sách lớn mà không cần duyệt tuần tự.
- Xử lý xung đột: Có thể sử dụng các phương pháp như chaining hoặc open addressing để xử lý xung đột và duy trì hiệu suất tìm kiếm.

Chương VIII. Các cấu trúc dữ liệu nâng cao

8.1. Cấu trúc dữ liệu ánh xạ (Map)

8.1.1. Tìm hiểu cấu trúc dữ liệu Map

Cấu trúc dữ liệu ánh xạ (Map) là một container trong C++ cho phép lưu trữ các cặp Khóa-Giá trị (key - value). Mỗi khóa là duy nhất và ánh xạ đến một giá trị cụ thể. C++ cung cấp hai loại map chính: `std::map` và `std::unordered_map`.

- `std::map`: Được triển khai dưới dạng cây nhị phân cân bằng (thường là cây đỏ - đen), các phần tử trong `std::map` được lưu trữ theo thứ tự của khóa.
- `std::unordered_map`: Sử dụng bảng băm để lưu trữ các phần tử, không đảm bảo thứ tự của các phần tử và có hiệu suất cao cho các thao tác tìm kiếm, chèn, và xóa.

Khai báo:

```
map<KeyType, ValueType> map;

unordered_map<KeyType, ValueType> map;
```

Các phương thức chính:

- `insert(pair)`: Thêm một cặp Khóa - Giá trị vào map.
- `erase(key)`: Xóa phần tử với khóa cụ thể.
- `find(key)`: Tìm kiếm phần tử với khóa cụ thể và trả về iterator đến phần tử nếu tìm thấy.
- `operator[]`: Truy cập hoặc thêm phần tử với khóa cụ thể.
- `size()`: Trả về số lượng phần tử trong map.
- `empty()`: Kiểm tra xem map có rỗng hay không.

Ví dụ về map:

```
#include <iostream>
#include <map>
using namespace std;
int main() {
	map<string, int> map;
	map["Alice"] = 30;
	map["Bob"] = 25;
	map["Charlie"] = 35;
	for(auto it: map){
		cout << it.first << ": " << it.second << endl;
	}
	map.erase("Bob");
	if (map.find("Charlie") != map.end()) {
		cout << "Charlie exists in the map." << endl;
	}
	cout << "Size of map: " << map.size() << endl;
}
//Kết quả:
//Alice: 30
//Bob: 25
//Charlie: 35
//Charlie exists in the map.
//Size of map: 2
```

Ví dụ về unordered_map:

```
#include <iostream>
#include <unordered_map>
using namespace std;
int main() {
unordered_map<string, int> map;
map["Alice"] = 30;
map["Bob"] = 25;
map["Charlie"] = 35;
for(auto it: map){
      cout << it.first << ": " << it.second << endl;
}
map.erase("Bob");
if (map.find("Charlie") != map.end()) {
cout << "Charlie exists in the map." << endl;
}
cout << "Size of map: " << map.size() << endl;
}
//Kết quả:
//Charlie: 35
//Bob: 25
//Alice: 30
//Charlie exists in the map.
//Size of map: 2
```

Thời gian chạy:

- `map`:
  - Trung bình: O(log n) cho các thao tác thêm, xóa, và tìm kiếm.
  - Xấu nhất: O(log n) do cấu trúc cây cân bằng.
  - Tốt nhất: O(log n) nếu cây cân bằng tốt.
- `unordered_map`:
  - Trung bình: O(1) cho các thao tác thêm, xóa, và tìm kiếm, nếu hàm băm phân phối đều.

- Xấu nhất: O(n) trong trường hợp xung đột băm cao hoặc phân phối không đều.
- Tốt nhất: O(1) nếu không có xung đột và hàm băm hoạt động tốt.

8.1.2. Ứng dụng của cấu trúc dữ liệu Map

Lưu trữ và tra cứu dữ liệu

Mục đích: `map` và `unordered_map` rất hữu ích khi cần lưu trữ dữ liệu các cặp Khóa - Giá trị. Chúng cho phép lưu trữ các phần tử mà không yêu cầu thứ tự cụ thể.

Công dụng:

- `map`: Duy trì các phần tử được sắp xếp theo thứ tự của khóa, phù hợp cho các ứng dụng yêu cầu dữ liệu được sắp xếp và tìm kiếm hiệu quả.
- `unordered_map`: Lưu trữ dữ liệu mà không yêu cầu thứ tự của khóa, giúp tra cứu nhanh chóng hơn trong các ứng dụng mà thứ tự không quan trọng.

Tìm kiếm nhanh

Mục đích: Cả hai cấu trúc dữ liệu đều hỗ trợ tìm kiếm nhanh các phần tử dựa trên khóa.

Công dụng:

- `map`: Tìm kiếm với độ phức tạp thời gian là O(log n) trong trường hợp xấu nhất, nhờ vào cấu trúc cây cân bằng.
- `unordered_map`: Tìm kiếm với độ phức tạp thời gian trung bình là O(1), nhờ vào bảng băm, mặc dù độ phức tạp có thể lên tới O(n) trong trường hợp xấu nhất (nếu có nhiều xung đột trong bảng băm).

Cấu trúc dữ liệu chính trong các ứng dụng:

- Từ điển (Dictionary): Lưu trữ các cặp từ và định nghĩa hoặc các dữ liệu tương tự. `map` có thể sử dụng để duy trì thứ tự từ, trong khi `unordered_map` sẽ cho phép truy xuất nhanh hơn nhưng không bảo đảm thứ tự.
- Bộ nhớ đệm (Cache): Lưu trữ tạm thời các dữ liệu để tăng tốc độ truy cập cho các thao tác lặp lại. `unordered_map` là lựa chọn phổ biến vì hiệu suất tìm kiếm nhanh hơn.
- Quản lý phiên (Session Management): Lưu trữ thông tin phiên người dùng hoặc các trạng thái tạm thời. `map` và `unordered_map` đều có thể được sử dụng để lưu trữ các thông tin phiên với khóa là ID phiên và giá trị là thông tin liên quan đến phiên.
- Lưu trữ cấu hình (Configuration Storage): Quản lý các thiết lập và cấu hình của ứng dụng. `map` có thể được sử dụng nếu cần duy trì thứ tự của các thiết lập, trong khi

`unordered_map` có thể được chọn cho hiệu suất truy xuất cao hơn mà không cần duy trì thứ tự.

Tóm lại:

- `map`: Thích hợp cho các tình huống yêu cầu sắp xếp theo khóa và tìm kiếm nhanh với độ phức tạp thời gian là O(log n). Thích hợp cho các ứng dụng như từ điển, quản lý cấu hình, và quản lý phiên.
- `unordered_map`: Tối ưu cho các tình huống yêu cầu tìm kiếm nhanh với độ phức tạp thời gian trung bình là O(1). Thích hợp cho bộ nhớ đệm, lưu trữ dữ liệu không yêu cầu thứ tự, và các ứng dụng cần truy xuất dữ liệu nhanh chóng.

8.2. Cấu trúc dữ liệu tập hợp (Set)

8.2.1. Tìm hiểu cấu trúc dữ liệu Set

Tập hợp (Set) là một cấu trúc dữ liệu lưu trữ một tập hợp các phần tử duy nhất mà không cần duy trì thứ tự của các phần tử. Tập hợp hỗ trợ các phép toán như thêm, xóa và kiểm tra sự tồn tại của phần tử. C++ cũng cung cấp 2 loại set là `std::set` và `std::unordered_set`

Khai báo:

- `set`: Tập hợp sắp xếp theo thứ tự các phần tử, sử dụng cây cân bằng (như cây đỏ đen).

```
set<int> mySet;
```

- `unordered_set`: Tập hợp không duy trì thứ tự của các phần tử, sử dụng bảng băm để cung cấp hiệu suất tìm kiếm trung bình là O(1).

```
unordered_set<int> myUnorderedSet;
```

Phương thức chính

- `insert(value)`: Thêm một phần tử vào tập hợp. Nếu phần tử đã tồn tại, không thực hiện thêm.
- `erase(value)`: Xóa phần tử khỏi tập hợp.
- `find(value)`:Tìm kiếm phần tử và trả về iterator đến phần tử nếu tồn tại, hoặc end() nếu không tồn tại.
- `count(value)`: Trả về số lượng phần tử bằng giá trị value (thường là 0 hoặc 1).
- `size()`: Trả về số lượng phần tử trong tập hợp.
- `empty()`: Kiểm tra xem tập hợp có rỗng không.

Ví dụ về set:

```
#include <iostream>
#include <set>
using namespace std;
int main() {
	set<int> numbers;
	numbers.insert(10);
	numbers.insert(20);
	numbers.insert(30);
	numbers.erase(20);
	if (numbers.find(10) != numbers.end()) {
		cout << "10 is in the set." << endl;
	}
	for (int& num : numbers) {
		cout << num << " ";
	}
}
```

Ví dụ về unordered_set:

```
#include <iostream>
#include <unordered_set>
using namespace std;
int main() {
	unordered_set<int> numbers;
	numbers.insert(10);
	numbers.insert(20);
	numbers.insert(30);
	numbers.erase(20);
	if (numbers.find(10) != numbers.end()) {
		cout << "10 is in the set." << endl;
	}
	for (int& num : numbers) {
		cout << num << " ";
	}
}
```

Thời gian chạy:

- `set`:
  - Thêm, Xóa, Tìm kiếm: O(log n) - Do cấu trúc cây cân bằng.
  - Duyệt qua phần tử: O(n).

- `unordered_set`:
  - Thêm, Xóa, Tìm kiếm: Trung bình O(1), trường hợp xấu nhất O(n) - Do cấu trúc bảng băm.
  - Duyệt qua phần tử: O(n).

8.2.2. Ứng dụng của cấu trúc dữ liệu Set

Lưu trữ các phần tử duy nhất:

Mục đích: `set` và `unordered_set` được thiết kế để lưu trữ các phần tử duy nhất. Khi bạn cần lưu trữ một tập hợp các phần tử mà không yêu cầu thứ tự cụ thể và đảm bảo rằng không có phần tử nào bị trùng lặp, các cấu trúc dữ liệu này là lựa chọn lý tưởng.

Chi tiết:

- `set`: Duy trì các phần tử trong một thứ tự đã được sắp xếp, cho phép bạn tra cứu, thêm, và xóa các phần tử một cách hiệu quả, đồng thời luôn duy trì thứ tự của các phần tử.
- `unordered_set`: Không duy trì thứ tự của các phần tử và sử dụng bảng băm để tối ưu hóa tốc độ truy cập, thêm và xóa phần tử. Điều này giúp giảm thời gian tìm kiếm xuống mức tối ưu, đặc biệt khi làm việc với các tập hợp lớn.

Tìm kiếm nhanh:

Mục đích: Cả `set` và `unordered_set` hỗ trợ tìm kiếm nhanh chóng để kiểm tra sự tồn tại của một phần tử trong tập hợp. Điều này rất hữu ích khi bạn cần thực hiện các phép toán kiểm tra và tra cứu hiệu quả trong các tập dữ liệu lớn.

Chi tiết:

- `set`: Sử dụng cấu trúc cây cân bằng để đảm bảo rằng việc tìm kiếm phần tử có thể được thực hiện trong thời gian O(log n). Thời gian này là ổn định ngay cả khi dữ liệu được sắp xếp theo một thứ tự cụ thể.
- `unordered_set`: Sử dụng bảng băm, cho phép thời gian tìm kiếm trung bình là O(1). Điều này làm cho việc kiểm tra sự tồn tại của phần tử rất nhanh, nhưng độ phức tạp thời gian có thể lên tới O(n) trong trường hợp có nhiều xung đột trong bảng băm.

Xử lý dữ liệu không trùng lặp:

Mục đích: Trong nhiều ứng dụng, cần loại bỏ các phần tử trùng lặp để chỉ giữ lại các phần tử duy nhất. `set` và `unordered_set` là công cụ hữu ích để đạt được điều này, giúp giảm bớt các phần tử không cần thiết và duy trì tính duy nhất của dữ liệu.

Chi tiết:

- `set`: Duy trì thứ tự các phần tử và tự động loại bỏ các phần tử trùng lặp. Ví dụ, khi lưu trữ ID người dùng hoặc mã sản phẩm, set có thể giúp đảm bảo rằng không có phần tử trùng lặp và các phần tử luôn được sắp xếp.
- `unordered_set`: Loại bỏ các phần tử trùng lặp nhanh chóng mà không quan tâm đến thứ tự của các phần tử. Điều này rất hữu ích trong các ứng dụng cần xử lý các tập dữ liệu lớn và không yêu cầu duy trì thứ tự.

Thuật toán và phân tích:

Mục đích: `set` và `unordered_set` thường được sử dụng trong các thuật toán và phân tích để thực hiện các phép toán trên tập hợp dữ liệu, như hợp nhất, giao nhau và hiệu chỉnh tập hợp.

Chi tiết:

- `set`: Cung cấp các phép toán tập hợp như hợp nhất (union), giao nhau (intersection), và hiệu chỉnh (difference) một cách dễ dàng, nhờ vào khả năng duy trì thứ tự của các phần tử. Điều này giúp dễ dàng thực hiện các phép toán trên các tập hợp dữ liệu.
- `unordered_set`: Cung cấp khả năng thực hiện các phép toán tập hợp với hiệu suất cao nhờ vào tốc độ tìm kiếm nhanh. Mặc dù không duy trì thứ tự của các phần tử, nhưng các phép toán như hợp nhất và giao nhau vẫn có thể được thực hiện nhanh chóng.

Tóm lại:

- `set` là lựa chọn lý tưởng cho các tình huống yêu cầu lưu trữ các phần tử duy nhất với thứ tự được duy trì và thực hiện các phép toán tập hợp với độ phức tạp thời gian hợp lý.
- `unordered_set` là lựa chọn tốt cho các tình huống yêu cầu hiệu suất cao trong tìm kiếm và xử lý dữ liệu lớn mà không cần duy trì thứ tự của các phần tử.

8.3. Cấu trúc dữ liệu hàng đợi ưu tiên (Priority Queue)

8.3.1. Tìm hiểu về cấu trúc dữ liệu Priority Queue

Hàng đợi ưu tiên (Priority Queue) là một cấu trúc dữ liệu cho phép quản lý các phần tử với mức độ ưu tiên. Mỗi phần tử trong hàng đợi có một mức độ ưu tiên gắn kèm, và phần tử với mức độ ưu tiên cao nhất sẽ được xử lý trước. Khác với hàng đợi thông thường (FIFO), hàng đợi ưu tiên không đảm bảo thứ tự vào ra của các phần tử, mà đảm bảo rằng phần tử với ưu tiên cao nhất sẽ được phục vụ trước.

Khai báo:

```
std::priority_queue<int> pq;

std::priority_queue<int, std::vector<int>, std::greater<int>> pq_min;
```

Phương thức chính:

- `push(value)`: Thêm một phần tử vào hàng đợi với mức độ ưu tiên tương ứng.
- `pop()`: Xóa phần tử với mức độ ưu tiên cao nhất khỏi hàng đợi.
- `top()`: Trả về phần tử với mức độ ưu tiên cao nhất mà không xóa nó khỏi hàng đợi.
- `empty()`: Kiểm tra xem hàng đợi có rỗng không.
- `size()`: Trả về số lượng phần tử hiện tại trong hàng đợi.

Ví dụ về 2 loại hàng đợi ưu tiên:

```
#include <iostream>

#include <queue>

using namespace std;

int main() {

	priority_queue<int> pq;

	pq.push(10);

	pq.push(30);

	pq.push(20);

	cout << "Top element: " << pq.top() << endl;

	pq.pop();

	cout << "New top element: " << pq.top() << endl;

	priority_queue<int, vector<int>, greater<int>> pq_min;

	pq_min.push(10);

	pq_min.push(30);

	pq_min.push(20);

	cout << "Top element: " << pq_min.top() << endl;

	pq_min.pop();

	cout << "New top element: " << pq_min.top() << endl;

}
```

Thời gian chạy:

- Thêm phần tử: O(log n) - Thao tác này yêu cầu tổ chức lại cấu trúc cây để đảm bảo thứ tự ưu tiên.
- Xóa phần tử với ưu tiên cao nhất: O(log n) - Xóa phần tử và sau đó tái tổ chức cấu trúc cây.
- Lấy phần tử với ưu tiên cao nhất: O(1) - Không cần tái tổ chức cấu trúc, chỉ cần truy xuất phần tử đầu tiên.

8.3.2. Ứng dụng của cấu trúc dữ liệu Priority Queue

Lên lịch (Scheduling):

Mục đích: Trong các hệ điều hành và hệ thống phân tán, hàng đợi ưu tiên đóng vai trò quan trọng trong việc quản lý và lên lịch các tác vụ hoặc tiến trình. Hệ điều hành sử dụng hàng đợi ưu tiên để đảm bảo rằng các tác vụ quan trọng hoặc khẩn cấp được thực hiện trước, đồng thời duy trì hiệu suất và phản hồi của hệ thống.

Chi tiết:

- Quản lý Tác vụ: Các tác vụ có độ ưu tiên cao (như các nhiệm vụ hệ thống quan trọng) được đưa vào hàng đợi ưu tiên để được xử lý trước. Các tác vụ này có thể bao gồm các hoạt động hệ thống, xử lý sự kiện, và các nhiệm vụ cần hoàn thành gấp.
- Lên lịch Tiến trình: Các tiến trình trong hệ điều hành được phân loại theo mức độ ưu tiên. Hàng đợi ưu tiên giúp hệ điều hành quyết định thứ tự xử lý các tiến trình để tối ưu hóa hiệu suất hệ thống và đảm bảo các tiến trình quan trọng được thực thi kịp thời.

Thuật toán Dijkstra:

Mục đích: Hàng đợi ưu tiên được sử dụng trong thuật toán Dijkstra để tìm đường đi ngắn nhất từ một điểm đến tất cả các điểm còn lại trong đồ thị. Điều này đặc biệt hữu ích trong các ứng dụng liên quan đến mạng lưới và đường đi, như định tuyến mạng và lập bản đồ.

Chi tiết:

- Tìm đường ngắn nhất: Thuật toán Dijkstra sử dụng hàng đợi ưu tiên để xử lý các đỉnh trong đồ thị dựa trên khoảng cách ngắn nhất từ điểm bắt đầu. Hàng đợi ưu tiên đảm bảo rằng đỉnh với khoảng cách ngắn nhất được xử lý trước, giúp cập nhật khoảng cách và tìm đường đi tối ưu một cách hiệu quả.
- Tối ưu hóa hiệu suất: Việc sử dụng hàng đợi ưu tiên giúp giảm độ phức tạp thời gian của thuật toán, từ O(V^2) xuống O(E log V) (với E là số cạnh và V là số đỉnh), làm cho thuật toán trở nên khả thi cho các đồ thị lớn.

Giải thuật Huffman coding:

Mục đích: Trong quá trình nén dữ liệu, hàng đợi ưu tiên được sử dụng để xây dựng cây Huffman, một cấu trúc dữ liệu cần thiết để mã hóa dữ liệu hiệu quả.

Chi tiết:

- Xây dựng cây Huffman: Giải thuật Huffman sử dụng hàng đợi ưu tiên để kết hợp các nút có tần suất thấp nhất thành các nút mới cho đến khi chỉ còn một nút. Điều này giúp xây dựng cây Huffman, nơi các ký tự xuất hiện thường xuyên có mã ngắn hơn và các ký tự ít xuất hiện có mã dài hơn.
- Tối ưu hóa nén dữ liệu: Sử dụng hàng đợi ưu tiên trong giải thuật Huffman giúp tối ưu hóa quá trình nén dữ liệu bằng cách đảm bảo rằng các phần tử với tần suất xuất hiện thấp được xử lý một cách hiệu quả, dẫn đến việc nén dữ liệu hiệu quả hơn.

Quản lý sự kiện:

Mục đích: Trong các hệ thống thời gian thực, hàng đợi ưu tiên giúp quản lý các sự kiện dựa trên mức độ ưu tiên. Điều này đảm bảo rằng các sự kiện quan trọng và khẩn cấp được xử lý trước, giúp hệ thống phản hồi kịp thời và hiệu quả.

Chi tiết:

- Xử lý sự kiện: Hàng đợi ưu tiên cho phép các hệ thống thời gian thực xử lý các sự kiện quan trọng (như thông báo lỗi, cảnh báo hệ thống) trước các sự kiện ít quan trọng hơn. Điều này giúp duy trì độ chính xác và hiệu suất của hệ thống.
- Quản lý tài nguyên: Hàng đợi ưu tiên có thể được sử dụng để quản lý tài nguyên và điều phối các tác vụ dựa trên mức độ ưu tiên, giúp hệ thống hoạt động hiệu quả hơn trong các tình huống yêu cầu xử lý đồng thời nhiều sự kiện.

Tìm kiếm và phân tích dữ liệu:

Mục đích: Hàng đợi ưu tiên hỗ trợ trong việc tìm kiếm và phân tích dữ liệu với các mức độ ưu tiên khác nhau. Các ứng dụng trong phân tích dữ liệu, học máy và hệ thống khuyến nghị thường sử dụng hàng đợi ưu tiên để quản lý và xử lý dữ liệu.

Chi tiết:

- Phân tích dữ liệu: Trong các ứng dụng phân tích dữ liệu, hàng đợi ưu tiên có thể được sử dụng để xử lý các phần tử dữ liệu dựa trên độ quan trọng hoặc tần suất. Ví dụ, trong phân tích nhật ký hệ thống, hàng đợi ưu tiên giúp xử lý các sự kiện lỗi hoặc cảnh báo quan trọng trước các sự kiện ít nghiêm trọng hơn.

- Học máy: Trong các ứng dụng học máy, hàng đợi ưu tiên hỗ trợ trong việc xử lý các mẫu dữ liệu hoặc dự đoán dựa trên độ ưu tiên. Điều này giúp cải thiện hiệu suất của các mô hình học máy bằng cách đảm bảo các dữ liệu quan trọng được xử lý kịp thời.

Tóm lại:

Hàng đợi ưu tiên là một công cụ mạnh mẽ cho việc quản lý và xử lý các phần tử dựa trên mức độ ưu tiên, hỗ trợ trong nhiều lĩnh vực từ lập lịch và tối ưu hóa thuật toán đến phân tích dữ liệu và quản lý sự kiện. Các ứng dụng này cho thấy khả năng linh hoạt và hiệu quả của hàng đợi ưu tiên trong việc giải quyết các vấn đề phức tạp trong các hệ thống và thuật toán.