HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG
KHOA CÔNG NGHỆ THÔNG TIN 1

BÀI GIẢNG
CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

Biên soạn : TS. NGUYỄN DUY PHƯƠNG
THS. NGUYỄN MẠNH SƠN

HÀ NỘI, 12/2020

LỜI NÓI ĐẦU

Cấu trúc dữ liệu là phương pháp biểu diễn các đối tượng ở thế giới thực thành dữ liệu được tổ chức, lưu trữ trên máy tính để phục vụ quá trình xử lý và khai thác thông tin một cách hiệu quả. Thuật toán được hiểu là phương pháp xử lý thông tin hay dữ liệu được biểu diễn bởi các cấu trúc dữ liệu một cách nhanh nhất. Sự kết hợp giữa cấu trúc dữ liệu và thuật toán trên cấu trúc dữ liệu đem lại hiệu quả cao trong xây dựng ứng dụng. Chính vì lý do này, Cấu trúc dữ liệu và giải thuật được xem là môn học bắt buộc mang tính chất kinh điển của các ngành Công nghệ thông tin và Điện tử Viễn thông. Tài liệu giảng dạy môn Cấu trúc dữ liệu và giải thuật được xây dựng dựa trên nội dung chương trình khung đã được Học Viện Công Nghệ Bưu Chính Viễn Thông ban hành.

Tài liệu được trình bày thành 6 chương. Trong đó, Chương 1 trình bày khái niệm và định nghĩa cơ bản về cấu trúc dữ liệu và giải thuật. Chương 2 trình bày một số mô hình thuật toán kinh điển ứng dụng trong Công nghệ Thông tin. Chương 3 trình bày về các kỹ thuật sắp xếp và tìm kiếm. Chương 4 trình bày về các kiểu dữ liệu tuyến tính (ngăn xếp, hàng đợi và danh sách liên kết). Chương 5, 6 trình bày về các cấu trúc dữ liệu rời rạc (cây, đồ thị). Đối với với mỗi cấu trúc dữ liệu, tài liệu tập trung trình bày bốn nội dung cơ bản: định nghĩa, biểu diễn, thao tác và ứng dụng của cấu trúc dữ liệu. Ứng với mỗi thuật toán, tài liệu trình bày bốn nội dung cơ bản: biểu diễn, đánh giá, thử nghiệm và cài đặt thuật toán.

Trong mỗi phần của tài liệu, chúng tôi cố gắng trình bày ngắn gọn trực tiếp vào bản chất của vấn đề, đồng thời cài đặt các thuật toán bằng ngôn ngữ lập trình C++ nhằm đạt được ba mục tiêu chính cho người học: làm chủ được các phương pháp biểu diễn dữ liệu, nâng cao tư duy phân tích, thiết kế, đánh giá thuật toán và kỹ thuật lập trình bằng thuật toán. Mặc dù đã rất cẩn trọng trong quá trình biên soạn, tuy nhiên tài liệu không tránh khỏi những thiếu sót và hạn chế. Chúng tôi rất mong được sự góp ý quý báu của tất cả bạn đọc.

Hà nội, tháng 12 năm 2020

MỤC LỤC

CHƯƠNG 1. GIỚI THIỆU CHUNG

Mục tiêu chính của chương này là giải thích rõ tầm quan trọng của việc phân tích thuật toán cùng với mối liên hệ và sự ảnh hưởng qua lại giữa dữ liệu và thuật toán. Để thực hiện được điều này, chúng ta sẽ bắt đầu từ những khái niệm và định nghĩa cơ bản về dữ liệu, thuật toán sau đó mở rộng sang những vấn đề quan trọng hơn độ phức tạp thuật toán, độ phức tạp chương trình. Cuối cùng, chúng ta sẽ xem xét đến qui trình giải quyết một vấn đề trong khoa học máy tính bằng thuật toán.

1.1. Kiểu và cấu trúc dữ liệu

Trước khi định nghĩa chính xác các khái niệm về kiểu dữ liệu (*data types*), biến (*variables*) ta xem xét lại với những gì ta đã từng biết trước đây trong toán học. Chẳng hạn khi ta giải phương trình:

$$x^2 - 2y - 2 = 1$$

Tiếp cận bằng toán học ta nói nghiệm của phương trình trên là tập các cặp (*x*, *y*) sao cho $x^2 - 2y - 2 = 1$. Ví dụ cặp (1, -1) là một nghiệm của phương trình. Tiếp cận bằng tin học ta sẽ thấy phương trình trên có hai tên là *x* và *y*. Nếu *x* và *y* có giá trị tương ứng là *1, -1* thì đó là một nghiệm của phương trình. Trong khoa học máy tính cũng gọi *x* và *y* là hai biến và các bộ giá trị của *x* và *y* được gọi là dữ liệu.

Hai biến *x* và *y* có thể nhận giá trị trong các miền khác nhau. Để giải được phương trình ta cần phải xác định được miền giá trị của hai biến *x* và *y*. Ví dụ, *x, y* xác định trong miền các số nguyên (10, 20, 30,..), số thực (0.23, 0.55,…) hoặc (0, 1). Để xác định miền giá trị của các biến, trong khoa học máy tính sử dụng một từ khóa đại diện cho một tập các giá trị còn được gọi là một kiểu dữ liệu (*a data type*). Ta sẽ bắt đầu bằng cách tổng quát hóa những khái niệm cơ bản này theo cách tiếp cận của khoa học máy tính.

1.1.1. Kiểu dữ liệu

Kiểu dữ liệu (*a data type*) là một tên hay từ khóa dùng để chỉ tập các đối tượng dữ liệu cùng các phép toán trên nó. Ví dụ trong C++, từ khóa *int* dùng để chỉ tập các số nguyên có độ lớn biểu diễn bằng 2byte (tùy thuộc vào các complier) cùng với các phép toán số học, các phép toán so sánh, các phép toán cấp bít, các phép toán dịch chuyển bit. Từ khóa *float* dùng để chỉ tập các số thực có độ chính xác đơn có độ lớn được biểu diễn bằng 4byte (tùy thuộc vào các complier) cùng với các phép toán số học, các phép toán so sánh. Không có phép lấy phần dư, các phép toán thao tác cấp bít với kiểu dữ liệu *float*. Kiểu dữ liệu được chia thành hai loại kiểu dữ liệu cơ bản hay còn gọi là kiểu dữ liệu nguyên thủy và các kiểu dữ liệu do người dùng định nghĩa.

Kiểu dữ liệu nguyên thủy (*primitve data types*) các kiểu dữ liệu được định nghĩa bởi hệ thống (*system defined data type*) được gọi là các kiểu dữ liệu nguyên thủy. Thông thường, các ngôn ngữ lập trình cung cấp ba kiểu dữ liệu nguyên thủy đó là ký tự (*character*), số (*numberic*), và kiểu logic (*bool*). Kiểu dữ liệu ký tự được chia thành hai loại ký tự ASCII (*char*) và ký tự unicode (*wchar_t*). Kiểu dữ liệu số cũng được chia thành hai loại: số kiểu số nguyên (*integer*) và kiểu số thực (*real*). Kiểu số nguyên được chia thành ba loại: số nguyên nhỏ (*int*), số nguyên lớn (*long*), số nguyên rất lớn (*long long*). Kiểu số thực được chia làm hai loại: số thực có độ chính xác đơn (*float*) và số thực có độ chính xác kép (*double*). Dữ liệu kiểu bool chỉ định nghĩa bộ hai giá trị đúng (*true*) và sai (*false*).

Đương nhiên, hai từ khóa khác nhau đại diện cho hai kiểu dữ liệu khác nhau. Quan sát này chỉ mang tính hình thức vì ta có thể quan sát được bằng mắt. Sự khác biệt bên trong giữa các kiểu dữ liệu là không gian bộ nhớ dùng để biểu diễn kiểu và các phép toán dành cho mỗi biến thuộc kiểu. Không gian nhớ dành cho kiểu phụ thuộc vào compiler của ngôn ngữ lập trình và hệ thống máy tính ta đang sử dụng. Chẳng hạn, kiểu dữ liệu *int* một số compiler dùng 2 byte biểu diễn, một số complier dùng 4 byte để biểu diễn. Các phép toán lấy phần dư (*modulo*), dịch chuyển bít (*bit operations*) định nghĩa cho các số *int, long* nhưng không định nghĩa cho các số *float* và *double*. Để xác định độ lớn của kiểu ta có thể sử dụng hàm *sizeof* (*tên kiểu*). Ví dụ dưới đây dùng để xác định không gian nhớ dành cho kiểu.

```
//Ví dụ 1.1. Xác định kích cỡ bộ nhớ biểu diễn kiểu
#include <iostream>
using namespace std;
int main(void){
        cout<<"KÍCH CỠ KIỂU CƠ BẢN"<<endl;
        cout<<"Kích cỡ kiểu bool:"<<sizeof(bool)<<endl;
        cout<<" Kích cỡ kiểu char:"<<sizeof(char)<<endl;
        cout<<" Kích cỡ kiểu wchar_t:"<<sizeof(wchar_t)<<endl;
        cout<<" Kích cỡ kiểu int:"<<sizeof(int)<<endl;
        cout<<" Kích cỡ kiểu long:"<<sizeof(long)<<endl;
        cout<<" Kích cỡ kiểu long long:"<<sizeof(long long)<<endl;
        cout<<" Kích cỡ kiểu float:"<<sizeof(float)<<endl;
        cout<<" Kích cỡ kiểu double:"<<sizeof(double)<<endl;
}
```

Kiểu dữ liệu do người dùng định nghĩa (*user defined data types*) là các kiểu dữ liệu được do người dùng xây dựng bằng cách tổ hợp các kiểu dữ liệu nguyên thủy theo một nguyên tắc nào đó. Chẳng hạn, kiểu mảng (*array*) là dãy có thứ tự các phần tử (*các biến*) có cùng chung một kiểu dữ liệu được tổ chức liên tục nhau trong bộ nhớ. Kiểu xâu ký tự (string) là một mảng mỗi phần tử là một ký tự và có ký tự kết thúc là '\0'. Như vậy, các kiểu dữ liệu không thuộc các kiểu dữ liệu nguyên thủy như mảng, cấu trúc, file đều được xem là các kiểu dữ liệu do người dùng định nghĩa.

Cấu trúc dữ liệu (*data structure*) là phương pháp biểu diễn các đối tượng ở thế giới thực thành một đối tượng dữ liệu được tổ chức và lưu trữ trong máy tính để có thể sử lý một cách hiệu quả. Theo nghĩa này, mảng (*array*), danh sách liên kết (*linked list*), ngăn xếp (*stack*), hàng đợi (*queue*), cây (*tree*), đồ thị (*graph*)… đều được gọi là các cấu trúc dữ liệu. Dựa vào biểu diễn của các cấu trúc dữ liệu, khoa học máy tính chia các cấu trúc dữ liệu thành hai loại: các cấu trúc dữ liệu tuyến tính (*linear data structures*) và các cấu trúc dữ liệu không tuyến tính (*non-linear data structures*). Một cấu trúc dữ liệu được gọi là tuyến tính nếu việc truy cập các phần tử được thực hiện tuần tự nhưng không nhất thiết được tổ chức liên tục. Điều này có nghĩa, các cấu trúc dữ liệu mảng, danh sách liên kết đơn, danh sách liên kết kép đều là các cấu trúc dữ liệu tuyến tính. Một cấu trúc dữ liệu được gọi là không tuyến tính nếu các phần tử của nó được tổ chức và truy cập không tuần tự. Theo nghĩa này, các cấu trúc dữ liệu cây, graph đều là các cấu trúc dữ liệu không tuyến tính.

Cấu trúc dữ liệu trừu tượng (*Abtract Data types: ADTs*) là phương pháp kết hợp giữa cấu trúc dữ liệu cùng với các phép toán trên dữ liệu cụ thể của cấu trúc dữ liệu. Như vậy, mỗi kiểu dữ liệu ADTs bao gồm hai thành phần:

- *Biểu diễn cấu trúc dữ liệu.*
- *Xây dựng các phép toán trên dữ liệu cụ thể của cấu trúc dữ liệu.*

Theo nghĩa này các cấu trúc dữ liệu danh sách liên kết (*linked list*), ngăn xếp (*stack*), hàng đợi (*queue*), hàng đợi ưu tiên (*priority queue*), cây nhị phân (*binary tree*), đồ thị (*graph*) đều là *ADTs*. Mỗi cấu trúc dữ liệu cụ thể cùng các thao tác trên nó sẽ được trình bày trong những chương tiếp theo của tài liệu.

Đối với mỗi cấu trúc dữ liệu trừu tượng, ta cần quan tâm và nắm bắt được những vấn đề sau:

- **Định nghĩa**: nhằm xác định rõ cấu trúc dữ liệu ADTs ta đang quan tâm đến là gì.
- **Biểu diễn**: nhằm định hình nên cấu trúc dữ liệu ADTs.
- **Thao tác (phép toán)**: những thao tác và phép toán nào được cài đặt trên cấu trúc dữ liệu ADTs.

- **Ứng dụng**: sử dụng cấu trúc dữ liệu ADTs để giải quyết lớp những bài toán nào trong khoa học máy tính.

1.1.2. Biến

Khi một cấu trúc dữ liệu được thiết lập thì thể hiện cụ thể của cấu trúc dữ liệu đó là các phần tử dữ liệu và ta thường gọi là biến. Biến trong tin học và biến trong toán học cũng có một số điểm khác biệt. Biến trong toán học hoàn toàn là một tên hình thức. Ngược lại, biến trong tin học có tên mang tính hình thức, nhưng tên này được lưu trữ tại một vị trí bộ nhớ xác định được gọi là địa chỉ của biến. Dựa vào địa chỉ của biến, giá trị của biến sẽ được lưu trữ tại địa chỉ ô nhớ dành cho biến trong khi xử lý dữ liệu.

Biến (*variables*): là một tên thuộc kiểu. Trong đó, mỗi tên biến dùng để chỉ bộ ba thành phần: tên (*name*), địa chỉ (*address*), giá trị (*value*). Chẳng hạn khi ta có khai báo *int a =10*; khi đó tên biến là *a*, địa chỉ của biến là *&a*, giá trị của biến là *10*. Trong các ngôn ngữ lập trình, biến cũng được chia thành hai loại: biến toàn cục (global variables) và biến cục bộ (local variables).

Một biến được gọi là biến toàn cục nếu nó được khai báo ngoài tất cả các hàm kể cả hàm main(). Không gian nhớ dành cho biến toàn cục sẽ được cấp phát cho biến ngay từ khi bắt đầu thực hiện chương trình cho đến khi kết thúc thực hiện chương trình. Phạm vi sử dụng biến mang tính toàn cục. Người lập trình được phép truy cập và thay đổi giá trị của biến toàn cục ở bất kể vị trí nào trong chương trình. Một biến được gọi là biến cục bộ nếu nó được khai báo trong thân của hàm kể cả hàm main(). Không gian nhớ dành cho biến toàn cục chỉ được cấp phát sau mỗi lần kích hoạt hàm. Phạm vi sử dụng biến cục bộ chỉ được phép trong nội bộ hàm. Chú ý, khi tên biến toàn cục trùng với tên biến cục bộ thì ưu tiên xử lý dành cho biến cục bộ. Ví dụ dưới đây sẽ minh họa cho biến toàn cục và biến cục bộ.

```
//Ví dụ 1.2. Biến toàn cục và biến cục bộ
#include <iostream>
using namespace std;
int a = 10, b = 20;
int Swap (int &a, int & b){
        int t = a; a=b; b = t;
}
int main(void){
        Swap(a, b);    //Tại điểm này ta thao tác với hai biến toàn cục a, b
        cout<<"a = "<<a <<"  b = "<<b<<endl;
        int a = 5, b = 7; //Từ vị trí này trở đi
        Swap(a,b);//Ta hoàn toàn thao tác với hai biến a, b cục bộ
        cout<<"a = "<<a <<"  b = "<<b<<endl;
}
```

1.2. Thuật toán và một số vấn đề liên quan

Như đã trình bày trong Mục 1.1.1, cấu trúc dữ liệu là phương pháp biểu diễn các đối tượng ở thế giới thực thành một đối tượng trong máy tính. Còn thuật toán được hiểu là phương pháp xử lý các đối tượng dữ liệu đã được biểu diễn để đưa ra kết quả mong muốn. Ta có thể tổng quát hóa khái niệm thuật toán như sau.

Định nghĩa thuật toán (Algorithm): Thuật toán F giải bài toán P là dãy các thao tác sơ cấp F_1, F_2,..,F_N trên tập dữ kiện đầu vào (*Input*) để đưa ra được kết quả ra (*Output*).

$F_1F_2..F_N(\ Input\ ) \rightarrow Ouput$.

- $F = F_1F_2..F_N$ được gọi là thuật toán giải bài toán P. Trong đó, mỗi F_i là các phép toán sơ cấp.
- *Input* được gọi là tập dữ kiện đầu vào hay tập thông tin đầu vào.
- *Output* là kết quả nhận được sau khi thực hiện thuật toán F trên tập *Input*.

Ví dụ. Thuật toán tìm USCLN(a, b).

```
int     USCLN ( int a,  int  b) {//đầu vào là số nguyên a, b
        while (b!=0 ) {//lặp trong khi b khác 0
                x = a % b; //lấy x là a mode b
                a = b; //đặt a bằng b
                b =x; //đặt b bằng x
        }
        return(a);//kết quả thực thi thuật toán
}
```

Những đặc trưng cơ bản của thuật toán:

- **Tính đơn định**. Ở mỗi bước của thuật toán, các thao tác sơ cấp phải hết sức rõ ràng, không tạo ra sự lộn xộn, nhập nhằng, đa nghĩa. Thực hiện đúng các bước của thuật toán trên tập dữ liệu đầu vào chỉ cho duy nhất một kết quả.
- **Tính dừng**. Thuật toán không được rơi vào quá trình vô hạn. Thuật toán phải dừng lại và cho kết quả sau một số hữu hạn các bước.
- **Tính đúng**. Sau khi thực hiện tất cả các bước của thuật toán theo đúng qui trình đã định, ta phải nhận được kết quả mong muốn với mọi bộ dữ liệu đầu vào. Kết quả đó được kiểm chứng bằng yêu cầu của bài toán.
- **Tính phổ dụng**. Thuật toán phải dễ sửa đổi để thích ứng được với bất kỳ bài toán nào trong lớp các bài toán cùng loại và có thể làm việc trên nhiều loại dữ liệu khác nhau.
- **Tính khả thi**. Thuật toán phải dễ hiểu, dễ cài đặt, thực hiện được trên máy tính với thời gian cho phép.

Đối với thuật toán ta cần quan tâm đến những vấn đề sau:

- **Biểu diễn thuật toán:** xác định ngôn ngữ để biểu diễn thuật toán.

- **Đánh giá độ phức tạp thuật toán**: ước lượng thời gian và không gian nhớ khi thực hiện thuật toán.
- **Kiểm nghiệm thuật toán**: kiểm nghiệm thuật toán với các bộ dữ liệu thực khác nhau.
- **Cài đặt thuật toán**: cài đặt thuật toán bằng ngôn ngữ lập trình cụ thể.

1.3. Biểu diễn thuật toán

Có ba ngôn ngữ chính để biểu diễn thuật toán: ngôn ngữ tự nhiên, ngôn ngữ máy tính và ngôn ngữ hình thức.

- **Ngôn ngữ tự nhiên** là phương tiện giao tiếp giữa con người với con người. Ta có thể sử dụng chính ngôn ngữ này vào việc biểu diễn thuật toán.
- **Ngôn ngữ máy tính** là phương tiện giao tiếp giữa máy tính và máy tính. Trong trường hợp này ta có thể sử dụng bất kỳ ngôn ngữ lập trình nào để biểu diễn thuật toán (C, Pascal, Java…).
- **Ngôn ngữ hình thức**. Ngôn ngữ hình thức là phương tiện giao tiếp trung gian giữa con người và hệ thống máy tính. Ví dụ ngôn ngữ sơ đồ khối, ngôn ngữ tựa tự nhiên, ngôn ngữ đặc tả. Đặc điểm chung của các loại ngôn ngữ hình thức là việc sử dụng nó rất gần gũi với ngôn ngữ tự nhiên, rất gần gũi với ngôn ngữ máy tính. Tuy nhiên, ngôn ngữ hình thức lại không phụ thuộc vào ngôn ngữ tự nhiên, không phụ thuộc vào ngôn ngữ máy tính. Chính vì lý do này, ngôn ngữ hình thức được sử dụng phổ biến trong biểu diễn thuật toán.

Ví dụ dưới đây sẽ minh họa cho các ngôn ngữ biểu diễn thuật toán.

//**Ví dụ 1.3.** Biểu diễn thuật toán bằng ngôn ngữ tự nhiên

Đầu vào (Input). Hai số tự nhiên a, b.

Đầu ra (Output). Số nguyên u lớn nhất để a và b đều chia hết cho u.

Thuật toán (Euclide Algorithm):

Bước 1. Đưa vào hai số tự nhiên a và b.

Bước 2. Nếu b≠ 0 thì chuyển đến bước 3, nếu b=0 thì thực hiện bước 4.

Bước 3. Đặt r = a mod b; a = b; b = r ; Quay quay trở lại bước 2.

Bước 4 (Output). Kết luận u=a là số nguyên cần tìm.

```
//Ví dụ 1.4. Biểu diễn thuật toán bằng ngôn ngữ máy tính (C++)
int     USCLN( int  a, int  b) {
        while ( b != 0 ) {//lặp trong khi b khác 0
                r = a % b; //đặt r bằng phần dư của a/b
                a = b; // đặt a bằng b
                b = r; //đặt b bằng r
        }
        return(a);//trả lại giá trị a
}
```

```
//Ví dụ 1.5. Biểu diễn thuật toán bằng ngôn ngữ hình thức
Thuật toán Euclide:
Đầu vào (Input): a∈N, a∈N.
Đầu ra (Output): s = max { u ∈N : a mod u =0 and b mod u =0}.
Format : s = Euclide (a, b).
Actions :
        while (b ≠ 0 )  do //lặp trong khi b khác 0
                r = a mod b; //đặt r bằng a mod b
                a = b; //đổi giá trị của a thành b
                b = r;// đổi giá trị của b thành r
        endwhile;//kết thúc cấu trúc lặp while
        return(a);//giá trị trả về của hàm
Endactions.
```

Một số lưu ý trong khi biểu diễn thuật toán bằng ngôn ngữ hình thức:

- Khi biểu diễn bằng ngôn ngữ hình thức ta được phép sử dụng cả ngôn ngữ tự nhiên hoặc ngôn ngữ máy tính thông dụng. Mỗi bước thực hiện của thuật toán không cần mô tả quá chi tiết mà chỉ cần mô tả một cách hình thức miễn là đầy đủ thông tin để chuyển đổi thành ngôn ngữ lập trình.
- Đối với những thuật toán phức tạp nặng nề về tính toán, các công thức cần được mô tả một cách tường minh, có ghi chú rõ ràng.
- Đối với các thuật toán kinh điển thì ta cần phải thuộc. Không bắt buộc phải chứng minh lại độ phức tạp của các thuật toán kinh điển.

1.4. Độ phức tạp thời gian của thuật toán

Một bài toán có thể thực hiện bằng nhiều thuật toán khác nhau. Lựa chọn giải thuật nhanh nhất để giải quyết bài toán là một nhu cầu của thực tế. Vì vậy ta cần phải có một ước lượng cụ bằng toán học để xác định mức độ nhanh chậm của mỗi giải thuật.

1.4.1. Khái niệm độ phức tạp thuật toán

Thời gian thực hiện một giải thuật bằng chương trình máy tính phụ thuộc vào các yếu tố:

- Kích thước dữ liệu đầu vào: một giải thuật hay một chương trình máy tính thực hiện trên tập dữ liệu có kích thước lớn hiển nhiên mất nhiều thời gian hơn thuật toán hoặc chương trình này thực hiện trên tập dữ liệu đầu vào có kích thước nhỏ.
- Phần cứng của hệ thống máy tính: hệ thống máy tính có tốc độ cao thực hiện nhanh hơn trên hệ thống máy tính có tốc độ thấp.

Tuy nhiên, nếu ta quan niệm thời gian thực hiện của một thuật toán là số các phép toán sơ cấp thực hiện trong thuật toán đó thì phần cứng máy tính không còn là yếu tố ảnh hưởng đến quá trình xác định thời gian thực hiện của một thuật toán. Với quan niệm này, độ phức tạp thời gian thực hiện của một thuật toán chỉ còn phụ thuộc duy nhất vào độ dài dữ liệu đầu vào.

Gọi độ dài dữ liệu đầu vào là $T(n)$. Khi đó, số lượng các phép toán sơ cấp để giải bài toán P thực hiện theo thuật toán $F=F_1F_2..F_n$ trên độ dài dữ liệu $T(n)$ là $F(T(n))$. Để xác định số lượng các phép toán sơ cấp F_i ($i=1, 2, .., n$) thực hiện trong thuật toán F ta cần phải giải bài toán đếm để xác định $F(T(n))$. Đây là bài toán vô cùng khó và không phải lúc nào cũng giải được []. Để đơn giản điều này, người ta thường tìm đến các phương pháp xấp xỉ để tính toán độ phức tạp thời gian của một thuật toán. Điều này có nghĩa, khi ta không thể xây dựng được công thức đếm $F(T(n))$, nhưng ta lại có khẳng định chắc chắn $F(T(n))$ không vượt quá một phiếm hàm biết trước $G(n)$ thì ta nói $F(T(n))$ thực hiện nhanh nhất là $G(n)$.

Tổng quát, *cho hai hàm $f(x)$, $g(x)$ xác định trên tập các số nguyên dương hoặc tập các số thực. Hàm $f(x)$ được gọi là $O(g(x))$ nếu tồn tại một hằng số $C>0$ và n_0 sao cho:*

$$|f(x)| \le C.|g(x)| \text{ với mọi } x \ge n_0.$$

Điều này có nghĩa với các giá trị $x \ge n_0$ hàm $f(x)$ bị chặn trên bởi hằng số C nhân với $g(x)$. Nếu $f(x)$ là thời gian thực hiện của một thuật toán thì ta nói giải thuật đó có cấp $g(x)$ hay độ phức tạp thuật toán $f(x)$ là $O(g(x))$.

Ghi chú. Các hằng số C, n_0 thỏa mãn điều kiện trên là không duy nhất. Nếu có đồng thời $f(x)$ là $O(g(x))$ và $h(x)$ thỏa mãn $g(x) < h(x)$ với $x>n_0$ thì ta cũng có $f(x)$ là $O(h(n))$.

Ví dụ 1.6. Cho $f(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$; trong đó, a_i là các số thực (i =0,1, 2, ..,n). Khi đó $f(x) = O(x^n)$.

Chứng minh. Thực vậy, với mọi $x>1$ ta có:

$$|f(x) = |a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0|$$
$$\le |a_n|x^n + |a_{n-1}|x^{n-1} + \cdots + |a_1|x + |a_0|$$

$$\leq |a_n|x^n + |a_{n-1}|x^n + \cdots + |a_1|x^n + |a_0|x^n$$
$$\leq x^n(|a_n| + |a_{n-1}| + \cdots + |a_{1|} + |a_{0|})$$
$$\leq C.x^n = O(x^n)$$. Trong đó, $C = (|a_n| + |a_{n-1}| + \cdots + |a_{1|} + |a_{0|})$.

1.4.2. Một số qui tắc xác định độ phức tạp thuật toán

Như đã đề cập ở trên, bản chất của việc xác định độ phức tạp thuật toán là giải bài toán đếm số lượng các phép toán sơ cấp thực hiện trong thuật toán đó. Do vậy, tất cả các phương pháp giải bài toán đếm thông thường đều được áp dụng trong khi xác định độ phức tạp thuật toán. Hai nguyên lý cơ bản để giải bài toán đếm là nguyên lý cộng và nguyên lý nhân cũng được mở rộng trong khi ước lượng độ phức tạp thuật toán.

Nguyên tắc tổng: Nếu $f_1(x)$ có độ phức tạp là $O(g_1(x))$ và $f_2(x)$ có độ phức tạp là $O(g_2(x))$ thì độ phức tạp của $(f_1(x) + f_2(x)$ là $O(Max(g_1(x), g_2(x))$.

Chứng minh. Vì $f_1(x)$ có độ phức tạp là $O(g_1(x)$ nên tồn tại hằng số C_1 và k_1 sao cho $|f_1(x)|\leq|g_1(x)|$ với mọi $x \geq k_1$. Vì $f_2(x)$ có độ phức tạp là $O(g_2(x))$ nên tồn tại hằng số C_2 và k_2 sao cho $|f_2(x)|\leq|g_2(x)|$ với mọi $x \geq k_2$.

Ta lại có :

$$|f_1(x)+ f_2(x)| \leq |f_1(x)| + |f_2(x)|$$
$$\leq C_1|g_1(x)| + C_2|g_2(x)|$$
$$\leq C|g(x)| \text{ với mọi } x > k;$$

Trong đó, $C = C_1 + C_2$; $g(x) = max(g_1(x), g_2(x))$; $k = max (k_1, k_2)$.

Tổng quát. Nếu độ phức tạp của $f_1(x)$, $f_2(x)$,.., $f_m(x)$ lần lượt là $O(g_1(x))$, $O(g_2(x))$,.., $O(g_n(x))$ thì độ phức tạp của $f_1(x) + f_2(x) + ..+f_m(x)$ là $O(max(g_1(x), g_2(x),..,g_m(x))$.

Nguyên tắc nhân: Nếu $f(x)$ có độ phức tạp là $O(g(x)$ thì độ phức tạp của $f^n(x)$ là $O(g^n(x))$. Trong đó:

$f^n(x) = f(x).f(x)….f(x)$. //n lần $f(x)$.

$g^n(x) = g(x).g(x)…g(x)$.//n lần $g(x)$

Chứng minh. Thật vậy theo giả thiết $f(x)$ là $O(g(x))$ nên tồn tại hằng số C và k sao cho với mọi $x>k$ thì $|f(x)|\leq C.|g(x)$. Ta có:

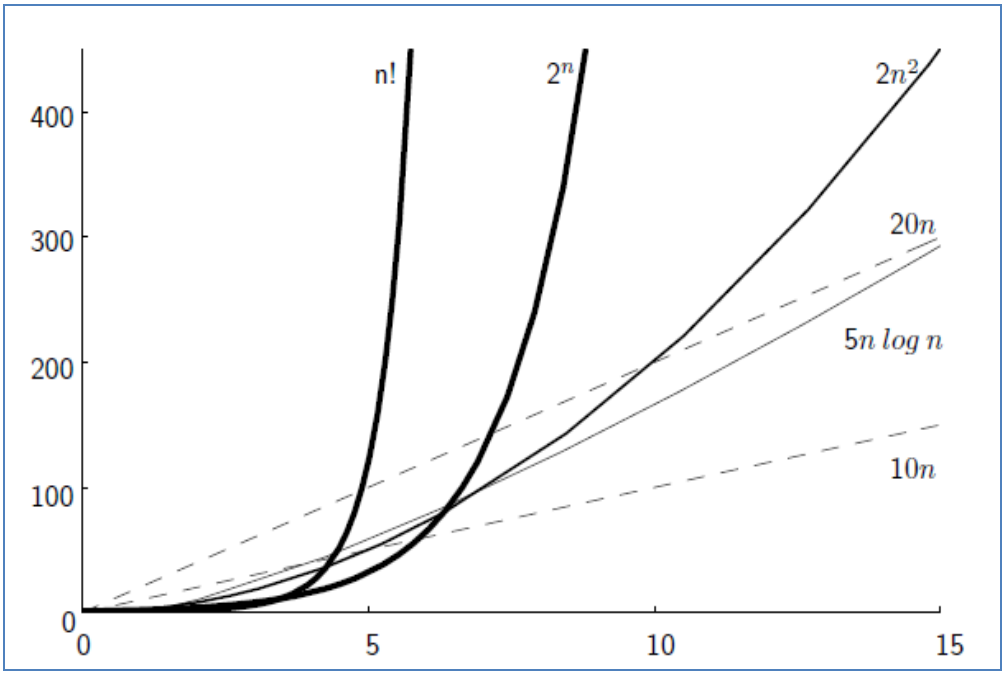
$$|f^n(x)| = |f^1(x).f^2(x) \dots f^n(x)|$$
$$\leq |C.g^1(x).C.g^2(x) \dots C.g^n(x)|$$
$$\leq |C^n.g^n(x)| = O(g^n(x))$$

1.4.3. Một số dạng hàm được dùng xác định độ phức tạp thuật toán

Như đã đề cập ở trên, để xác định chính xác độ phức tạp thuật toán $f(x)$ là bài toán khó nên ta thường xấp xỉ độ phức tạp thuật toán với một phiếm hàm O(g(x)). Dưới đây là một số phiếm hàm của O(g(x)).

Bảng 1.1. Các dạng hàm xác định độ phức tạp thuật toán

Dạng phiếm hàm	Tên gọi
O(1)	Hằng số
O(log(log(n)))	Logarit của logarit
O (log(n))	Logarithm
O(n)	Tuyến tính
$O(n^2)$	Bậc 2
$O(n^3)$	Bậc 3
$O(n^m)$	Đa thức (m là hằng số)
$O(m^n)$	Hàm mũ
O(n!)	Giai thừa



Hình 1.1. *Độ tăng của các hàm theo độ dài dữ liệu*

Dưới đây là một số qui tắc xác định O(g(x)):

- Nếu một thuật toán có độ phức tạp hằng số thì thời gian thực hiện thuật toán đó không phụ thuộc vào độ dài dữ liệu.
- Một thuật toán có độ phức tạp logarit của f(n) thì ta viết O(log(n)) mà không cần chỉ rõ cơ số của phép logarit.
- Với P(n) là một đa thức bậc k thì $O(P(n)) = O(n^k)$.
- Thuật toán có độ phức tạp đa thức hoặc nhỏ hơn được xem là những thuật toán thực tế có thể thực hiện được bằng máy tính. Các thuật toán có độ phức tạp hàm mũ, hàm giai thừa được xem là những thuật toán thực tế không giải được bằng máy tính.

1.5. Độ phức tạp của các cấu trúc lệnh

Để đánh giá độ phức tạp của một thuật toán đã được mã hóa thành chương trình máy tính ta thực hiện theo một số qui tắc sau.

Độ phức tạp hằng số O(1): đoạn chương trình không chứa vòng lặp hoặc lời gọi đệ qui có tham biến là một hằng số.

Ví dụ 1.7. Đoạn chương trình dưới đây có độ phức tạp hằng số.

```
for (i=1; i<=c; i++) {
        <Tập các chỉ thị có độ phức tạp O(1)>;
}
```

Độ phức tạp O(n): Độ phức tạp của hàm hoặc đoạn code là O(n) nếu biến trong vòng lặp tăng hoặc giảm bởi mộ hằng số c.

Ví dụ 1.8. Đoạn code dưới đây có độ phức tạp hằng số.

```
for (i=1; i<=n; i = i + c ) {
        <Tập các chỉ thị có độ phức tạp O(1)>;
}
for (i=n; i>0; i = i - c ){
        <Tập các chỉ thị có độ phức tạp O(1)>;
}
```

Độ phức tạp đa thức $O(n^c)$: Độ phức tạp của *c* vòng lặp lồng nhau, mỗi vòng lặp đều có độ phức tạp O(n) là **$O(n^c)$.**

Ví dụ 1.9. Đoạn code dưới đây có độ phức tạp $O(n^2)$.

```
for (i=1; i<=n; i = i + c ) {
        for (j=1; j<=n; j = j + c ){
                <Tập các chỉ thị có độ phức tạp O(1)>;
        }
}
```

```
for (i = n; i >0 ; i = i - c ) {
        for (j = i- 1; j>1; j = j -c ){
                <Tập các chỉ thị có độ phức tạp O(1)>;
        }
}
```

Độ phức tạp logarit O(Log(n)): Độ phức tạp của vòng lặp là log(n) nếu biểu thức khởi đầu lại của vòng lặp được chia hoặc nhân với một hằng số c.

Ví dụ 1.10. Đoạn code dưới đây có độ phức tạp Log(n).

```
for (i=1; i <=n; i = i *c ){
        <Tập các chỉ thị có độ phức tạp O(1)>;
}
for (j=n; j >0 ; j = j / c ){
        <Tập các chỉ thị có độ phức tạp O(1)>;
}
```

Độ phức tạp hằng số O(Log (Log(n))): nếu biểu thức khởi đầu lại của vòng lặp được nhân hoặc chia cho một hàm mũ.

Ví dụ 1.11. Đoạn code dưới đây có độ phức tạp Log Log(n).

```
for (i=1; j<=n; j*= Pow(i, c) ){
        <Tập các chỉ thị có độ phức tạp O(1)>;
}
for (j=n; j>=0; j = j- Function(j)  ){  //Function(j) =sqrt(j) hoặc lớn hơn 2.
        <Tập các chỉ thị có độ phức tạp O(1)>;
}
```

Độ phức tạp của chương trình: độ phức tạp của một chương trình bằng số lần thực hiện một chỉ thị tích cực trong chương trình đó. Trong đó, một chỉ thị được gọi là tích cực trong chương trình nếu chỉ thị đó phụ thuộc vào độ dài dữ liệu và thực hiện không ít hơn bất kỳ một chỉ thị nào khác trong chương trình.

Ví dụ 1.12. Tìm độ phức tạp thuật toán sắp xếp kiểu Bubble-Sort?

```
Void Bubble-Sort ( int  A[],  int n ) {
        for ( i=1; i<n; i++) {
                for ( j = i+1; j<=n; j++){
                        if (A[i] > A[j]) {//đây chính là chỉ thị tích cực
                                t = A[i]; A[i] = A[j]; A[j] = t;
                        }
                }
        }
}
```

Lời giải. Sử dụng trực tiếp nguyên lý cộng ta có:

- Với i =1 ta cần sử dụng n-1 phép so sánh A[i] với A[j];
- Với i = 2 ta cần sử dụng n-1 phép so sánh A[i] với A[j];
- . . . . .
- Với i = n-1 ta cần sử dụng 1 phép so sánh A[i] với A[j];

Vì vậy tổng số các phép toán cần thực hiện là:

$S = (n-1) + (n-2) + \ldots + 2 + 1 = n(n-1)/2 \leq n^2 = O(n^2)$.

Ghi chú. Độ phức tạp thuật toán cũng là số lần thực hiện phép toán tích cực. Phép toán tích cực là phép toán thực hiện nhiều nhất đối với thuật toán.

1.6. Qui trình giải quyết bài toán trên máy tính

Để giải quyết một bài toán hoặc vấn đề trong tin học ta thực hiện thông qua 6 bước như sau:

Bước 1. Xác định yêu cầu bài toán. Xem xét bài toán cần xử lý vấn đề gì? Giả thiết nào đã được biết trước và lời giải cần đạt những yêu cầu gì? Ví dụ thời gian, hay không gian nhớ.

Bước 2. Tìm cấu trúc dữ liệu thích hợp biểu diễn các đối tượng cần xử lý của bài toán. Cấu trúc dữ liệu phải biểu diễn đầy đủ các đối tượng thông tin vào của bài toán. Các thao tác trên cấu trúc dữ liệu phải phù hợp với những thao tác của thuật toán được lựa chọn. Cấu trúc dữ liệu phải cài đặt được bằng ngôn ngữ lập trình cụ thể đáp ứng yêu cầu bài toán.

Bước 3. Lựa chọn thuật toán. Thuật toán phải đáp ứng được yêu của bài toán và phù hợp với cấu trúc dữ liệu đã được lựa chọn Bước 1.

Bước 4. Cài đặt thuật toán. Thuật toán cần được cài đặt bằng một ngôn ngữ lập trình cụ thể. Ngôn ngữ lập trình sử dụng phải có các cấu trúc dữ liệu đã lựa chọn.

Bước 5. Kiểm thử chương trình. Thử nghiệm thuật toán (chương trình) trên các bộ dữ liệu thực. Các bộ dữ liệu cần phải bao phủ lên tất cả các trường hợp của thuật toán.

- **Tối ưu chương trìn**h: Cải tiến để chương trình tốt hơn.

BÀI TẬP

BÀI 1. ƯỚC SỐ CHUNG LỚN NHẤT VÀ BỘI SỐ CHUNG NHỎ NHẤT

Viết chương trình tìm ước số chung lớn nhất và bội số chung nhỏ nhất của hai số nguyên dương a,b.

Dữ liệu vào: Dòng đầu ghi số bộ test. Mỗi bộ test ghi trên một dòng 2 số nguyên a và b không quá 9 chữ số.

Kết quả: Mỗi bộ test ghi trên 1 dòng, lần lượt là USCLN, sau đó đến BSCNN.

Ví dụ:

Input	Output
2 12 34 1234 5678	2 204 2 3503326

BÀI 2. BẮT ĐẦU VÀ KẾT THÚC

Viết chương trình kiểm tra một số nguyên dương bất kỳ (2 chữ số trở lên, không quá 9 chữ số) có chữ số bắt đầu và kết thúc bằng nhau hay không.

Dữ liệu vào: Dòng đầu tiên ghi số bộ test. Mỗi bộ test viết trên một dòng số nguyên dương tương ứng cần kiểm tra.

Kết quả: Mỗi bộ test viết ra YES hoặc NO, tương ứng với bộ dữ liệu vào

Ví dụ:

Input	Output
2 12451 1000012	YES NO

BÀI 3. MẢNG ĐỐI XỨNG

Nhập một dãy số nguyên có n phần tử (n không quá 100, các phần tử trong dãy không quá 10^9). Hãy viết chương trình kiểm tra xem dãy có phải đối xứng hay không. Nếu đúng in ra YES, nếu sai in ra NO.

Dữ liệu vào: Dòng đầu ghi số bộ test, mỗi bộ test gồm hai dòng. Dòng đầu là số phần tử của dãy, dòng sau ghi ra dãy đó, mỗi số cách nhau một khoảng trống.

Kết quả: Ghi ra YES hoặc NO trên một dòng.

Ví dụ

Input	Ouput
2 4 1 4 4 1 5 1 5 5 5 3	YES NO

BÀI 4. PHÂN TÍCH THỪA SỐ NGUYÊN TỐ

Hãy phân tích một số nguyên dương thành tích các thừa số nguyên tố.
Dữ liệu vào: Dòng đầu tiên ghi số bộ test. Mỗi bộ test viết trên một dòng số nguyên dương n không quá 9 chữ số.
Kết quả: Mỗi bộ test viết ra thứ tự bộ test, sau đó lần lượt là các số nguyên tố khác nhau có trong tích, với mỗi số viết thêm số lượng số đó. Xem ví dụ để hiểu rõ hơn về cách viết kết quả.
Ví dụ

Input	Output
3 60 128 10000	Test 1: 2(2) 3(1) 5(1) Test 2: 2(7) Test 3: 2(4) 5(4)

BÀI 5. PHÉP CỘNG

Cho một phép toán có dạng **a + b = c** với a,b,c chỉ là các số nguyên dương có một chữ số. Hãy kiểm tra xem phép toán đó có đúng hay không.

Dữ liệu vào: Chỉ có một dòng ghi ra phép toán (gồm đúng 9 ký tự)

Kết quả: Ghi ra YES nếu phép toán đó đúng. Ghi ra NO nếu sai.

Ví dụ:

Test 1	Test 2
Input 1 + 2 = 3 Output YES	Input 2 + 2 = 5 Output NO

BÀI 6. SỐ TĂNG GIẢM

Một số được gọi là số tăng giảm nếu số đó có các chữ số thỏa mãn hoặc không giảm, hoặc không tăng từ trái qua phải. Hãy kiểm tra xem một số có phải số tăng giảm hay không.

Dữ liệu vào: Dòng đầu tiên ghi số bộ test. Mỗi bộ test viết trên một dòng một số nguyên dương cần kiểm tra, không quá 500 chữ số.

Kết quả: Mỗi bộ test viết ra chữ YES nếu đó đúng là số tăng giảm, chữ NO nếu ngược lại.

Input	Output
3 234556677777777777788888888888999999999 9877777777777777777777765544222222211111111111000 434353124325436576576587698980978764654656879 87	YES YES NO

BÀI 7. CHUẨN HÓA 1

Các xâu họ tên trong Tiếng Việt được viết theo dạng chuẩn như sau:

- Chữ cái đầu mỗi từ viết hoa. Các chữ cái sau viết thường
- Các từ cách nhau đúng 1 khoảng trống

Hãy viết chương trình đưa danh sách các xâu họ tên về dạng chuẩn

Dữ liệu vào: Dòng 1 ghi số N là xâu họ tên trong danh sách. N dòng tiếp theo ghi lần lượt các xâu họ tên (không quá 80 ký tự). **Kết quả:** Ghi ra các xâu chuẩn.

Ví dụ:

Input	Output
4 nGUYEn quaNG vInH tRan thi THU huOnG nGO quoC VINH lE tuAn aNH	Nguyen Quang Vinh Tran Thi Thu Huong Ngo Quoc Vinh Le Tuan Anh

BÀI 8. CHUẨN HÓA 2

Một trong các cách viết họ tên theo dạng chuẩn trong Tiếng Anh là phần họ được viết sau cùng, phân tách với phần tên đệm và tên bởi dấu phẩy. Các chữ cái của phần họ đều viết hoa.

Cho trước các xâu họ tên. Hãy đưa về dạng chuẩn tương ứng.

Dữ liệu vào:

- Dòng 1 ghi số N là xâu họ tên trong danh sách
- N dòng tiếp theo ghi lần lượt các xâu họ tên (không quá 50 ký tự)

Kết quả: Ghi ra các xâu chuẩn.

Ví dụ:

Input	Output
4 nGUYEn quaNG vInH tRan thi THU huOnG nGO quoC VINH lE tuAn aNH	Quang Vinh, NGUYEN Thi Thu Huong, TRAN Quoc Vinh, NGO Tuan Anh, LE

BÀI 9. VÒNG TRÒN

Tí viết bảng chữ cái 2 lần lên trên một vòng tròn, mỗi kí tự xuất hiện đúng 2 lần. Sau đó nối lần lượt các kí tự giống nhau lại. Tổng cộng có 26 đoạn thẳng.

Hình vẽ quá chằng chịt, Tí muốn đố các bạn xem có tất cả bao nhiêu giao điểm?

Một giao điểm được tính khi hai đường thẳng của một cặp kí tự cắt nhau.

Input

Gồm một xâu có đúng 52 kí tự in hoa. Mỗi kí tự xuất hiện đúng 2 lần.

Output

In ra đáp án tìm được.

Ví dụ:

Input	Output
ABCCABDDEEFFGGHHIIJJKKLLMMNNOOPPQQRRSSTTUUVVWWXXYYZZ	1

Giải thích test: Chỉ có duy nhất cặp kí tự 'A', 'B' thỏa mãn.

BÀI 10. CHIA HẾT CHO 2

Cho số nguyên dương N.
Nhiệm vụ của bạn là hãy xác định xem có bao nhiêu ước số của N chia hết cho 2.

Input:Dòng đầu tiên là số lượng bộ test T ($T \leq 100$). Mỗi bộ test gồm một số nguyên N ($1 \leq N \leq 10^9$)

Output: Với mỗi test, in ra đáp án tìm được trên một dòng.

Ví dụ:

Input:	**Output:**
2 9 8	0 3

CHƯƠNG 2. MỘT SỐ LƯỢC ĐỒ THUẬT TOÁN KINH ĐIỂN

Nội dung chính của chương trình bày một số lược đồ thuật toán kinh điển dùng để giải lớp các bài toán liệt kê, bài toán đếm, và bài toán tối ưu và bài toán tồn tại. Mỗi lược đồ thuật toán giải quyết một lớp các bài toán thỏa mãn một số tính chất nào đó. Đây là những lược đồ thuật toán quan trọng nhằm giúp người học vận dụng nó trong khi giải quyết các vấn đề trong tin học. Các lược đồ thuật toán được trình bày trong chương này bao gồm: thuật toán sinh, thuật toán đệ qui, thuật toán quay lui, thuật toán tham lam, thuật toán nhánh cận, thuật toán qui hoạch động.

2.1. Mô hình thuật toán sinh (Generative Algorithm)

Mô hình thuật toán sinh được dùng để giải lớp các bài toán liệt kê, bài toán đếm, bài toán tối ưu, bài toán tồn tại thỏa mãn hai điều kiện:

- **Điều kiện 1:** *Có thể xác định được một thứ tự trên tập các cấu hình cần liệt kê của bài toán. Biết cấu hình đầu tiên, biết cấu hình cuối cùng.*
- **Điều kiện 2:** *Từ một cấu hình chưa phải cuối cùng, ta xây dựng được thuật toán sinh ra cấu hình đứng ngay sau nó.*

Mô hình thuật toán sinh được biểu diễn thành hai bước: bước khởi tạo và bước lặp. Tại bước khởi tạo, cấu hình đầu tiên của bài toán sẽ được thiết lập. Điều này bao giờ cũng thực hiện được theo giả thiết của bài toán. Tại bước lặp, quá trình lặp được thực hiện khi gặp phải cấu hình cuối cùng. Điều kiện lặp của bài toán bao giờ cũng tồn tại theo giả thiết của bài toán. Hai chỉ thị cần thực hiện trong thân vòng lặp là đưa ra cấu hình hiện tại và sinh ra cấu hình kế tiếp. Mô hình sinh kế tiếp được thực hiện tùy thuộc vào mỗi bài toán cụ thể. Tổng quát, mô hình thuật toán sinh được thể hiện như dưới đây.

```
Thuật toán Generation;
begin
        Bước1 (Khởi tạo):
                <Thiết lập cấu hình đầu tiên>;
        Bước 2 (Bước lặp):
                while (<Lặp khi cấu hình chưa phải cuối cùng>) do
                        <Đưa ra cấu hình hiện tại>;
                        <Sinh ra cấu hình kế tiếp>;
        endwhile;
End.
```

Ví dụ 2.1. Vector $X = (x_1, x_2, .., x_n)$, trong đó $x_i = 0, 1$ được gọi là một xâu nhị phân có độ dài *n*. Hãy liệt kê các xâu nhị phân có độ dài *n*. Ví dụ với n=4, ta sẽ liệt kê được 24 xâu nhị phân độ dài 4 như trong Bảng 2.1.

Bảng 2.1. Các xâu nhị phân độ dài 4

STT	$X=(x_1, x_2, x_3, x_4)$	STT	$X=(x_1, x_2, x_3, x_4)$
0	0 0 0 0	**8**	1 0 0 0
1	0 0 0 1	**9**	1 0 0 1
2	0 0 1 0	**10**	1 0 1 0
3	0 0 1 1	**11**	1 0 1 1
4	0 1 0 0	**12**	1 1 0 0
5	0 1 0 1	**13**	1 1 0 1
6	0 1 1 0	**14**	1 1 1 0
7	0 1 1 1	**15**	1 1 1 1

Lời giải:

Điều kiện 1: Gọi thứ tự của xâu nhị phân $X=(x_1, x_2,.., x_n)$ là $f(X)$. Trong đó, $f(X)= k$ là số chuyển đổi xâu nhị *X* thành số ở hệ cơ số 10. Ví dụ, xâu $X = (1, 0, 1, 1)$ được chuyển thành số hệ cơ số 10 là 11 thì ta nói xâu *X* có thứ tự 11. Với cách quan niệm này, xâu đứng sau xâu có thứ tự 11 là 12 chính là xâu đứng ngay sau xâu $X = (1, 0, 1, 1)$. Xâu đầu tiên có thứ tự là *0* ứng với xâu có *n* số *0*. Xâu cuối cùng có thứ tự là 2^n-1 ứng với xâu có *n* số *1*. Như vậy, điều kiện 1 của thuật toán sinh đã được thỏa mãn.

Điều kiện 2: Về nguyên tắc ta có thể lấy $k = f(X)$ là thứ tự của một xâu bất kỳ theo nguyên tắc ở trên, sau đó lấy thứ tự của xâu kế tiếp là $(k + 1)$ và chuyển đổi $(k+1)$ thành số ở hệ cơ số 10 ta sẽ được xâu nhị phân tiếp theo. Xâu cuối cùng sẽ là xâu có n số 1 ứng với thứ tự $k = 2^n-1$. Với cách làm này, ta có thể coi mỗi xâu nhị phân là một số, mỗi thành phần của xâu là một bít và chỉ cần cài đặt thuật toán chuyển đổi cơ số ở hệ 10 thành số ở hệ nhị phân. Ta có thể xây dựng thuật toán tổng quát hơn bằng cách xem mỗi xâu nhị phân là một mảng các phần tử có giá trị 0 hoặc 1. Sau đó, duyệt từ vị trí bên phải nhất của xâu nếu gặp số 1 ta chuyển thành 0 và gặp số 0 đầu tiên ta chuyển thành 1. Ví dụ với xâu X = (0, 1, 1, 1) được chuyển thành xâu X= (1, 0, 0, 0), xâu X = (1,0,0,0) được chuyển thành xâu X =(1, 0, 0, 1). Lời giải và thuật toán sinh xâu nhị phân kế tiếp được thể hiện trong chương trình dưới đây. Trong đó, thuật toán sinh xâu nhị phân kế tiếp từ một xâu nhị phân bất kỳ là hàm *Next_Bits_String*().

```
#include <iostream>
#include <iomanip>
#define MAX 100
using namespace std;
```

```
int X[MAX], n, dem = 0; //sử dụng các biến toàn cục X[], n, OK, dem
bool OK =true;
void Init(void){ //khởi tạo xâu nhị phân đầu tiên
        cout<<"Nhập n="; cin>>n;
        for(int i = 1; i<=n; i++) //thiết lập xâu với n số 0
                X[i]=0;
}
void Result(void){ //đưa ra xâu nhị phân hiện tại
        cout<<"\n Xâu thứ "<<++dem<<":";
        for(int i=1; i<=n; i++)
           cout<<X[i]<<setw(3);
}
void Next_Bits_String(void){ //thuật toán sinh xâu nhị phân kế tiếp
        int i=n;
        while(i>0 && X[i]){ //duyệt từ vị trí bên phải nhất
                X[i]=0; //nếu gặp X[i] = 1 ta chuyển thành 0
                i--; //lùi lại vị trí sau
        }
        if (i>0)  X[i]=1; //gặp X[i] =0 đầu tiên ta chuyển thành 1
        else OK = false; //kết thúc khi gặp xâu có n số 1
}
int main(void){ //đây là thuật toán sinh
        Init(); //thiết lập cấu hình đầu tiên
        while(OK){//lặp khi chưa phải cấu hình cuối cùng
                Result(); //đưa ra cấu hình hiện tại
                Next_Bits_String(); //sinh ra cấu hình kế tiếp
        }
}
```

Ví dụ 2.2. Liệt kê các tập con k phần tử của 1, 2, .., n.

Lời giải. Mỗi tập con k phần tử của 1, 2, .., N là một tổ hợp chập K của 1, 2,.., N. Ví dụ với n=5, k= 3 ta sẽ có C(n,k) tập con trong Bảng 2.2.

Điều kiện 1. Ta gọi tập con X =(x_1,..x_K) là đứng trước tập con Y =(y_1, y_2,..y_K) nếu tìm được chỉ số t sao cho $x_1 = y_1$, $x_2 = y_2$,.., $x_{t-1} = y_{t-1}$, $x_t < y_t$. Ví dụ tập con $X = (1, 2, 3)$ đứng trước tập con $Y = (1, 2, 4)$ vì ta tìm được t=3 thỏa mãn $x_1 = y_1$, $x_2 = y_2$, $x_3 < y_3$. Tập con đầu tiên là $X = (1, 2,.., k)$, tập con cuối cùng là $(n–k+1,.., N)$. Như vậy điều kiện 1 của thuật toán sinh được thỏa mãn.

Điều kiện 2. Để ý rằng, tập con cuối cùng (*n-k+1,..., n*) luôn thỏa mãn đẳng thức *X*[*i*] = *n – k + i*. Ví dụ tập con cuối cùng *X*[] = (3, 4, 5) ta đều có: *X*[*1*] = *3 = 5 – 3 + 1*; *X*[*2*] = *4 = 5 – 3 + 2*; *X*[*3*] = *5 = 5 – 3 + 3*. Để tìm tập con kế tiếp từ tập con bất kỳ ta chỉ cần duyệt từ phải qua trái tập con *X*[] = (x_1, x_2, .., x_k) để xác định chỉ số *i* thỏa mãn điều kiện *X*[*i*] ≠ *n – k + i*. Ví dụ với *X*[] = (*1*, *4*, *5*), ta xác định được *i=1* vì *X*[*3*] = *5 = 5-3+3*, *X*[*2*] = *4 = 5-3+2*, và *X*[*1*] = *1 ≠ 5-3+1*. Sau khi xác định được chỉ số *i*, tập con mới sẽ được sinh là *Y*[] = (y_1,.., y_i, …, y_k) ra thỏa mãn điều kiện: $y_1 = x_1$, $y_2 = x_2$,.., $y_{i-1} = x_{i-1}$, $y_i = x_i+1$, và $y_j = x_t + j – i$ với (*j = j+1, ..., k*).

Bảng 2.2. Tập con 3 phần tử của 1, 2, 3, 4, 5

STT	Tập con
1	1 2 3
2	1 2 4
3	1 2 5
4	1 3 4
5	1 3 5
6	1 4 5
7	2 3 4
8	2 3 5
9	2 4 5
10	3 4 5

Chương trình cài đặt thuật toán sinh tập con k phần tử được thể hiện như dưới đây. Trong đó, thuật toán sinh tổ hợp kế tiếp có tên là Next_Combination().

```
#include <iostream>
#include <iomanip>
#define MAX 100
int X[MAX], n, k, dem=0;
bool OK = true;
using namespace std;
void Init(void){ //thiết lập tập con đầu tiên
        cout<<"\n Nhập n, k:"; cin>>n>>k;
        for(int i=1; i<=k; i++)  //tập con đầu tiên là 1, 2, .., k
                X[i] = i;
}
```

```
void Result(void){ //đưa ra tập con hiện tại
        cout<<"\n Kết quả "<<++dem<<":";
                for(int i=1; i<=k; i++) //đưa ra X[] =( x1, x2, .., xk)
                cout<<X[i]<<setw(3);
}
void Next_Combination(void){ //sinh tập con k phần tử từ tập con bất kỳ
        int i = k; //duyệt từ vị trí bên phải nhất của tập con
        while(i>0 && X[i]== n-k+i) //tìm i sao cho xi ≠ n-k+i
                i--;
        if (i>0){//nếu chưa phải là tập con cuối cùng
                X[i]= X[i]+1; //thay đổi giá trị tại vị  trí i: xi = xi +1;
                for(int j=i+1; j<=k; j++) //các vị trí j từ i+1,.., k
                        X[j] = X[i] + j - i; // được thay đổi là xj = xi +j - i;
        }
        else  //nếu là tập con cuối cùng
                OK = false; //ta kết thúc duyệt
}
int main(void){
        Init(); //khởi tạo cấu hình đầu tiên
        while(OK){ //lặp trong khi cấu hình chưa phải cuối cùng
                Result(); //đưa ra cấu hình hiện tại
                Next_Combination(); //sinh ra cấu hình kế tiếp
        }
}
```

Ví dụ 2.3. Liệt kê các hoán vị của 1, 2, .., n.

Lời giải. Mỗi hoán vị của 1, 2, .., N là một cách xếp có tính đến thứ tự của 1, 2,..,N. Số các hoán vị là N!. Ví dụ với N =3 ta có 6 hoán vị dưới đây.

Bảng 2.3. Hoán vị của 1, 2, 3.

STT	Hoán vị
1	1 2 3
2	1 3 2
3	2 1 3
4	2 3 1
5	3 1 2
6	3 2 1

Điều kiện 1. Có thể xác định được nhiều trật tự khác nhau trên các hoán vị. Tuy nhiên, thứ tự đơn giản nhất có thể được xác định như sau. Hoán vị $X=(x_1, x_2,.., x_n)$ được gọi là đứng sau hoán vị $Y = (y_1, y_2,..,y_n)$ nếu tồn tại chỉ số k sao cho $x_1 = y_1$, $x_2 = y_2,\ldots, x_{k-1} =y_{k-1}$, $x_k<y_k$. Ví dụ hoán vị X = (1, 2, 3) được gọi là đứng sau hoán vị Y =(1, 3, 2) vì tồn tại k =2 để $x_1 = y_1$, và $x_2<y_2$. Hoán vị đầu tiên là X[] = (1, 2, …, n), hoán vị cuối cùng là X[] = (n, n-1, …, 1).

Điều kiện 2. Được thể hiện thông qua hàm Next_Permutation() như chương trình dưới đây.

```
#include <iostream>
#include <iomanip>
#define MAX 100
int X[MAX], n,  dem=0;
bool OK = true;
using namespace std;
void Init(void){ //thiết lập hoán vị đầu tiên
        cout<<"\n Nhap n:"; cin>>n;
        for(int i=1; i<=n; i++) //thiết lập X[] = (1, 2, ..,n)
                X[i] = i;
}
void Result(void){ //đưa ra hoán vị hiện tại
        cout<<"\n Kết quả "<<++dem<<":";
        for(int i=1; i<=n; i++)
           cout<<X[i]<<setw(3);
}
void Next_Permutation(void){ //sinh ra hoán vị kế tiếp
        int j = n-1; //xuất phát từ vị trí j = n-1
        while(j>0 && X[j]>X[j+1]) //tìm chỉ số j sao cho X[j] < X[j+1]
                j--;
        if ( j > 0){ // nếu chưa phải hoán vị cuối cùng
                int k = n; //xuất phát từ vị trí k = n
                while(X[j]>X[k]) //tìm chỉ số k sao cho X[j] < X[k]
                        k--;
                int t = X[j]; X[j] = X[k]; X[k]=t; //đổi chỗ X[j] cho X[k]
                int r = j+1, s = n;
                while (r<=s){ //lật ngược lại đoạn từ j+1,..,n
                        t=X[r]; X[r]=X[s]; X[s]=t;
                        r++; s--;
```

```
            }
        }
        else  //nếu là cấu hình cuối cùng
            OK = false; //ta kết thúc duyệt
}
int main(void){ //đây là thuật toán sinh
        Init(); //thiết lập cấu hình đầu tiên
        while(OK){ //lặp trong khi cấu hình chưa phải cuối cùng
            Result(); //đưa ra cấu hình hiện tại
            Next_Permutation(); //sinh ra cấu hình kế tiếp
        }
}
```

2.2. Mô hình thuật toán đệ qui (Recursion Algorithm)

Một đối tượng được định nghĩa trực tiếp hoặc gián tiếp thông qua chính nó được gọi là phép định nghĩa bằng đệ qui. Thuật toán giải bài toán *P* một cách trực tiếp hoặc gián tiếp thông qua bài toán *P*' giống như *P* được gọi là thuật toán đệ qui giải bài toán *P*. Một hàm được gọi là đệ qui nếu nó được gọi trực tiếp hoặc gián tiếp đến chính nó.

Tổng quát, một bài toán có thể giải được bằng đệ qui nếu nó thỏa mãn hai điều kiện:

- **Phân tích được***: Có thể giải được bài toán P bằng bài toán P' giống như P. Bài tóa P' và chỉ khác P ở dữ liệu đầu vào. Việc giải bài toán P' cũng được thực hiện theo cách phân tích giống như P.*
- **Điều kiện dừng***: Dãy các bài toán P' giống như P là hữu hạn và sẽ dừng tại một bài toán xác định nào đó.*

Thuật toán đệ qui tổng quát có thể được mô tả như sau:

Thuật toán Recursion (P) {

1. *Nếu P thỏa mãn điều kiện dừng:*
   <Giải P với điều kiện dừng>;
2. Nếu *P* không thỏa mãn điều kiện dừng:
   <Giải P' giống như P:Recursion(P')>;

}

Ví dụ 2.4. Tìm tổng của n số tự nhiên đầu tiên bằng phương pháp đệ qui.

Lời giải. Gọi S_n là tổng của n số tự nhiên. Khi đó:

- **Bước phân tích**: dễ dàng phận thấy tổng n số tự nhiên $S_n = n + S_{n-1}$, với $n \geq 1$.
- **Điều kiện dừng**: $S_0 = 0$ nếu n = 0;

Từ đó ta có lời giải của bài toán như sau:

```
int     Tong (int  n ) {
```

```
        if (n ==0 ) return(0); //Điều kiện dừng
        else return(n + Tong(n-1)); //Điều kiện phân tích được
    }
```

Chẳng hạn ta cần tìm tổng của 5 số tự nhiên đầu tiên, khi đó:

```
S  = Tong(5)
   = 5 + Tong(4)
   = 5 + 4 + Tong(3)
   = 5 + 4 + 3 + Tong(2)
   = 5 + 4 + 3 + 2+ Tong(1)
   = 5 + 4 + 3 + 2 + 1 + Tong(0)
   = 5 + 4 + 3 + 2 + 1 + 0
   = 15
```

Ví dụ 2.5. Tìm n!.

Lời giải. Gọi Sn là n!. Khi đó:

- **Bước phân tích**: Sn = n*(n-1)! nếu n>0;
- **Điều kiện dừng**: s0=1 nếu n=0.

Từ đó ta có lời giải của bài toán như sau:

```
long  Giaithua (int  n ) {
        if (n ==0 ) return(1); //Điều kiện dừng
        else return(n *Giaithua(n-1)); //Điều kiện phân tích được
    }
```

Ví dụ 2.6. Tìm ước số chung lớn nhất của a và b bằng phương pháp đệ qui.

Lời giải. Gọi d =USCLN(a,b). Khi đó:

- **Bước phân tích**: nếu b≠0 thì d = USCLN(a, b) = USCLN(b, r), trong đó a =b, b = r = a mod b .
- **Điều kiện dừng**: nếu b = 0 thì a là ước số chung lớn nhất của a và b.

Từ đó ta có lời giải của bài toán như sau:

```
int   USCLN (int a, int b ) {
      if (a ==b ) return(a); //Điều kiện dừng
      else { //Điều kiện phân tích được
            int r = a % b; a = b; b = r;
            return(USCLN(a, b)); //giải bài toán USCLN(a, b)
      }
}
```

2.3. Mô hình thuật toán quay lui (Back-track Algorithm)

Giả sử ta cần xác định bộ $X = (x_1, x_2, .., x_n)$ thỏa mãn một số ràng buộc nào đó. Ứng với mỗi thành phần x_i ta có n_i khả năng cần lựa chọn. Ứng với mỗi khả năng $j \in n_i$ dành cho thành phần x_i ta cần thực hiện:

- *Kiểm tra xem khả năng j có được chấp thuận cho thành phần x_i hay không? Nếu khả năng j được chấp thuận thì ta xác định thành phần x_i theo khả năng j. Nếu i là thành phần cuối cùng (i=n) ta ghi nhận nghiệm của bài toán. Nếu i chưa phải cuối cùng ta xác định thành phần thứ i +1.*
- *Nếu không có khả năng j nào được chấp thuận cho thành phần x_i thì ta quay lại bước trước đó (i-1) để thử lại các khả năng còn lại.*

Thuật toán quay lui được mô tả như sau:

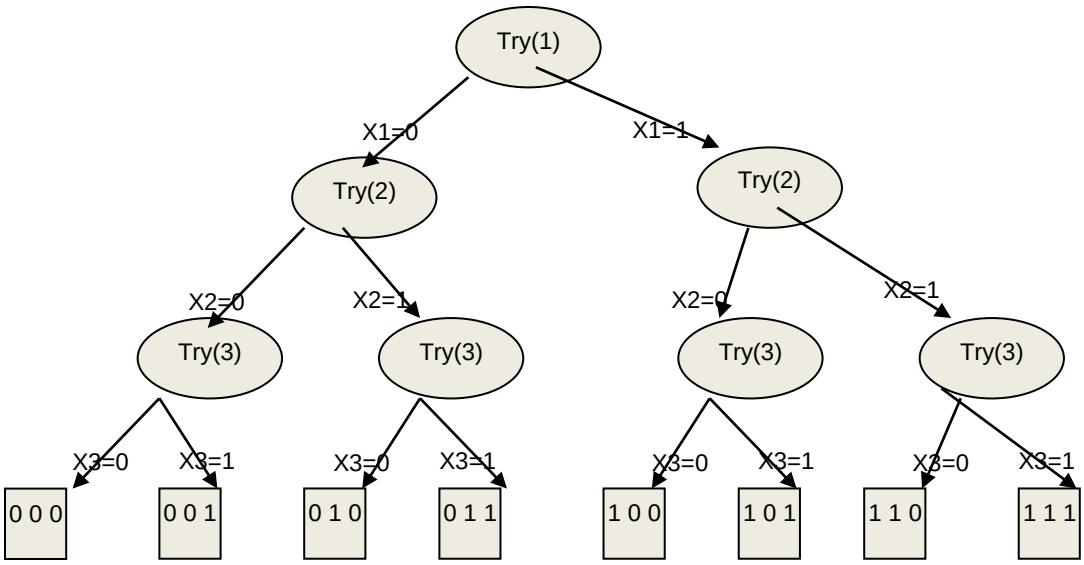
```
Thuật toán Back-Track ( int i ) {
        for ( j =<Khả năng 1>; j <=ni;  j++ ){
                if (<chấp thuận khả năng j>) {
                        X[i] = <khả năng j>;
                        if ( i ==n) Result();
                        else Back-Track(i+1);
        }
}
```

Ví dụ 2.7. Duyệt các xâu nhị phân có độ dài n.

Lời giải. Xâu nhị phân $X = (x_1, x_2, .., x_n) | x_i = 0, 1$. Mỗi $x_i \in X$ có hai lựa chọn $x_i = 0, 1$. Cả hai giá trị này đều được chấp thuận mà không cần có thêm bất kỳ điều kiện gì. Thuật toán được mô tả như sau:

```
        void Try ( int i ) {
                for (int j =0; j<=1; j++){
                        X[i] = j;
                        if ( i ==n) Result();
                        else Try (i+1);
                }
        }
```

Khi đó, việc duyệt các xâu nhị phân có độ dài n ta chỉ cần gọi đến thủ tục Try(1). Cây quay lui được mô tả như Hình 2.1 dưới đây.



Hình 2.1. *Duyệt các xâu nhị phân độ dài 3*

Chương trình duyệt các xâu nhị phân có độ dài n bằng thuật toán quay lui được thể hiện như dưới đây.

```
#include <iostream>
#include <iomanip>
#define MAX 100
using namespace std;
int X[MAX], n, dem=0;
void Init(){ //thiết lập độ dài xâu nhị phân
        cout<<"\n Nhập n="; cin>>n;
}
void Result(void){ //In ra xâu nhị phân X[] = x1, x2,.., xn
        cout<<"\n Kết quả "<<++dem<<":";
        for(int i =1; i<=n; i++)
                cout<<X[i]<<setw(3);
        }
void Try(int i){ //thuật toán quay lui
        for (int j=0; j<=1; j++){ //duyệt các khả năng j dành cho xi
                X[i]=j; //thiết lập thành phần xi là j
                if(i==n) //nếu i là thành phần cuối cùng
                        Result(); // ta đưa ra kết quả
                else //trong trường hợp khác
                        Try(i+1); //ta xác định tiếp thành phần xi+1
        }
}
```

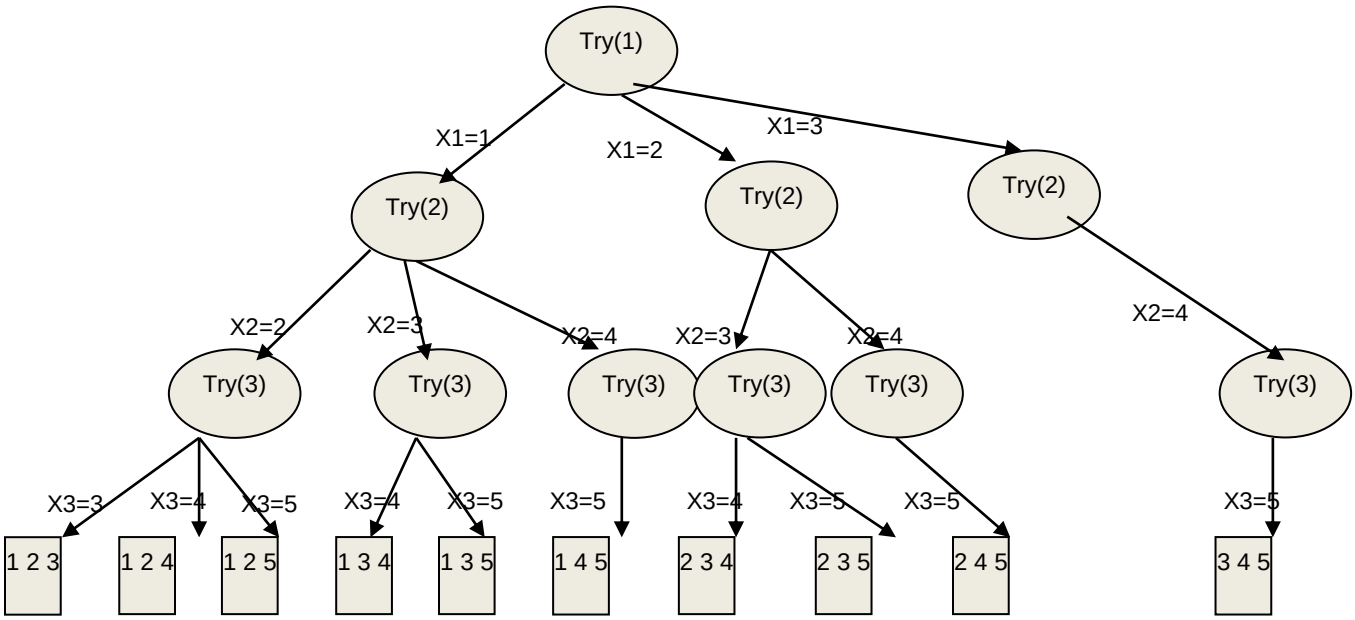
```
int main(void){       Init(); Try(1);}
```

Ví dụ 2.8. Duyệt các tập con K phần tử của 1, 2, .., N.

Lời giải. Mỗi tập con K phần tử $X = (x_1, x_2,..,x_K)$ là bộ không tính đến thứ tự K phần tử của 1, 2, .., N. Mỗi $x_i \in X$ có N-K+i lựa chọn. Các giá trị này đều được chấp thuận mà không cần có thêm bất kỳ điều kiện gì. Thuật toán được mô tả như sau:

```
void Try ( int i ) {
    for (int j =X[i-1]+1; j<=N-K+ i; j++){
        X[i] = j;
        if ( i ==K) Result();
        else Try (i+1);
    }
}
```

Khi đó, việc duyệt các tập con K phần tử của 1, 2, .., N ta chỉ cần gọi đến thủ tục Try(1). Cây quay lui được mô tả như hình dưới đây.



Hình 2.2. *Duyệt các tập con 3 phần tử của 1, 2, 3, 4, 5.*

Chương trình liệt kê các tập con k phần tử của 1, 2, ..,n được thể hiện như sau.

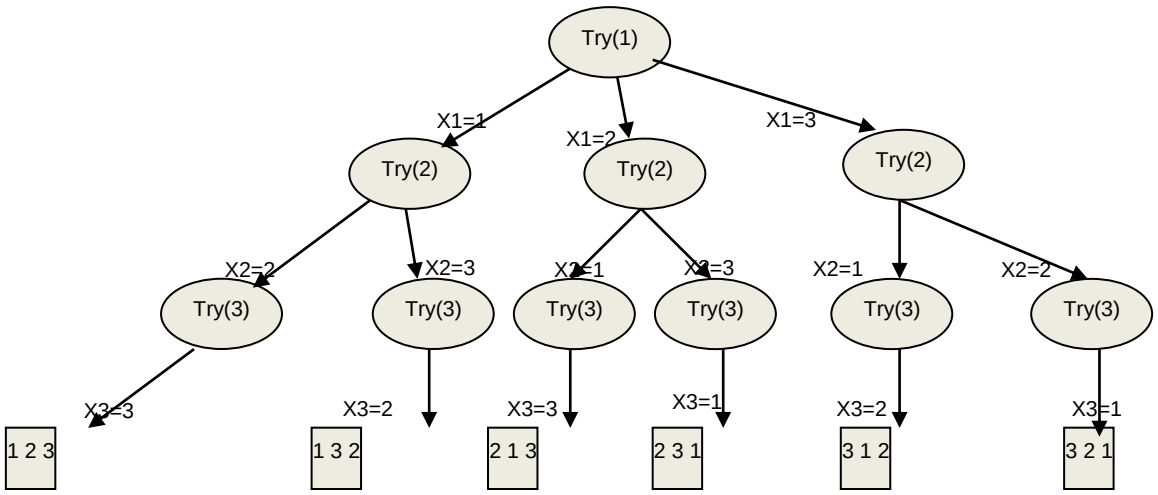
```
#include <iostream>
#include <iomanip>
#define MAX 100
using namespace std;
int X[MAX], n, k, dem=0;
void Init(){//thiết lập giá trị cho n, k
        cout<<"\n Nhập n, k: "; cin>>n>>k;
}
void Result(void){    cout<<"\n Kết quả "<<++dem<<":";//đưa ra kết quả
        for(int i =1; i<=k; i++)    cout<<X[i]<<setw(3);
}

void Try(int i){//thuật toán quay lui
        for (int j=X[i-1]+1; j<=n-k+i; j++){ //duyệt trên tập khả năng dành cho x_i
                X[i]=j; //thiết lập thành phần x_i là j
                if(i==k) //nếu x_i đã là thành phẩn cuối
                        Result(); //ta đưa ra kết quả
                else //trong trường hợp khác
                        Try(i+1); //ta đi xác định thành phần thứ x_i+1
        }
}

int main(void){
        Init();
        X[0] =0 ;
        Try(1);
}
```

Ví dụ 2.9. Duyệt các hoán vị của 1, 2, .., N.

Lời giải. Mỗi hoán vị $X = (x_1, x_2,..,x_K)$ là bộ có tính đến thứ tự của 1, 2, .., N. Mỗi $x_i \in X$ có N lựa chọn. Khi $x_i = j$ được lựa chọn thì giá trị này sẽ không được chấp thuận cho các thành phần còn lại. Để ghi nhận điều này, ta sử dụng mảng chuaxet[] gồm N phần tử. Nếu chuaxet[i] = True điều đó có nghĩa giá trị i được chấp thuận và chuaxet[i] = False tương ứng với giá trị i không được phép sử dụng. Thuật toán được mô tả như sau:

```
void Try ( int i ) {
    for (int j =1; j<=N; j++){
        if (chuaxet[j] ) {
            X[i] = j;chuaxet[j] = False;
            if ( i ==N) Result();
            else Try (i+1);
            Chuaxet[j] = True;
        }
    }
}
```

Khi đó, việc duyệt các hoán vị của 1, 2, .., N ta chỉ cần gọi đến thủ tục Try(1). Cây quay lui được mô tả như hình dưới đây.



Hình 2.3. Duyệt các hoán vị của 1, 2, 3.

Chương trình liệt kê tất cả các hoán vị của 1, 2, .., n được thể hiện như sau:

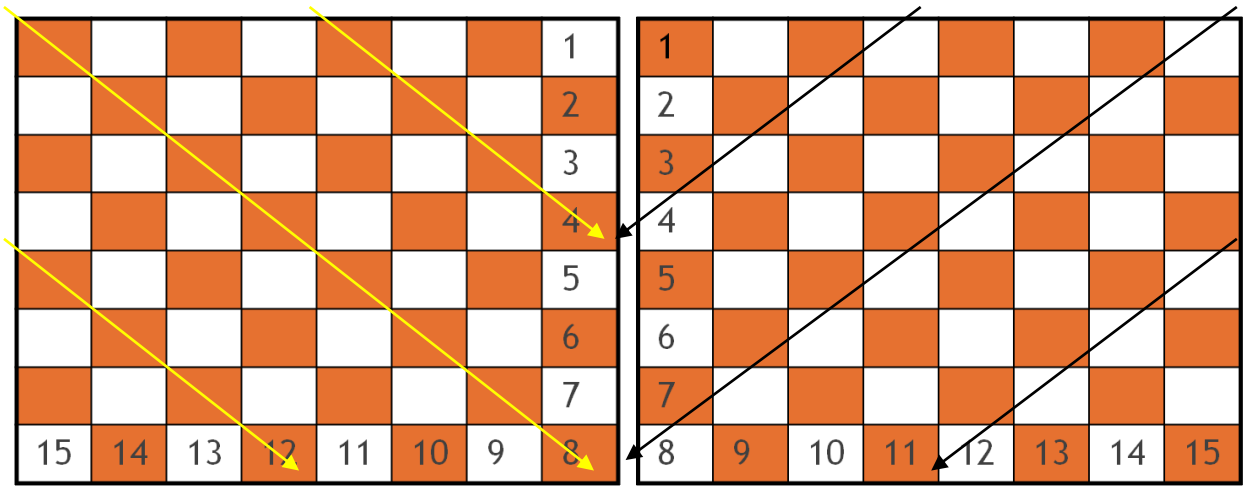
```
#include <iostream>
#include <iomanip>
#define MAX 100
using namespace std;
int X[MAX], n, dem=0;
bool  chuaxet[MAX];
```

```
void Init(){//thiết lập giá trị cho n
        cout<<"\n Nhập n="; cin>>n;
        for(int i=1; i<=n; i++) //thiết lập giá trị cho mảng chuaxet[]
                chuaxet[i]=true;
}
void Result(void){ //Đưa ra hoán vị hiện tại
        cout<<"\n Kết quả "<<++dem<<":";
        for(int i =1; i<=n; i++) cout<<X[i]<<setw(3);
}
void Try(int i){ //thuật toán quay lui duyệt các hoán vị của 1, 2, .., n.
        for (int j=1; j<=n; j++){ //duyệt các khả năng j cho thành phần x_i
                if(chuaxet[j]){ //nếu khả năng j đúng chưa được dùng đến
                        X[i]=j; //thiết lập  thành phần x_i là j
                        chuaxet[j]=false; //thiết lập chuaxet[j] đã được dùng
                        if(i==n)  //nếu x_i đã là thành phần cuối cùng
                                Result();//ta đưa ra kết quả
                        else ///trong trường hợp khác
                                Try(i+1); //ta xác định tiếp thành phần thứ i+1
                        chuaxet[j]=true; //nhớ hoàn trả lại giá trị cho chuaxet[j]
                }
        }
}
int main(void){
        Init(); Try(1);
}
```

Ví dụ 2.10. Bài toán N quân hậu. Trên bàn cờ kích cỡ N×N, hãy đặt N quân hậu mỗi quân trên 1 hàng sao cho tất cả các quân hậu đều không ăn được lẫn nhau.

Lời giải. Gọi X =(x_1, x_2,..,x_n) là một hoán vị của 1, 2, .., n.. Khi đó, x_i = j được hiểu là quân hậu hàng thứ i đặt ở cột j. Để các quân hậu khác không thể ăn được, quân hậu thứ i cần không được lấy trùng với bất kỳ cột nào, không được cùng đường chéo xuôi, không được cùng trên đường chéo ngược. Ta có n cột Cot = (c_1,..c_n), có Xuoi[2*n-1] đường chéo xuôi, Nguoc[2*n-1] đường chéo ngược. Quân hậu ở hàng *i* được đặt vào cột *j* nếu A[j] = True (chưa có quân hậu nào án ngữ cột j), Xuoi[i-j+n] = True (chưa có quân hậu nào án ngữ đường chéo i-j+n), Nguoc[i + j -1] = True (chưa có quân hậu nào án ngữ đường chéo ngược i + j-1).

Đường chéo xuôi Xuoi[i-j+n] Đường chéo ngược Nguoc[i+j-1]



Hình 2.4. *Mô tả các đường chéo, xuôi đường chép ngược*

Thuật toán quay lui giải bài toán n quân hậu được mô tả như dưới đây.

```
void Try (int    i){
        for(int j=1; j<=n; j++){
                if( Cot[j] && Xuoi[ i – j + n ] && Nguoc[i + j -1]){
                        X[i] =j;  Cot[j]=FALSE;
                        Xuoi[ i - j + n]=FALSE;
                        Nguoc[ i + j - 1]=FALSE;
                        if(i==n) Result();
                        else Try(i+1);
                        Cot[j] = TRUE;
                        Xuoi[ i - j + n] = TRUE;
                        Nguoc[ i + j - 1] = TRUE;
                }
        }
}
```

Chương trình giải bài toán n quân hậu được thể hiện như dưới đây.

```
#include <iostream>
#include <iomanip>
#define MAX 100
using namespace std;
int X[MAX], n, dem=0;
bool  COT[MAX], DCXUOI[MAX], DCNGUOC[MAX];;
void Init(){ //thiết lập kích cỡ bàn cờ
        cout<<"\n Nhap n="; cin>>n;
        for(int i=1; i<=n; i++){ //thiết lập tất cả các cột đều chưa bị án ngữ
```

```
            COT[i]=true;
        }
        for(int i=1; i<2*n; i++){ //thiết lập các đường chéo
            DCXUOI[i]=true; // đường chéo xuôi chưa bị án ngữ
            DCNGUOC[i]=true; // đường chéo ngược chưa bị án ngữ
        }
}
void Result(void){ //đưa ra một phương án
        cout<<"\n Kết quả "<<++dem<<":";
        for(int i =1; i<=n; i++)
            cout<<X[i]<<setw(3);
}
void Try(int i){ //đây là thuật toán quay lui
        for (int j=1; j<=n; j++){ //duyệt các khả năng j đặt quân hậu vào hàng i
            if( COT[j] && DCXUOI[i-j+n]&& DCNGUOC[i+j-1]){
            //nếu đúng cột j, đường chéo xuôi i-j +n, đường chéo ngược i+j-1
            // chưa bị án ngữ
                X[i]=j; //ta đặt được quân hậu hàng i vào vột j
                COT[j] = false; // cột j đã bị án ngữ
                DCXUOI[i-j+n]=false; // đường chéo xuôi i-j+n bị án ngữ
                DCNGUOC[i+j-1]=false;//đường chéo ngược i+j-1 bị án ngữ
                if(i==n) // nếu đây là quân hậu hàng n
                    Result();// ta đưa ra phương án hiện tại
                else //trong trường hợp khác
                    Try(i+1); // ta đặt tiếp quân hậu hành i+1
                COT[j] = true; // nhớ trả lại giá trị cột j
                DCXUOI[i-j+n]=true; //trả lại giá trị đường chéo xuôi
                DCNGUOC[i+j-1]=true; // trả lại giá trị đường chéo ngược
            }
        }
}
int main(void){ Init(); Try(1); }
```

2.4. Mô hình thuật toán tham lam (Greedy Algorithm)

Thuật toán tham lam (Greedy Algorithm) xây dựng một chiến lược "*tham từng miếng*", miếng dễ tham nhất thường là một giải pháp tối ưu cục bộ nhưng lại luôn có mặt trong cấu trúc tối ưu toàn cục. Tiếp tục phát triển chiến lược "*tham từng miếng*" cho các

bước tiếp theo để đạt được kết quả tối ưu toàn cục. Tóm lại, thuật toán tham lam dùng để giải lớp các bai toán tối ưu thỏa mãn hai điều kiện:

- Ở mỗi bước ta luôn tạo ra một lựa chọn tốt nhất tại thời điểm đó.
- Việc liên kết lại kết quả mỗi bước sẽ cho ta kết quả tối ưu toàn cục.

Tổng quát, thuật toán Greedy bao gồm 5 thành phần chính:

1) Một tập các ứng viên (*candidate members*) mà giải pháp có thể tham lam.
2) Một hàm lựa chọn (*selection fuction*) để chọn ứng viên tốt nhất cho giải pháp tham lam cục bộ.
3) Một hàm thực thi (*feasibility function*) được sử dụng để quyết định xem một ứng viên có được dùng để xây dựng lời giải hay không?
4) Một hàm mục tiêu (objective function) dùng để xác định giá trị của lời giải hoặc một phần của lời giải.
5) Một hàm giải pháp (solution function) dùng để xác định khi nào giải pháp hoàn chỉnh.

Khi sử dụng giải pháp tham lam, hai thành phần quyết định nhất tới quyết định tham lam đó là:

Lựa chọn tính chất để tham. Khi chưa tìm được lời giải tối ưu toàn cục nhưng ta biết chắc chắn một giải pháp tối ưu cục bộ dựa vào tính chất cụ thể của mỗi bài toán. Bài toán con tiếp theo cũng thực hiện với tính chất như vậy cho đến khi đạt được giải pháp tối ưu toàn cục.

Lựa chọn cấu trúc con tối ưu. Một bài toán được gọi là "có cấu trúc tối ưu" nếu một lời giải tối ưu của bài toán con nằm trong lời giải tối ưu của bài toán lớn hơn. Điều này cho phép ta xác định từng cấu trúc con tối ưu cho mỗi bài toán con, sau đó phát triển các bài toán con cho đến bài toán con cuối cùng.

Ví dụ 2.11. Bài toán lựa chọn hành động (*Activity Selection Problem*). Lựa chọn hành động là bài toán tối ưu tổ hợp điển hình quan tâm đến việc lựa chọn các hành động không mâu thuẫn nhau trong các khung thời gian cho trước. Bài toán được phát biểu như sau:

Cho tập gồm n hành động, mỗi hành động được biểu diễn như bộ đôi thời gian bắt đầu s_i và thời gian kết thúc f_i (i=1, 2, .., n). Bài toán đặt ra là hãy lựa chọn nhiều nhất các hành động có thể thực hiện bởi một máy hoặc một người mà không xảy ra mâu thuẫn. Giả sử mỗi hành động chỉ thực hiện đơn lẻ tại một thời điểm.

Input:

- Số lượng hành động: 6
- Thời gian bắt đầu Start []= { 1, 3, 0, 5, 8, 5}
- Thời gian kết thúc Finish[]= { 2, 4, 6, 7, 9, 9}

Output: Số lượng lớn nhất các hành động có thể thực hiện bởi một người.

OPT[] = {1, 2, 4, 5 }

Lời giải. Rõ ràng, hệ sẽ xảy ra mâu thuẫn khi tồn tại hai hành động kế tiếp nhau *i* và *i+1* có thời gian kết thúc của hành động *i* lớn hơn thời gian bắt đầu của hành động *i+1*. Nói cách khác hệ mâu thuẫn khi Finish[i]>Start[i+1] (1<i<n). Vấn đề còn lại là làm thế nào ta có thể lựa chọn được tập lớn nhất các hành động không tồn tại mâu thuẫn. Để giải quyết điều này, ta có thể sử dụng thuật toán sinh hoặc thuật toán quay lui thể duyệt trên tập tất cả các tập con các hành động, sau đó lựa chọn tập con có số lượng hành động lớn nhất và không xảy ra mâu thuẫn. Tuy nhiên lời giải này cho ta độ phức tạp hàm mũ (2^n) nên lời giải này thực tế không giải được bằng máy tính.

Để khắc phục điều này ta có thể xây dựng thuật toán tham lam một cách hiệu quả dựa vào những nhận xét trực quan như sau:

- Trường hợp xấu nhất xảy ra khi hệ chỉ có thể đáp ứng được nhiều nhất là một hành động, hễ cứ thêm một hành động hệ sẽ xảy ra mâu thuẫn. Rõ ràng, lựa chọn tối ưu nhất trong trường hợp này là chọn hành động kết thúc với thời gian sớm nhất.
- Nếu hệ có thể đáp ứng nhiều hơn một hành động, thì lựa chọn tốt tiếp theo chính là kết nạp hành động có thời gian bắt đầu lớn hơn thời gian kết thúc của hành động trước đó và có thời gian kết thúc nhỏ nhất trong số các hành động còn lại. Cứ tiếp tục làm như vậy ta sẽ có được giải pháp tối ưu toàn cục.

Thuật toán tham lam dùng để giải bài toán Activity Selectio được thể hiện như sau:

```
Algorithm Greedy-Activities- Selection( N,  S[], F[]  ):
Input:
        • N là số lượng hành động (công việc).
        • S[] thời gian bắt đầu mỗi hành động.
        • F[] thời gian kết thúc mỗi hành động.
Ouput:
        • Danh sách thực thi nhiều nhất các hành động.
Actions:
    Bước 1 (sắp xếp). Sắp xếp các hành động theo thứ tự
    tăng dần của thời gian kết thúc.
    Bước 2 (Khởi tạo) Lựa chọn hành động đầu tiên làm
    phương án tối ưu ( OPT=1). N = N\ {i};
    Bước 3 (Lặp).
        for each activities j∈N do {
            if ( S[j] >= F[i] ) {
                OPT = OPT∪j; i = j; N = N\{i}
            }
        }
    Bước 4 (Trả lại kết quả).
        Return (OPT)
EndActions.
```

Hình 2.5. Thuật toán tham lam giải bài toán Activity Selection

Kiểm nghiệm thuật toán:

Ví dụ. Thuật toán Greedy-ActivitiesN-Selection(int N, int S[], int F[]):

Input:

- Số lượng hành động n = 8.
- Thời gian bắt đầu S[] = {1, 3, 0, 5, 8, 5, 9, 14}.
- Thời gian kết thúc F[] = {2, 4, 6, 7, 9, 9, 12, 18}.

Output:

- OPT = { Tập nhiều nhất các hành động có thể thực hiện bởi một máy}.

Phương pháp tiến hành::

Bước	i= ? j =?	(S[j] >=F[i]) ? i=?	OPT=?
			OPT = OPT∪{1}.
1	i=1; j=2	(3>=2): Yes; i=2.	OPT = OPT∪{2}.
2	i=2; j=3;	(0>=4): No; i=2.	OPT = OPT∪ϕ.
3	i=2; j=4;	(5>=4): Yes; i=4.	OPT = OPT∪{4}.
4	i=4; j=5;	(8>=7): Yes; i=5.	OPT = OPT∪{5}.
5	i=5; j=6;	(5>=9): No; i=5.	OPT = OPT∪ϕ.
6	i=5; j=7;	(9>=9): Yes; i=7.	OPT = OPT∪{7}.
7	i=7; j=8;	(14>=12): Yes; i=8.	OPT = OPT∪{8}.
OPT = { 1, 2, 4, 5, 7, 8}			

Cài đặt thuật toán: thuật toán được cài đặt với khuôn dạng dữ liệu vào trong file data.in và kết quả ra trong file ketqua.out như sau:

File data.in:

- Dòng đầu tiên ghi lại số tự nhiên N là số lượng hành động.
- N dòng kế tiếp, mỗi dòng ghi lại bộ đôi Start[i], Finish[i] tương ứng với thời gian bắt đầu và thời gian kết thúc mỗi hành động. Hai số được viết cách nhau một vài khoảng trống.

File ketqua.out:

- Dòng đầu ghi lại số lượng lớn nhất các hành động.
- Dòng kế tiếp ghi lại các hành động được lựa chọn. Các hành động được viết cách nhau một vài khoảng trống.

Ví dụ về file data.in và ketqua.out:

data.in	ketqua.out
6	4

```
1       2               1       2       4       5
3       4
0       6
5       7
8       9
5       9
```

Chương trình giải bài toán Activity Selection như dưới đây:

```
#include <iostream>
#include <fstream>
#include <iomanip>
#define MAX 1000
using namespace std;
int Start[MAX], Finish[MAX], n, XOPT[MAX], dem=0;
//đọc dữ liêu từ file data.in
void Read_Data(){
        ifstream fp("data.in"); fp>>n;
        for (int i=1; i<=n; i++){
                fp>>Start[i];
                fp>>Finish[i];
                XOPT[i]=false;
        }
        fp.close();
}
//Sắp xếp tăng dần theo thời gian kết thúc các hành động
void Sapxep(void) {
        for(int i=1; i<=n-1; i++){
                for(int j=i+1; j<=n; j++){
                        if(Finish[i]> Finish[j]){
                                int t = Finish[i]; Finish[i]=Finish[j];Finish[i]=t;
                                t = Start[i];Start[i]=Start[j];Start[i]=t;
                        }
                }
        }
}
//đưa ra kết quả tối ưu
void Result(void){
        ofstream fp("ketqua.out");
```

```
        fp<<dem<<endl;
        for(int i=1; i<=n; i++){
                if(XOPT[i])
                        fp<<i<<setw(3);
        }
}
void Greedy_Solution(void){ //Thuật toán tham lam
        Read_Data();//đọc dữ liêu từ file data.in
        Sapxep();//Bước 1: Sắp xếp
        int i =1; XOPT[i]=true; dem=1; //Bước 2. Khởi tạo
        for(int j=2; j<=n; j++){//Bước 3: lặp
                if(Finish[i]<=Start[j]){
                        dem++; i = j; XOPT[i]=true;

                }
        }
        Result();//Bước 4. Trả lại kết quả
}
int main(void){
        Greedy_Solution();
}
```

Ví dụ 2.12. *Sắp đặt lại các kỹ tự giống nhau trong xâu ký tự.* Cho xâu ký tự s[] có độ dài n và số tự nhiên d. Hãy sắp đặt lại các ký tự trong xâu s[] sao cho các ký tự giống nhau đều cách nhau một khoảng là d. Nếu bài toán có nhiều nghiệm, hãy đưa ra một cách sắp đặt đầu tiên tìm được. Nếu bài toán không có lời giải hãy đưa ra thông báo "Vô nghiệm".

Ví dụ.

Input:

- Xâu ký tự S[] ="ABB";
- Khoảng cách d = 2.

Output: BAB

Input:

- Xâu ký tự S[] ="AAA";
- Khoảng cách d = 2.

Output: Vô nghiệm.

Input:

- Xâu ký tự S[] ="GEEKSFORGEEKS";

- Khoảng cách d = 3.

Output: EGKEGKESFESFO

Lời giải. Dễ dàng nhận thấy, trường hợp xấu nhất xảy ra khi ta không có phương án sắp đặt lại các ký tự giống nhau trong xâu s[] cách nhau một khoảng d. Trường hợp này dễ dàng đoán nhận được bằng cách chọn ký tự $x \in s$ có số lần xuất hiện nhiều lần nhất trong s[], sau đó dãn các ký tự *x* cách nhau một khoảng *d* bắt đầu tại vị trí $i = 0$. Nếu (i+k*d)≥n thì ta có kết luận ngay bài toán vô nghiệm (*k* số lần xuất hiện ký tự *x*). Trong trường hợp, (i+k*d)<n thì ký tự *x* sắp đặt được với khoảng cách d ta chỉ cần đặt k ký tự *x*, mỗi ký tự cách nhau một khoảng là *d*. Quá trình sắp đặt được tiến hành với ký tự xuất hiện nhiều lần nhất còn lại cho đến ký tự cuối cùng. Thuật toán tham lam dùng giải quyết bài toán được thể hiện trong Hình 2.6 dưới đây.

Thuật toán Greedy-Arrang-String (S[], d):
Input:
- Xâu ký tự S[].
- Khoảng cách giữa các ký tự d.

Output: Xâu ký tự được sắp đặt lại thỏa mãn yêu cầu bài toán.
Formats : Greedy-Arrang-String(S, d, KQ);
Actions:

Bước 1. Tìm Freq[] là số lần xuất hiện mỗi ký tự trong xâu.
Bước 2. Sắp xếp theo thứ tự giảm dần theo số xuất hiện ký tự.
Bước 3. (Lặp).

```
i = 0; k =<Số lượng ký tự trong Freq[]>;
While ( (i < k ) {
        p = Max(Freq); //Chọn ký tự xuất hiện nhiều lần nhất.
        For ( t = 0; t<p; t++ ) // điền các ký tự i, i+d, i+2d, ..,i + pd
                if (i+(t*d)>n ) { < Không có lời giải>; return;>
                KQ[i + (t*d)]  = Freq[i].kytu;
        }
        i++;
}
```

Bước 4(Trả lại kết quả): Return(KQ);
EndActions.

Hình 2.6. Thuật toán tham lam giải quyết bài toán.

Thử nghiệm thuật toán:

Greedy-Arrang-String (S[], d):Input : S[] = "GEEKSFORGEEKS; d= 3.

Bước 1. Tìm tập ký tự và số lần xuất hiện mỗi ký tự.

Freq[]	
G	2
E	4
K	2
S	2
F	1
O	1
R	1

Sắp xếp theo thứ tự giảm dần của số lần xuất hiện.

Bước 2. Sắp xếp theo thứ tự giảm dần số lần xuất hiện.

Freq[]	
E	4
G	2
K	2
S	2
F	1
O	1
R	1

Bước 3. Lặp.

	KQ[I + p*d]												
I =0	E			E			E			E			
i=1	E	G		E	G		E			E			
i=2	E	G	K	E	G	K	E			E			
i=3	E	G	K	E	G	K	E	S		E	S		
i=4	E	G	K	E	G	K	E	S	F	E	S		
i=5	E	G	K	E	G	K	E	S	F	E	S	O	
i=6	E	G	K	E	G	K	E	S	F	E	S	O	R

Cài đặt thuật toán: thuật toán tham lam giải quyết bài toán được cài đặt như dưới đây.

```
#include <iostream>
#include <cstring>
#include <iomanip>
#define MAX 1000
using namespace std;
typedef struct Tanxuat{ //định nghĩa ký tự và số lần xuất hiện ký tự trong xâu S
        char kytu;
        int solan;
};
int Search( Tanxuat X[], int n, char t){ //xác định vị trí của t trong tập ký tự X[]
        for(int i=0; i<=n; i++){
                if(X[i].kytu==t)
                   return i;
        }
        return (n+1);
}
int Tach_Kytu(char S[], Tanxuat X[], int n){ //tìm số ký tự và số lần xuất hiện
        int k=strlen(S);
        for(int i=0;i<k; i++){
                int p = Search(X,n,S[i]);
                if(p<=n)X[p].solan++;
```

```
            else {n=p;      X[p].kytu = S[i];      X[p].solan = 1; }
        }
        return n;
}
void Sort(Tanxuat X[], int n){ //sắp xếp giảm dần theo số lần xuất hiện ký tự
        Tanxuat t;
        for(int i=0; i<n; i++){
            for(int j=i+1; j<=n; j++){
                if(X[i].solan<X[j].solan){
                    t = X[i]; X[i]=X[j]; X[j]=t;
                }
            }
        }
}
void Greedy_Solution(void){ //giải pháp tham lam
        char S[MAX]; cout<<"Nhap xau S:";cin>>S; //thiết lập S
        int d; cout<<"Khoang cach d:";cin>>d;//thiết lập khoảng cách d
        int m=-1, n = strlen(S), chuaxet[MAX]; Tanxuat X[255];
        char STR[MAX]; //xâu kết quả
        for (int i=0; i<n; i++)//dùng để xác định vị trí ký tự cần đặt
            chuaxet[i]=true;
        //Bước 1: tìm tập ký tự và số lần xuất hiện mỗi ký tự
        m = Tach_Kytu(S, X, m);
        //Bước 2: sắp xếp giảm dần theo số lần xuất hiện
        Sort(X, m);
        //Bước 3: lặp
        for(int i=0; i<=m; i++){//tham lam trên tập ký tự
            int k = X[i].solan; //lấy ký tự xuất hiện nhiều lần nhất
            int t =0; //tìm vị trí bắt đầu đặt ký tự
            while (!chuaxet[t]) t++;
            for(int q = 0; q<k; q++){//duyệt theo số lần xuất hiện
                if((t + q*d)>=n){//nếu dãn khoảng cách d vượt quá độ dài S
                    cout<<"No Solution";
                    return; //kết luận không có giải pháp
                }
                STR[t+q*d]=X[i].kytu; //đặt tại vị trí t + q*d là OK
                chuaxet[t+q*d]=false;//đánh dấu vị trí này đã được đặt
```

```
            }
        }
        STR[n]='\0'; //kết thúc xâu kết quả
        cout<<STR;//đưa ra phương án
}
int main(void){
        Greedy_Solution();
}
```

2.5. Mô hình thuật toán chia và trị (Devide and Conquer Algorithm)

Thuật toán chia để trị (Devide and Conquer) dùng để giải lớp các bài toán có thể thực hiện được thông qua ba bước:

- **Bước chia (Devide)**. Chia bài toán lớn thành những bài toán con có cùng kiểu với bài toán lớn.
- **Bước trị (Conquer)**. Giải các bài toán con đã chia ở bước trước đó. Thông thường các bài toán con chỉ khác nhau về dữ liệu vào nên ta có thể thực hiện bằng một thủ tục đệ qui.
- **Bước tổng hợp (Combine)**. Tổng hợp lại kết quả của các bài toán con để nhận được kết quả của bài toán lớn.

Ví dụ 2.13. Nâng *x* lên lũy thừa *n*.

Lời giải.

Bước chia. Ta có $x^n = \begin{cases} x^{n/2}.x^{n/2} & nếu\ n\ chẵn \\ x.x^{n/2}.x^{n/2} & nếu\ x\ lẻ \end{cases}$.

Bước trị. Tính toán $x^{n/2}$ trong trường hợp x chẵn, $x.x^{n/2}$ trong trường hợp *x* lẻ.

Bước tổng hợp. Kết quả chính là $x^{n/2}.x^{n/2}$ trong trường hợp x chẵn và $x.x^{n/2}.x^{n/2}$ trong trường hợp x lẻ.

Chương trình nâng x lên lũy thừa n bằng kỹ thuật chia và trị được thể hiện như sau:

```
#include <iostream>
#include <iomanip>
using namespace std;
int power(int x, unsigned int n){
        if( n == 0)
                return 1;
        else if (n%2 == 0)
                return power(x, n/2)*power(x, n/2);
        else
                return x*power(x, n/2)*power(x, n/2);
}
```

```
int main(){
        int x = 2;
        unsigned int n = 5;
        cout<<power(x, n)<<setw(3);
}
```

Thử nghiệm thuật toán: tính S = power(2, 5)

S = power(2, 5)
= 2* power(2, 2)* power(2, 2)
= 2* power(2, 1)* power(2, 1)* power(2, 1)* power(2, 1)
= 2* 2* power(0, 1)*2* power(2, 0)* 2*power(2, 0)* 2*power(2, 0)
= 2*2*2*2*2
=32

Thuật toán trên có độ phức tạp là O(n). Thuật toán trình bày dưới đâu có độ phức tạp O(log(n)).

```
#include <iostream>
#include <iomanip>
using namespace std;
int power(int x, unsigned int n) {
        int temp;
        if( n == 0)
                return 1;
        temp = power(x, n/2);
        if (n%2 == 0)
                return temp*temp;
        else
                return x*temp*temp;
}
int main(){
        int x = 2;
        unsigned int n = 5;
        cout<<power(x, n)<<setw(3);
}
```

2.6. Mô hình thuật toán nhánh cận (Branch and Bound Algorithm)

Thuật toán nhánh cận thường được dùng trong việc giải quyết các bài toán tối ưu tổ hợp. Bài toán được phát biểu dưới dạng sau:

Tìm min { $f(X) : X \in D$ }, với $D = \{ X = (x_1, x_2, .., x_n) \in A_1 \times A_2 \times A_n$: *X thỏa mãn tính chất P* }.

- $X \in D$ được gọi là một phương án của bài toán.
- Hàm $f(X)$ được gọi là hàm mục tiêu của bài toán.
- Miền D được gọi là tập phương án của bài toán.

Để giải quyết bài toán trên, ta có thể dùng thuật toán quay lui duyệt các phần tử $X \in D$, phần tử X* làm cho F(X*) đạt giá trị nhỏ nhất (lớn nhất) là phương án tối ưu của bài toán. Thuật toán nhánh cận có thể giải được bài toán đặt ra nếu ta xây dựng được một hàm g xác định trên tất cả phương án bộ phận cấp k của bài toán sao cho:

$$g(a_1, a_2, .., a_k) \leq \min \{ f(X): X \in D, x_i = a_i, i = 1, 2, .., k\}$$

Nói cách khác, giá trị của hàm g tại phương án bộ phận cấp k là $g(a_1, a_2, .., a_k)$ không vượt quá giá trị nhỏ nhất của hàm mục tiêu trên tập con các phương án.

$$D(a_1, a_2, .., a_k) = \{ X \in D: x_i = a_i, i = 1, 2, .., k\}$$

Giá trị của hàm $g(a_1, a_2, .., a_k)$ là cận dưới của hàm mục tiêu trên tập $D(a_1, a_2, .., a_k)$. Hàm g được gọi là hàm cận dưới, $g(a_1, a_2, .., a_k)$ gọi là cận dưới của tập $D(a_1, a_2, .., a_k)$.

Giả sử ta đã có hàm g. Để giảm bớt khối lượng duyệt trên tập phương án, bằng thuật toán quay lui ta xác định được X^* là phương án làm cho hàm mục tiêu có giá trị nhỏ nhất trong số các phương án tìm được $f^* = f(X^*)$. Ta gọi X^* là phương án tốt nhất hiện có, f^* là kỷ lục hiện tại.

Nếu

$$f^* < g(a_1, a_2, .., a_k)$$

thì

$$f^* < g(a_1, a_2, .., a_k) \leq \min \{ f(X): X \in D, x_i = a_i, i = 1, 2, .., k\}.$$

Điều này có nghĩa tập $D(a_1, a_2, .., a_k)$ chắc chắn không chứa phương án tối ưu. Trong trường hợp này ta không cần phải triển khai phương án bộ phận $(a_1, a_2, .., a_k)$. Tập $D(a_1, a_2, .. a_k)$ cũng bị loại bỏ khỏi quá trình duyệt. Nhờ đó, số các phương án cần duyệt nhỏ đi trong quá trình tìm kiếm. Thuật toán nhánh cận tổng quát được mô tả chi tiết trong Hình 2.7.

```
Thuật toán Branch_And_Bound (k) {
        for a_k∈A_k do {
                if (<chấp nhận a_k>){
                        x_k = a_k;
                        if ( k==n )
                                <Cập nhật kỷ lục>;
                        else if ( g(a_1, a_2,..,a_k) ≤ f*)
                                 Branch_And_Bound (k+1)    ;
                }
        }
}
```

Hình 2.7. Thuật toán Branch-And-Bound

Ví dụ 2.14. *Giải bài toán cái túi bằng thuật toán nhánh cận*. Một nhà thám hiểm cần đem theo một cái túi trọng lượng không quá *b*. Có *n* đồ vật cần đem theo. Đồ vật thứ *i* có trọng lượng a_i và giá trị sử dụng c_i. Hãy tìm cách đưa đồ vật vào túi cho nhà thám hiểm sao cho tổng giá trị sử dụng các đồ vật trong túi là lớn nhất.

Lời giải. Bạn đọc tự tìm hiểu lời giải của bài toán trong các tài liệu liên quan. Dưới đây là chương trình cài đặt bài toán với dữ liệu vào trong file caitui.in và kết quả ra trong file ketqua.out theo khuôn dạng:

Caitui.in:

- Dòng đầu tiên ghi lại hai số n và b tương ứng với số lượng đồ vật và trọng lượng túi.
- N dòng kế tiếp, mỗi dòng ghi lại bộ đôi a_i, c_i tương ứng với trọng lượng và giá trị sử dụng đồ vật *i*.

Ketqua.out:

- Dòng đầu tiên ghi lại giá trị tối ưu của bài toán.
- Dòng kế tiếp ghi lại phương án tối ưu của bài toán.

Ví dụ:

Caitui.in	ketqua.out
4 8	15
5 10	1 1 0 0
3 5	

2 3
4 6

Chương trình giải quyết bài toán được thực hiện như dưới đây:

```
#include <iostream>
#include <fstream>
#define MAX 100
using namespace std;
int X[MAX]; // phương án tối ưu của bài toán
int n; //số lượng đồ vật
float b, weight=0; //trọng lượng túi
float cost=0; // giá trị sử dụng phương án hiện tại
int XOPT[MAX]; //phương án tối ưu của bài toán
float FOPT=0; //giá trị tối ưu của bài toán
float A[MAX], C[MAX];//vector trọng lượng và giá trị sử dụng
void Init (void) { //đọc dữ liệu
    ifstream fp("caitui.in"); fp>>n;fp>>b;
    for(int i=1; i<=n; i++){
        fp>>A[i]>>C[i];
    }
    fp.close();
}
void Result(void) {//đưa ra giá trị và phương án tối ưu
    cout<<"Giá trị tối ưu:"<<FOPT<<endl;
    cout<<"Phương án tối ưu:";
    for(int i=1; i<=n; i++)
        cout<<XOPT[i]<<"  ";
}
void Update(void) { //cập nhật phương án tối ưu
    if (cost> FOPT){
        FOPT =cost;
        for (int i=1; i<=n; i++)
            XOPT[i]=X[i];
    }
}
void Back_Track(int i){
    int j, t = (int)((b-weight)/A[i]); //giá trị khởi đầu
    for(int j= t; j>=0; j--){
```

```
                X[i] = j;
                weight = weight+A[i]*X[i]; //trọng lượng phương án bộ phận
                cost = cost + C[i]*X[i]; //giá trị phương án bộ phận
                if (i==n) Update(); //ghi nhận kỷ lục
                else if ( cost + ( C[i+1]*((b- weight)/A[i+1]))>FOPT) //kiểm tra cận
                        Back_Track(i+1);
                weight = weight-A[i]*X[i];
                cost = cost - C[i]*X[i];
        }
}
int main (void ){ //chương trình chính
        Init(); Back_Track(1);Result();
}
```

2.7. Mô hình thuật toán qui hoạch động (Dynamic Programming Algorithm)

Phương pháp qui hoạch động dùng để giải lớp các bài toán thỏa mãn những điều kiện sau:

- Bài toán lớn cần giải có thể phân rã được thành nhiều bài toán con. Trong đó, sự phối hợp lời giải của các bài toán con cho ta lời giải của bài toán lớn. Bài toán con có lời giải đơn giản được gọi là cơ sở của qui hoạch động. Công thức phối hợp nghiệm của các bài toán con để có nghiệm của bài toán lớn được gọi là công thức truy hồi của qui hoạch động.
- Phải có đủ không gian vật lý lưu trữ lời giải các bài toán con (Bảng phương án của qui hoạch động). Vì qui hoạch động đi giải quyết tất cả các bài toán con, do vậy nếu ta không lưu trữ được lời giải các bài toán con thì không thể phối hợp được lời giải giữa các bài toán con.
- Quá trình giải quyết từ bài toán cơ sở (bài toán con) để tìm ra lời giải bài toán lớn phải được thực hiện sau hữu hạn bước dựa trên bảng phương án của qui hoạch động.

Ví dụ 2.15. Tìm số các cách chia số tự nhiên n thành tổng các số tự nhiên nhỏ hơn n. Các cách chia là hoán vị của nhau chỉ được tính là một cách.

Lời giải. Gọi F [m, v] là số cách phân tích số v thành tổng các số nguyên dương nhỏ hơn hoặc bằng m. Khi đó, số các cách chia số v thành tổng các số nhỏ hơn hoặc bằng m được chia thành hai loại:

- **Loại 1**: Số các cách phân tích số v thành tổng các số nguyên dương không chứa số m. Điều này có nghĩa số các cách phân tích Loại 1 là số các cách chia số v thành tổng các số nguyên dương nhỏ hơn hoặc bằng m-1. Như vậy số Loại 1 chính là F[m-1,v].

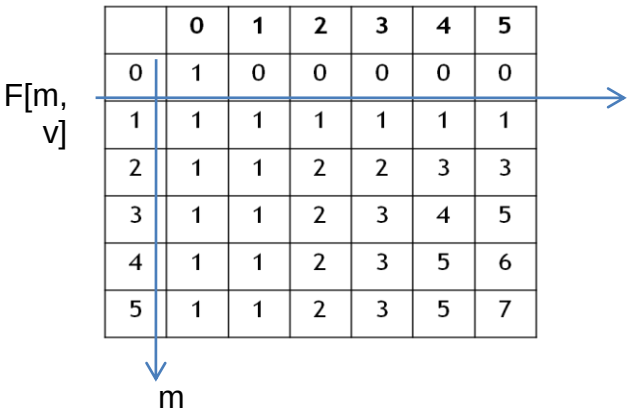
- **Loại 2**: Số các cách phân tích số v thành tổng các số nguyên dương chứa ít nhất một số m. Nếu ta xem sự xuất hiện của một hay nhiều số m trong phép phân tích v chỉ là 1, thì theo nguyên lý nhân số lượng các cách phân tích loại này chính là số cách phân tích số v thành tổng các số nhỏ hơn m-v hay F[m, v-m].

Trong trường hợp m> v thì F[m,v-m] = 0 nên ta chỉ có các cách phân tích loại 1. Trong trường hợp m≤v thì ta có cả hai cách phân tích loại 1 và loại 2. Vì vậy:

- F[m, v] = F[m-1, v] nếu m>v.
- F[m, v] = F[m-1, v] + F[m, v-m] nếu m ≤v.

Tổng quát.

- **Bước cơ sở**: F[0,0] = 1; F[0,v] = 0 với v>0.
- **Tính toán giá trị bảng phương án dựa vào công thức truy hồi**:
  - F[m, v] = F[m-1, v] nếu v > m;
  - F[m, v] = F[m-1, v] + F[m, v-m] nếu m≥v.
- **Lưu vết**. tìm F[5, 5] = F[4, 5] + F[5, 0] = 6 + 1 = 7.



	0	1	2	3	4	5
0	1	0	0	0	0	0
1	1	1	1	1	1	1
2	1	1	2	2	3	3
3	1	1	2	3	4	5
4	1	1	2	3	5	6
5	1	1	2	3	5	7

Chương trình giải quyết bài toán bằng qui hoạch động như dưới đây.

```
#include      <iostream>
using namespace std;
int main (void ) {
        int  F[100][100], n, m, v;
        cout<<"Nhập n="; cin>>n; //Nhap n=5
        for ( int j=0; j <=n; j++) F[0][j] =0; //thiết lập dòng 0 là 0.
        F[0][0] = 1; //duy có F[0][0] thiết lập là 1
        for (m =1; m<=n; m++) { //xây dựng bảng phương án
                for (v= 0; v<=n; v++){
                        if ( m > v ) F[m][v] = F[m-1][v];
                        else F[m][v] = F[m-1][v] + F [m][v-m];
                }
        }
```

```
    cout<<"Kết quả:"<<F[n][n]<<endl;
}
```

Ví dụ 2.16. Giải bài toán cái túi bằng qui hoạch động.

Lời giải. Gọi A =(a_1, a_2, …, a_n), C =(c_1, c_2, …, c_n) là vector trọng lượng và giá trị sử dụng các đồ vật. Gọi F[i, w] là giá trị lớn nhất bằng cách chọn các đồ vật {1, 2, ..,i} với giới hạn trọng lượng w. Khi đó, giá trị lớn nhất khi chọn n đồ vật với trọng lượng b là F[n,m], trong đó m là giới hạn trọng lượng túi.

Bước cơ sở: F[0, w] = 0 vì giá trị lớn nhất khi chọn 0 gói với giới hạn trọng lượng w là 0.

Bước truy hồi: với mọi i>0 và trọng lượng w, khi đó việc chọn tối các gói từ {1, 2, .., i} sẽ có hai khả năng: chọn được đồ vật i và không chọn được đồ vật i. Giá trị lớn nhất của một trong hai cách chọn này chính là F[i, w].

- Nếu không chọn được đồ vật i thì giá trị lớn nhất ta chỉ có thể chọn từ i-1 đồ vật với trọng lượng w. Như vậy, F[i, w] = F[i-1, w].
- Nếu chọn được đồ vật i thì F[i, w] chính là giá trị sử dụng đồ vật i là c_i cộng thêm với giá trị lớn nhất được chọn từ các đồ vật {1, 2, .., i-1} với giới hạn trọng lượng là w-a_i. Nói cách khác F[i, w] = c_i + F[i-1, w-a_i].

Xây dựng bảng phương án: Bảng phương án gồm n+1 dòng và m+1 cột (m là giới hạn trọng lượng túi). Dòng đầu tiên ghi toàn bộ số 0 đó là lời giải bài toán cơ sở. Sử dụng công thức truy hồi tính dòng 1 dựa vào dòng 0, tính dòng 2 dựa vào các hàng còn lại cho đến khi nhận được đầy đủ n +1 dòng.

Truy vết: F[n, m] trong bảng phương án chính là giá trị lớn nhất chọn n đồ vật với giới hạn trọng lượng m. Nếu F[n,m] = F[n-1, m] thì phương án tối ưu không chọn đồ vật n và ta truy tiếp F[n-1, m]. Nếu F[n,m] ≠ F[n-1, m] thì phương án tối ưu có đồ vật n và ta truy tiếp F[n-1, m-a_n]. Truy tiếp đến hàng thứ 0 của bảng phương án ta nhận được phương án tối ưu.

Chương trình giải bài toán cái túi bằng qui hoạch động được thể hiện như dưới đây.

```
#include      <iostream>
#include      <fstream>
#define   MAX     100
using namespace std;
int X[MAX], A[MAX], C[MAX],F[MAX][MAX],n, B;
void Init(void) {//khởi tạo dữ liệu
        ifstream fp("caitui.in");fp>>n>>B; //đọc số lượng đồ vật và trọng lượng túi
        for(int i=1; i<=n; i++) //đọc vector trọng lượng và giá trị sử dụng
                fp>>A[i]>>C[i];
        for(int i=1; i<=n; i++) //khởi đầu bảng phương án
```

```
            for(int i=0; i<=B; i++)
                F[0][i]=0;
        fp.close();
}
void Optimization(void) {//thuật toán qui hoạch động
        int i, j;
        for (i =1; i<=n; i++){
            for(j=0; j<=B; j++){
                F[i][j] = F[i-1][j];
                if (j>=A[i] && F[i][j]<F[i-1][j-A[i]]+ C[i])
                    F[i][j]= F[i-1][j-A[i]]+C[i];
            }
        }
}
void Trace(void){ //lần vết
        cout<<"\n Giá trị tối ưu:"<<F[n][B];
        cout<<"\n Phương án tối ưu:";
        while(n!=0) {
            if(F[n][B]!=F[n-1][B]) {
                cout<<n<<"   ";
                B = B-A[n];
            }
            n--;
        }
}
int main (void ) {
   Init();Optimization();Trace();
}
```

BÀI TẬP

BÀI 1. XÂU NHỊ PHÂN KẾ TIẾP

Cho xâu nhị phân X[], nhiệm vụ của bạn là hãy đưa ra xâu nhị phân tiếp theo của X[]. Ví dụ X[] =”010101” thì xâu nhị phân tiếp theo của X[] là “010110”.

Input:

- Dòng đầu tiên đưa vào số lượng test T.
- Những dòng kế tiếp đưa vào các bộ test. Mỗi bộ test là một xâu nhi phân X.
- T, X[] thỏa mãn ràng buộc: $1 \le T \le 100$; $1 \le length(X) \le 10^3$.

Output:

- Đưa ra kết quả mỗi test theo từng dòng.

Input	Output
2 010101 111111	010110 000000

BÀI 2. TẬP CON KẾ TIẾP

Cho hai số N, K và một tập con K phần tử X[] =(X_1, X_2,.., X_K) của 1, 2, .., N. Nhiệm vụ của bạn là hãy đưa ra tập con K phần tử tiếp theo của X[]. Ví dụ N=5, K=3, X[] ={2, 3, 4} thì tập con tiếp theo của X[] là {2, 3, 5}.

Input:

- Dòng đầu tiên đưa vào số lượng test T.
- Những dòng kế tiếp đưa vào các bộ test. Mỗi bộ test gồm hai dòng: dòng thứ nhất là hai số N và K; dòng tiếp theo đưa vào K phần tử của X[] là một tập con K phần tử của 1, 2, .., N.
- T, K, N, X[] thỏa mãn ràng buộc: $1 \le T \le 100$; $1 \le K \le N \le 10^3$.

Output:

- Đưa ra kết quả mỗi test theo từng dòng.

Input	Output
2 5 3 1 4 5 5 3 3 4 5	2 3 4 1 2 3

BÀI 3. HOÁN VỊ KẾ TIẾP

Cho số tự nhiên N và một hoán vị X[] của 1, 2, .., N. Nhiệm vụ của bạn là đưa ra hoán vị tiếp theo của X[]. Ví dụ N=5, X[] = {1, 2, 3, 4, 5} thì hoán vị tiếp theo của X[] là {1, 2, 3, 5, 4}.

Input:

- Dòng đầu tiên đưa vào số lượng test T.
- Những dòng kế tiếp đưa vào các bộ test. Mỗi bộ test gồm hai dòng: dòng thứ nhất là số N; dòng tiếp theo đưa vào hoán vị X[] của 1, 2, .., N.
- T, N, X[] thỏa mãn ràng buộc: $1 \le T \le 100$; $1 \le N \le 10^3$.

Output:

- Đưa ra kết quả mỗi test theo từng dòng.

Input	Output
2 5 1 2 3 4 5 5 5 4 3 2 1	1 2 3 5 4 1 2 3 4 5

BÀI 4. SẮP XẾP QUÂN HẬU

Cho một bàn cờ vua có kích thước n * n, ta biết ràng quân hậu có thể di chuyển theo chiều ngang, dọc, chéo. Vấn đề đặt ra rằng, có n quân hậu, bạn cần đếm số cách đặt n quân hậu này lên bàn cờ sao cho với 2 quân hậu bất kì, chúng không "ăn" nhau.

Input: Dòng đầu ghi số bộ test T (T<5). Mỗi bộ test ghi một số nguyên dương n duy nhất (không quá 10)

Output: Ghi kết quả mỗi bộ test trên một dòng. Số cách đặt quân hậu.

Ví dụ:

Input	Output
1 4	2

BÀI 5. SẮP XẾP CÔNG VIỆC

Cho hệ gồm N hành động. Mỗi hành động được biểu diễn như một bộ đôi $<S_i, F_i>$ tương ứng với thời gian bắt đầu và thời gian kết thúc của mỗi hành động. Hãy tìm phương án thực hiện nhiều nhất các hành động được thực hiện bởi một máy hoặc một người sao cho hệ không xảy ra mâu thuẫn.

Input:

- Dòng đầu tiên đưa vào số lượng bộ test T.
- Những dòng kế tiếp đưa vào các bộ test. Mỗi bộ test gồm 3 dòng: dòng thứ nhất đưa vào số lượng hành động N; dòng tiếp theo đưa vào N số S_i tương ứng với thời gian bắt đầu mỗi hành động; dòng cuối cùng đưa vào N số F_i tương ứng với thời gian kết thúc mỗi hành động; các số được viết cách nhau một vài khoảng trống.
- T, N, S_i, F_i thỏa mãn ràng buộc: $1 \le T \le 100$; $1 \le N, F_i, S_i \le 1000$.

Output:

- Đưa số lượng lớn nhất các hành động có thể được thực thi bởi một máy hoặc một người.

Ví dụ:

Input	Output
1 6 1 3 0 5 8 5 2 4 6 7 9 9	4

BÀI 6. NỐI DÂY

Cho N sợi dây với độ dài khác nhau được lưu trong mảng A[]. Nhiệm vụ của bạn là nối N sợi dây thành một sợi sao cho tổng chi phí nối dây là nhỏ nhất. Biết chi phí nối sợi dây thứ i và sợi dây thứ j là tổng độ dài hai sợi dây A[i] và A[j].

Input:

- Dòng đầu tiên đưa vào số lượng bộ test T.
- Những dòng kế tiếp đưa vào các bộ test. Mỗi bộ test gồm hai dòng: dòng thứ nhất đưa vào số lượng sợi dây N; dòng tiếp theo đưa vào N số A[i] là độ dài của các sợi dây; các số được viết cách nhau một vài khoảng trống.
- T, N, A[i] thỏa mãn ràng buộc: $1 \le T \le 100$; $1 \le N \le 10^6$; $0 \le A[i] \le 10^6$.

Output:

- Đưa ra kết quả mỗi test theo từng dòng.

Ví dụ:

Input	Output
2 4 4 3 2 6 5 4 2 7 6 9	29 62

BÀI 7. XÂU CON CHUNG DÀI NHẤT

Cho 2 xâu S1 và S2. Hãy tìm xâu con chung dài nhất của 2 xâu này *(các phần tử không nhất thiết phải liên tiếp nhau).*

Input: Dòng đầu tiên là số lượng bộ test T ($T \leq 20$). Mỗi test gồm hai dòng, mô tả xâu S1 và S2, mỗi xâu có độ dài không quá 1000 và chỉ gồm các chữ cái in hoa.

Output: Với mỗi test, in ra độ dài dãy con chung dài nhất trên một dòng.

Ví dụ:

Input	**Output**
2 AGGTAB GXTXAYB AA BB	4 0

Giải thích test 1: Dãy con chung là G, T, A, B.

BÀI 8. DÃY CON TĂNG DÀI NHẤT

Cho một dãy số nguyên gồm N phần tử A[1], A[2], ... A[N].

Biết rằng dãy con tăng là 1 dãy $A[i_1],... A[i_k]$

thỏa mãn $i_1 < i_2 < ... < i_k$ và $A[i_1] < A[i_2] < .. < A[i_k]$.

Hãy cho biết dãy con tăng dài nhất của dãy này có bao nhiêu phần tử?

Input: Dòng 1 gồm 1 số nguyên là số N ($1 \leq N \leq 1000$). Dòng thứ 2 ghi N số nguyên A[1], A[2], .. A[N] ($1 \leq A[i] \leq 1000$).

Output: Ghi ra độ dài của dãy con tăng dài nhất.

Ví dụ:

Input	**Output**
6 1 2 5 4 6 2	4

BÀI 9. DÃY CON CÓ TỔNG BẰNG S

Cho N số nguyên dương tạo thành dãy $A=\{A_1, A_2, ..., A_N\}$. Tìm ra một dãy con của dãy A (không nhất thiết là các phần tử liên tiếp trong dãy) có tổng bằng S cho trước.

Input: Dòng đầu ghi số bộ test T (T<10). Mỗi bộ test có hai dòng, dòng đầu tiên ghi hai số nguyên dương N và S ($0 < N \leq 200$) và S ($0 < S \leq 40000$). Dòng tiếp theo lần lượt ghi N số hạng của dãy A là các số $A_1, A_2, ..., A_N$ ($0 < A_i \leq 200$).

Output: Với mỗi bộ test, nếu bài toán vô nghiệm thì in ra "NO", ngược lại in ra "YES"

Ví dụ:

Input	Output
2 5 6 1 2 4 3 5 10 15 2 2 2 2 2 2 2 2 2 2	YES NO

BÀI 10. TỔ HỢP C(n, k)

Cho 2 số nguyên n, k. Bạn hãy tính C(n, k) modulo 10^9+7.

Input:

- Dòng đầu tiên là số lượng bộ test T ($T \leq 20$).
- Mỗi test gồm 2 số nguyên n, k ($1 \leq k \leq n \leq 1000$).

Output:

- Với mỗi test, in ra đáp án trên một dòng.

Ví dụ:

Input	Output
2 5 2 10 3	10 120

CHƯƠNG 3. SẮP XẾP VÀ TÌM KIẾM

Một trong những vấn đề quan trọng bậc nhất của khoa học máy tính là tìm kiếm thông tin. Có thể nói, hầu hết các hoạt động của người dùng hoặc các ứng dụng tin học đều liên quan đến tìm kiếm. Muốn tìm kiếm thông tinh nhanh, hiệu quả, chính xác ta cần có phương pháp tổ chức và sắp xếp dữ liệu tốt. Chính vì vậy, sắp xếp được xem như giai đoạn đầu chuẩn bị cho quá trình tìm kiếm. Nội dung chương này trình bày các thuật toán sắp xếp và tìm kiếm, bao gồm: các thuật toán sắp xếp đơn giản, các thuật toán sắp xếp nhanh, các thuật toán tìm kiếm tuyến tính, tìm kiếm nhị phân, tìm kiếm nội suy & tìm kiếm Jumping.

3.1. Giới thiệu vấn đề

Bài toán tìm kiếm có thể được phát biểu như sau: Cho dãy gồm n đối tượng r_1, r_2, .., r_n. Mỗi đối tượng r_i được tương ứng với một khóa k_i ($1 \le i \le n$). Nhiệm vụ của tìm kiếm là xây dựng thuật toán tìm đối tượng có giá trị khóa là *X* cho trước. *X* còn được gọi là khóa tìm kiếm hay tham biến tìm kiếm (arrgument). Bài toán tìm kiếm bao giờ cũng hoàn thành bởi một trong hai tình huống:

- Nếu tìm thấy đối tượng có khóa *X* trong tập các đối tượng thì ta nói phép tìm kiếm thành công (successful).
- Nếu không tìm thấy đối tượng có khóa *X* trong tập các đối tượng thì ta nói phép tìm kiếm không thành công (unsuccessful).

Sắp xếp là phương pháp bố trí lại các đối tượng theo một trật tự nào đó. Ví dụ bố trí theo thứ tự tăng dần hoặc giảm dần đối với dãy số, bố trị theo thứ tự từ điển đối với các xâu ký tự. Mục tiêu của sắp xếp là để lưu trữ và tìm kiếm đối tượng (thông tin) để đạt hiệu quả cao trong tìm kiếm. Có thể nói, sắp xếp là sân sau quả quá trình tìm kiếm. Muốn tìm kiếm và cung cấp thông tin nhanh thì ta cần phải sắp xếp thông tin sao cho hợp lý. Bài toán sắp xếp có thể được phát biểu như sau:

Bài toán sắp xếp: Cho dãy gồm n đối tượng r_1, r_2, .., r_n. Mỗi đối tượng r_i được tương ứng với một khóa k_i ($1 \le i \le n$). Nhiệm vụ của sắp xếp là xây dựng thuật toán bố trí các đối tượng theo một trật tự nào đó của các giá trị khóa. Trật tự của các giá trị khóa có thể là tăng dần hoặc giảm dần tùy thuộc vào mỗi thuật toán tìm kiếm cụ thể.

Trong các mục tiếp theo, chúng ta xem tập các đối tượng cần sắp xếp là tập các số. Việc mở rộng các số cho các bản ghi tổng quát cũng được thực hiện tương tự bằng cách thay đổi các kiểu dữ liệu tương ứng. Cũng giống như tìm kiếm, việc làm này không làm mất đi bản chất của thuật toán.

3.2. Các thuật toán sắp xếp đơn giản

Các thuật toán sắp xếp đơn giản được trình bày ở đây bao gồm:

- Thuật toán sắp xếp kiểu lựa chọn (Selection Sort).
- Thuật toán sắp xếp kiểu chèn trực tiếp (Insertion Sort).

- Thuật toán sắp xếp kiểu sủi bọt (Bubble Sort).

3.2.1. Thuật toán Selection-Sort

Thuật toán sắp xếp đơn giản nhất được đề cập đến là thuật toán sắp xếp kiểu 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. Thuật toán được trình bày chi tiết trong Hình 3.1.

a) Biểu diễn thuật toán

Thuật toán Selection-Sort:

Input:

- Dãy các đối tượng (các số) : Arr[0], Arr[1],..,Arr[n-1].
- Số lượng các đối tượng cần sắp xếp: n.

Output:

- Dãy các đối tượng đã được sắp xếp (các số) : Arr[0], Arr[1],..,Arr[n-1].

Formats: Selection-Sort(Arr[] , n);

Actions:

```
for (i =0; i<n-1; i++) { //duyệt các phần tử i=0,1,.., n-1
        min_idx = i;  //gọi min_idx là vị trí của phần tử nhỏ nhất trong dãy con
        for ( j = i +1; j<n; j++ ) { //duyệt từ phần tử tiếp theo j=i+1,..,n.
                if (Arr[i] > Arr[j] ) // nếu Arr[i] không phải nhỏ nhất trong dãy con
                        min_idx = j; //ghi nhận đây mới là vị trí phần tử nhỏ nhất.
        }
        //đặt phần tử nhỏ nhất vào vị trí đầu tiên của dãy con chưa được sắp
        Temp = Arr[i] ; Arr[i] = Arr[min_idx]; Arr[min_idx] = temp;
}
End.
```

Hình 3.1. Thuật toán Selection Sort.

b) Độ phức tạp thuật toán

Độ phức tạp thuật toán Selection Sort là $O(N^2)$, trong đó N là số lượng phần tử cần sắp xếp. Bạn đọc tự tìm hiểu phương pháp xác định độ phức tạp thuật toán Selection Sort trong các tài liệu tham khảo liên quan.

c) Kiểm nghiệm thuật toán

Kiểm nghiệm thuật toán: Arr[] = { 9, 7, 12, 8, 6, 5 }, n = 6.

Bước	min-idx	Dãy số Arr[] =?
i = 0	min-idx=5	Arr[] = {**5**, 7, 12, 8, 6, 9}
i = 1	min-idx=4	Arr[] = {**5**, **6**, 12, 8, 7, 9}
i = 2	min-idx=4	Arr[] = {**5**, **6**, **7**, 8, 12, 9}
i = 3	min-idx=3	Arr[] = {**5**, **6**, **7**, **8**, 12, 9}
i = 4	min-idx=5	Arr[] = {**5**, **6**, **7**, **8**, **9**, 12}

d) Cài đặt thuật toán

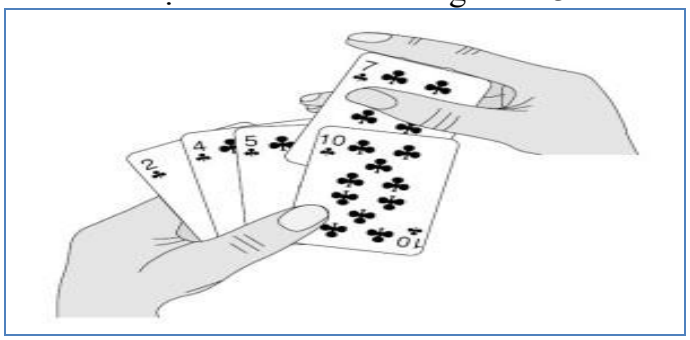
```
#include <iostream>
#include <iomanip>
using namespace std;
void swap(int  *x, int  *y){ //đổi giá trị của x và y
        int temp = *x;  *x = *y;      *y = temp;
}
void SelectionSort(int arr[], int n){ //thuật toán selection sort
        int i, j, min_idx; //min_idx là vị trí để arr[min_index] nhỏ nhất
        for (i = 0; i < n-1; i++)    { //duyệt n-1 phần tử
                min_idx = i; //vị trí số bé nhất tạm thời là i
                for (j = i+1; j < n; j++){ //tìm vị trí số bé nhất arr[i+1],.., arr[n-1]
                        if (arr[j] < arr[min_idx])
                                min_idx = j;
                }
                swap(&arr[min_idx], &arr[i]);//tráo đổi arr[min_idx] và arr[i]
        }
}
void printArray(int arr[], int size){ //hiển thị kết quả
        for (int i=0; i < size; i++)
                cout<<arr[i]<<setw(5);
        cout<<endl;
}
int main(){ //chương trình chính
        int arr[] = {64, 25, 12, 22, 11};
```

```
        int n = sizeof(arr)/sizeof(arr[0]);
        SelectionSort(arr, n);
        cout<<"Dãy số được sắp: \n";
        printArray(arr, n);
}
```

3.2.2. Thuật toán Insertion Sort

Thuật toán sắp xếp kiểu chèn được thực hiện đơn giản theo cách của người chơi bài thông thường. Phương pháp được thực hiện như sau:

- Lấy phần tử đầu tiên Arr[0] (quân bài đầu tiên) như vậy ta có dãy một phần tử được sắp.
- Lấy phần tiếp theo (quân bài tiếp theo) Arr[1] và tìm vị trí thích hợp chèn Arr[1] vào dãy Arr[0] để có dãy hai phần tử đã được sắp.
- Tổng quát, tại bước thứ i ta lấy phần tử thứ i và chèn vào dãy Arr[0],..,Arr[i-1] đã được sắp trước đó để nhận được dãy i phần tử được sắp. Quá trình sắp xếp sẽ kết thúc khi quân bài cuối cùng (i = n) được chèn đúng vị trí. Thuật toán Inserttion Sort được mô tả chi tiết trong Hình 3.2.



a) Biểu diễn thuật toán

Thuật toán Insertion-Sort:

Input:

- Dãy các đối tượng (các số) : Arr[0], Arr[1],..,Arr[n-1].
- Số lượng các đối tượng cần sắp xếp: n.

Output:

- Dãy các đối tượng đã được sắp xếp (các số) : Arr[0], Arr[1],..,Arr[n-1].

Formats: Insertion-Sort(Arr, n):

Actions:

```
for (i = 1; i < n; i++) { //lặp i=1, 2,..,n.
        key = Arr[i]; //key là phần tử cần chèn vào dãy Arr[0],.., Arr[i-1]
        j = i-1;
        while (j >= 0 && Arr[j] > key) { //Duyệt lùi từ vị trí j=i-1
                Arr[j+1] = Arr[j]; //dịch chuyển Arr[j] lên vị trí Arr[j+1]
                j = j-1;
        }
        Arr[j+1] = key; // vị trí thích hợp của key trong dãy là Arr[j+1]
}
```

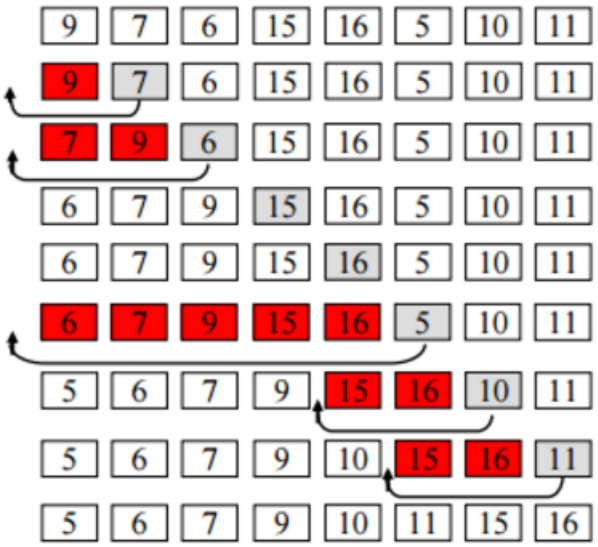
End.

Hình 3.2. Thuật toán Insertion Sort

b) Độ phức tạp thuật toán

Độ phức tạp thuật toán là $O(N^2)$, với N là số lượng phần tử. Thuật toán có thể cải tiến bằng cách sử dụng hàng đợi ưu tiên với độ phức tạp O(N.Log(N)).

c) Kiểm nghiệm thuật toán



d) Cài đặt thuật toán

```
#include <iostream>
#include <iomanip>
```

```
using namespace std;
void insertionSort(int arr[], int n){
	int i, key, j;
	for (i = 1; i < n; i++){
		key = arr[i];
		j = i-1;
		while (j >= 0 && arr[j] > key){
			arr[j+1] = arr[j];
			j = j-1;
		}
		arr[j+1] = key;
	}
}
void printArray(int arr[], int n){
	int i;cout<<"\n Day so duoc sap:";
	for (i=0; i < n; i++)
		cout<<arr[i]<<setw(3);
}
int main(){
	int arr[] = {12, 11, 13, 5, 6};
	int n = sizeof(arr)/sizeof(arr[0]);
	insertionSort(arr, n);
	printArray(arr, n);
}
```

3.2.3. Thuật toán Bubble Sort

Thuật toán sắp xếp kiểu sủi bọt được thực hiện đơn giản bằng cách tráo đổi hai phần từ liền kề nhau nếu chúng chưa được sắp xếp. Thuật toán được mô tả chi tiết trong Hình 3.3.

a) Biểu diễn thuật toán

Thuật toán Bubble-Sort:
Input:
- Dãy các đối tượng (các số) : Arr[0], Arr[1],..,Arr[n-1].
- Số lượng các đối tượng cần sắp xếp: n.

Output:
- Dãy các đối tượng đã được sắp xếp (các số) : Arr[0], Arr[1],..,Arr[n-1].

Formats: Insertion-Sort(Arr, n);
Actions:

```
for (i = 0; i < n; i++) { //lặp i=0, 1, 2,..,n.
        for (j=0; j<n-i-1; j++ ) {//lặp j =0, 1,.., n-i-1
                if (Arr[j] > Arr[j+1] ) { //nếu Arr[j]>Arr[j+1] thì đổi chỗ
                        temp = Arr[j];
                        Arr[j] = Arr[j+1];
                        Arr[j+1] = temp;
                }
        }
}
```

End.

Hình 3.3. Thuật toán Bubble Sort

b) Độ phức tạp thuật toán

Độ phức tạp thuật toán là $O(N^2)$, với N là số lượng phần tử. Bạn đọc tự tìm hiểu phương pháp ước lượng và chứng minh độ phức tạp thuật toán Bubble Sort trong các tài liệu liên quan.

c) Kiểm nghiệm thuật toán

Kiểm nghiệm thuật toán: Arr[] = { 9, 7, 12, 8, 6, 5 }, n = 6.

Bước	Dãy số Arr[] =?
i = 0	Arr[] = {7, 9, 8, 6, 5, **12**}
i = 1	Arr[] = {7, 8, 6, 5, **9, 12**}
i = 2	Arr[] = {7, 6, 5, **8, 9, 12**}
i = 3	Arr[] = {6, 5, **7, 8, 9, 12**}
i = 4	Arr[] = {**5, 6, 7, 8, 9, 12**}
i = 5	Arr[] = {**5, 6, 7, 8, 9, 12**}

d) Cài đặt thuật toán

```
#include <iostream>
#include <iomanip>
using namespace std;
void swap(int *x, int *y){ //đổi chỗ hai số x và y
        int temp = *x; *x = *y;*y = temp;
```

```
}
void bubbleSort(int arr[], int n){ //thuật toán bubble sort
        int i, j;
        for (i = 0; i < n; i++) {
                for (j = 0; j < n-i-1; j++)   {
                        if (arr[j] > arr[j+1])
                                swap(&arr[j], &arr[j+1]);
                }
        }
}
void printArray(int arr[], int size){
        cout<<"\n Dãy được sắp:";
        for (int i=0; i < size; i++)
                cout<<arr[i]<<setw(3);
}
int main(){
        int arr[] = {64, 34, 25, 12, 22, 11, 90};
        int n = sizeof(arr)/sizeof(arr[0]);
        bubbleSort(arr, n);    printArray(arr, n);
}
```

3.3. Thuật toán Quick Sort

Thuật toán sắp xếp Quick-Sort được thực hiện theo mô hình chia để trị (Devide and Conquer). Thuật toán được thực hiện xung quanh một phần tử gọi là chốt (key). Mỗi cách lựa chọn vị trí phần tử chốt trong dãy sẽ cho ta một phiên bản khác nhau của thuật toán. Các phiên bản (version) của thuật toán Quick-Sort bao gồm:

- Luôn lựa chọn phần tử đầu tiên trong dãy làm chốt.
- Luôn lựa chọn phần tử cuối cùng trong dãy làm chốt.
- Luôn lựa chọn phần tử ở giữa dãy làm chốt.
- Lựa chọn phần tử ngẫu nhiên trong dãy làm chốt.

Mấu chốt của thuật toán Quick-Sort là làm thế nào ta xây dựng được một thủ tục phân đoạn (Partition). Thủ tục Partition có hai nhiệm vụ chính:

- Định vị chính xác vị trí của chốt trong dãy nếu được sắp xếp;
- Chia dãy ban đầu thành hai dãy con: dãy con ở phía trước phần tử chốt bao gồm các phần tử nhỏ hơn hoặc bằng chốt, dãy ở phía sau chốt có giá trị lớn hơn chốt.

Thuật toán Partion được mô tả chi tiết trong Hình 3.4 với khóa chốt là phần tử cuối cùng của dãy. Phiên bản Quick Sort tương ứng được mô tả chi tiết trong Hình 3.5.

a) Biểu diễn thuật toán

Thuật toán Partition:
Input :
- Dãy Arr[] bắt đầu tại vị trí l và kết thúc tại h.
- Cận dưới của dãy con: l
- Cận trên của dãy con: h

Output:
- Vị trí chính xác của Arr[h] nếu dãy Arr[] được sắp xếp.

Formats: Partition(Arr, l, h);
Actions:

```
x = Arr[h];    // chọn x là chốt của dãy Arr[l],Arr[l+1],.., Arr[h]
i = (l - 1);  // i là chỉ số của các phần tử nhỏ hơn chốt x.
for ( j = l;  j <= h- 1;  j++) { //duyệt các chỉ số j=l, l+1, …, h-1.
if (Arr[j] <= x){ //nếu Arr[j] nhỏ hơn hoặc bằng chốt x.
                    i++; //tăng vị trí các phần tử nhỏ hơn chốt
                    swap(&Arr[i], &Arr[j]);  // đổi chỗ Arr[i] cho Arr[j].
}
}
swap(&Arr[i + 1], &Arr[h]); // đổi chỗ Arr[i+1] cho Arr[h].
return (i + 1); //vị trí i+1 chính là vị trí chốt nếu dãy được sắp xếp
```

End.

Hình 3.4. Thuật toán Partition với chốt là vị trí cuối cùng của dãy

Thuật toán Quick-Sort:
Input :
- Dãy Arr[] gồm n phần tử.
- Cận dưới của dãy: l.
- Cận trên của dãy : h

Output:
- Dãy Arr[] được sắp xếp.

Formats: Quick-Sort(Arr, l, h);
Actions:

```
if( l<h) { // Nếu cận dưới còn nhỏ hơn cận trên
        p = Partition(Arr, l, h); //thực hiện Partition() chốt h.
        Quick-Sort(Arr, l, p-1); //thực hiện Quick-Sort nửa bên trái.
        Quick-Sort(Arr, p+1, h);//thực hiện Quick-Sort nửa bên phải
}
```

End.

Hình 3.5. Thuật toán Quick-Sort với chốt là vị trí cuối cùng của dãy

b) Độ phức tạp thuật toán

Độ phức tạp thuật toán trong trường hợp xấu nhất là $O(N^2)$, trong trường hợp tốt nhất là O(N.Log(N)), với N là số lượng phần tử. Bạn đọc tự tìm hiểu và chứng minh độ phức tạp thuật toán Quick Sort trong các tài liệu liên quan.

c) Kiểm nghiệm thuật toán

Kiểm nghiệm thuật toán Quick-Sort: Quick-Sort(Arr, 0, 9);
Arr[] = {10, 27, 15, 29, 21, 11, 14, 18, 12, 17};
Cận dưới l=0, cận trên h = 9.

p = Partition(Arr,l,h)	Giá trị Arr[]=?
p=5:l=0, h=9	{10,15,11,14,12}, **(17)**,{29,18, 21, 27}
P=2:l=0, h=4	{10,11},**(12)**, {14,15}, (17),{29,18, 21, 27}
P=1:l=0, h=1	{10,**11**},(12), {14,15}, (17),{29,18, 21, 27}
P=4: l=3, h=4	{10,11},(12), {14,**15**}, (17),{29,18, 21, 27}
P=8: l=6, h=9	{10,11},(12), {14,15}, (17),{18,21},(**27**),{29}
P=7:l=6, h=7	{10,11},(12), {14,15}, (17),{18,**21**},(27),{29}
Kết luận dãy được sắp Arr[] ={ 10, 11, 12, 14, 15, 17, 18, 21, 27, 29}	

d) Cài đặt thuật toán

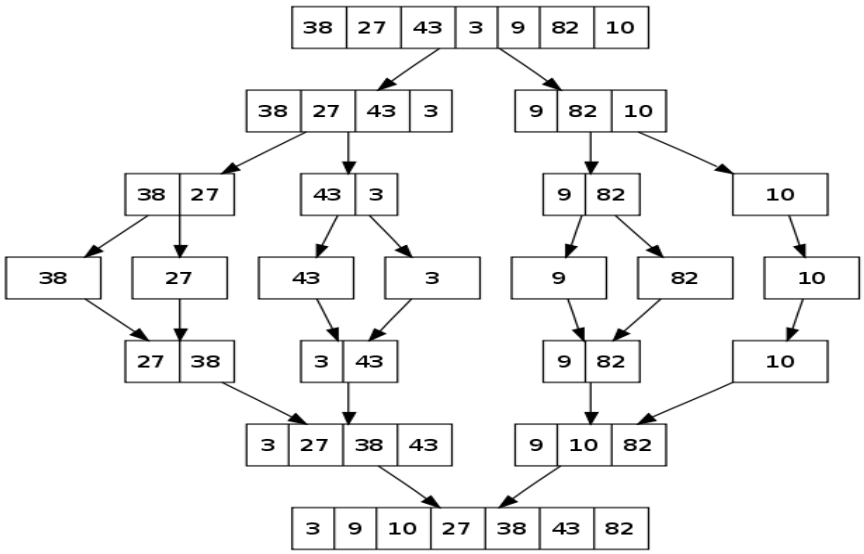
```
#include<iostream>
#include<iomanip>
using namespace std;
void swap(int* a, int* b){ //đổi chỗ a và b
        int t = *a; *a = *b; *b = t;
}
int partition (int arr[], int l, int h){ //thuật toán partition chốt h
        int x = arr[h];    // x chính là chốt
        int i = (l - 1);  // i lấy vị trí nhỏ hơn l
        for (int j = l; j <= h- 1; j++) {//duyệt từ l đến h-1
                // If current element is smaller than or equal to pivot
                if (arr[j] <= x){ //nếu arr[j] bé hơn hoặc bằng chốt
                        i++;    // tăng i lên một đoen vị
                        swap(&arr[i], &arr[j]);  // đổi chỗ arr[i] cho arr[j]
                }
        }
        swap(&arr[i + 1], &arr[h]);  //đổi chỗ cho arr[i+1] và arr[h]
        return (i + 1); //đây là vị trí của chốt
}
void quickSort(int arr[], int l, int h){
        if (l < h){
                int p = partition(arr, l, h); // tìm vị trí của chốt
                quickSort(arr, l, p - 1);//trị nửa bên trái
                quickSort(arr, p + 1, h);//trị nửa bên phải
        }
}
void printArray(int arr[], int size){ //thủ tục in kết quả
```

```
        int i; cout<<"\n Dãy được sắp:";
        for (i=0; i < size; i++)
                cout<<arr[i]<<setw(3);
}
int main(){ //chương trình chính
        int arr[] = {10, 27, 15, 29, 21, 11, 14, 18, 12, 17};
        int n = sizeof(arr)/sizeof(arr[0]);
        quickSort(arr, 0, n-1);
        printArray(arr, n);
}
```

3.4. Thuật toán Merge Sort

Giống như Quick-Sort, Merge-Sort cũng được xây dựng theo mô hình chia để trị (Devide and Conquer). Thuật toán chia dãy cần sắp xếp thành hai nửa. Sau đó gọi đệ qui lại cho mỗi nửa và hợp nhất lại các đoạn đã được sắp xếp. Thuật toán được tiến hành theo 4 bước dưới đây:

- Tìm điểm giữa của dãy và chi dãy thành hai nửa.
- Thực hiện Merge-Sort cho nửa thứ nhất.
- Thực hiện Merge-Sort cho nửa thứ hai.
- Hợp nhất hai đoạn đã được sắp xếp.



Mấu chốt của thuật toán Merge-Sort là làm thế nào ta xây dựng được một thủ tục hợp nhất (Merge). Thủ tục Merge thực hiện hòa nhập hai dãy đã được sắp xếp để tạo thành một dãy cũng được sắp xếp. Bài toán có thể được phát biểu như sau:

Bài toán hợp nhất Merge: Cho hai nửa của một dãy Arr[1,..,m] và A[m+1,..,r] đã được sắp xếp. Nhiệm vụ của ta là hợp nhất hai nửa của dãy Arr[1,..,m] và Arr[m+1,..,r] để trở thành một dãy Arr[1, 2,..,r] cũng được sắp xếp.

Thuật toán Merge được mô tả chi tiết trong Hình 3.6. Thuật toán Merge Sort được mô tả chi tiết trong Hình 3.7.

a) Biểu diễn thuật toán

```
Thuật toán  Merge:
        Input: Arr[], l, m, r. Trong đó, Arr[l]..Arr[m] và Arr[l+1]..Arr[r] đã
        được sắp
        output: Arr[] có  Arr[l]..Arr[r] đã được sắp
Begin:
        int i, j, k, n1 = m - l + 1; n2 =  r - m;
        int L[n1], R[n2]; //tạo lập hai mảng phụ L và R
        for(i = 0; i < n1; i++)  //lưu đoạn được sắp thứ nhất vào L
                L[i] = Arr[l + i];
        for(j = 0; j < n2; j++) //lưu đoạn được sắp thứ nhất vào R
                R[j] = Arr[m + 1+ j];
        // hợp nhất các mảng phụ và trả lại vào Arr[]
        i = 0; j = 0; k = l;
        while (i < n1 && j < n2){
                if (L[i] <= R[j]){  Arr[k] = L[i];  i++; }
                else {  Arr[k] = R[j];  j++;  }
                k++;
        }
         while (i < n1)  {
                Arr[k] = L[i];  i++; k++;
        }
        while (j < n2) {
                arr[k] = R[j];  j++; k++;
        }
End.
```

Hình 3.6. Thuật toán hợp nhất hai đoạn đã được sắp xếp.

Thuật toán Merge-Sort:

Input :

- Dãy số : Arr[];
- Cận dưới: l;
- Cận trên m;

Output:

- Dãy số Arr[] được sắp theo thứ tự tăng dần.

Formats: Merge-Sort(Arr, l, r);

Actions:

```
if ( l< r ) {
        m = (l + r -1) / 2; //phép chia Arr[] thành hai nửa
        Merge-Sort(Arr, l, m);//trị nửa thứ nhất
        Merge-Sort(Arr, m+1, r); //trị nửa thứ hai
        Merge(Arr, l, m, r); //hợp nhất hai nửa đã sắp xếp
}
```

End.

Hình 3.7. Thuật toán Merge Sort.

b) Độ phức tạp thuật toán

Độ phức tạp thuật toán là O(N.Log(N)) với N là số lượng phần tử. Bạn đọc tự tìm hiểu và chứng minh độ phức tạp thuật toán Merge Sort trong các tài liệu liên quan.

c) Kiểm nghiệm thuật toán

Kiểm nghiệm thuật toán Merge-Sort: Merge-Sort(Arr,0,7)

Bước	Kết quả Arr[]=?
1	Arr[] = { 27, 38, 43, 3, 9, 82, 10}
2	Arr[] = { 27, 38, 3, 43, 9, 82, 10}
3	Arr[] = { 3, 27, 38, 43, 9, 82, 10}
4	Arr[] = {3, 27, 38, 43, 9, 82, 10}
5	Arr[] = { 3, 27, 38, 43, 9, 10, 82}
6	Arr[]= { 3, 9, 10, 27, 38, 43, 82}

d) Cài đặt thuật toán

```
#include<iostream>
#include<iomanip>
using namespace std;
void merge(int arr[], int l, int m, int r){//thuật toán hợp nhất hai đoạn đã sắp xếp
        int i, j, k;
```

```
        int n1 = m - l + 1; //số lượng phần tử đoạn 1
        int n2 =  r - m; //số lượng phần tử đoạn 3
        int L[n1], R[n2]; //tạo hai mảng phụ để lưu hai đoạn được sắp
        for(i = 0; i < n1; i++)//lưu đoạn thứ nhất vào L[]
                L[i] = arr[l + i];
        for(j = 0; j < n2; j++)//lưu đoạn thứ hai vào R[]
                R[j] = arr[m + 1+ j];
        i = 0; j = 0; k = l; //bắt đầu hợp nhất
        while (i < n1 && j < n2){ //quá trình hợp nhất
                if (L[i] <= R[j]){
                        arr[k] = L[i];  i++;
                }
                else {
                        arr[k] = R[j];    j++;
                }
                k++;
        }
        while (i < n1) { //lấy các phần tử còn lại trong L[] vào arr[]
                arr[k] = L[i];
                i++; k++;
        }
        while (j < n2){ //lấy các phần tử còn lại trong R[] vào arr[]
                arr[k] = R[j];
                j++; k++;
        }
}
void mergeSort(int arr[], int l, int r){ //thuật toán Merge Sort
        if (l < r){ //nếu cận dưới còn bé hơn cận trên
                int m = l+(r-l)/2; //tìm vị trí ở giữa đoán l, r
                mergeSort(arr, l, m); //trị nửa thứ nhất
                mergeSort(arr, m+1, r); //trị nửa thứ hai
                merge(arr, l, m, r); //hợp nhất hai đoạn đã được sắp
        }
}
void printArray(int Arr[], int size){ //in kết quả
        int i;cout<<"\n Dãy được sắp:";
        for (i=0; i < size; i++)
                cout<<Arr[i]<<setw(3);
}
int main(){
        int arr[] = {38, 27, 43, 3, 9, 82, 10};
        int arr_size = sizeof(arr)/sizeof(arr[0]);
        mergeSort(arr, 0, arr_size - 1);
```

```
        printArray(arr, arr_size);
}
```

3.5. Thuật toán Heap Sort

Thuật toán Heap-Sort được thực hiện dựa trên cấu trúc dữ liệu Heap. Nếu ta muốn sắp xếp theo thứ tự tăng dần ta sử dụng cấu trúc Max Heap, ngược lại ta sử dụng cấu trúc Min-Heap. Vì Heap là một cây nhị phân đầy đủ nên việc biểu diễn Heap một cách hiệu quả có thể thực hiện được bằng mảng. Nếu ta xem xét phần tử thứ i trong mảng thì phần tử 2*i +1, 2*i +2 tương ứng là node con trái và node con phải của i.

Tư tưởng của Heap Sort giống như Selection Sort, chọn phần tử lớn nhất trong dãy đặt vào vị trí cuối cùng, sau đó lặp lại quá trình này cho các phần tử còn lại. Tuy nhiên, điểm khác biệt ở đây là phần tử lớn nhất của Heap luôn là phần tử đầu tiên trên Heap và các phần tử node trái và phải bao giờ cũng nhỏ hơn nội dung node gốc.

Thuật toán được thực hiện thông qua ba bước chính như sau:

- Xây dựng Max Heap từ dữ liệu vào. Ví dụ với dãy A[] = {9, 7, 12, 8, 6, 5} thì Max Heap được xây dựng là A[] = {12, 8, 9, 7, 6, 5}.
- Bắt đầu tại vị trí đầu tiên là phần tử lớn nhất của dãy. Thay thế, phần tử này cho phần tử cuối cùng ta nhận được dãy A[] = {5, 8, 9, 7, 6, 12}.
- Xây dựng lại Max Heap cho n-1 phần tử đầu tiên của dãy và lặp lại quá trình này cho đến khi Heap chỉ còn lại 1 phần tử. Thuật toán được mô tả chi tiết trong Hình 3.8.

a) Biểu diễn thuật toán

```
void heapify(int arr[], int n, int i){ //tạo heap từ mảng Arr[]
        int largest = i;  // thiết lập node gốc
        int l = 2*i + 1;  // node lá trái là là  2*i + 1
        int r = 2*i + 2;  // node lá phải  2*i + 2
        if (l < n && arr[l] > arr[largest])//nếu node con trái lớn hơn gốc
                largest = l;
        if (r < n && arr[r] > arr[largest]) //nếu node con phải lớn hơn gốc
                largest = r;
        if (largest != i){//nếu node lớn nhất không phải node gốc
                swap(arr[i], arr[largest]);
                heapify(arr, n, largest);//đệ qui tiếp tục từ node lớn nhất
        }
}
void heapSort(int arr[], int n){
        for (int i = n / 2 - 1; i >= 0; i--) //xây dựng heap
                heapify(arr, n, i);
        for (int i=n-1; i>=0; i--){ //trích rút từng phần tử trong heap
                swap(arr[0], arr[i]);//luôn đổi chỗ cho phần tử đầu tiên
                heapify(arr, i, 0);//xây dựng max heap từ phần tử thứ i
        }
}
```

Hình 3.8. Thuật toán Heap-Sort

b) Độ phức tạp thuật toán

Độ phức tạp thuật toán là O(N.Log(N)) với N là số lượng phần tử. Bạn đọc tự tìm hiểu và chứng minh độ phức tạp thuật toán Merge Sort trong các tài liệu liên quan.

c) Kiểm nghiệm thuật toán

Kiểm nghiệm thuật toán:

Bước	Dãy số Arr[]
1	Arr [] = {18, 9, 13, 8, 6, 5, 12, 3, 7, 4}
2	Arr[] = { 13, 9, 12, 8, 6, 5, 4, 3, 7, 18}
3	Arr[] = {12, 9, 7, 8, 6, 5, 4, 3, 13, 18}
4	Arr[] = {9, 8, 7, 3, 6, 5, 4, 12, 13, 18}
5	Arr[] = {8, 6, 7, 3, 4, 5, 9, 12, 13, 18}
6	Arr[] = {7, 6, 5, 3, 4, 8, 9, 12, 13, 18}
7	Arr[] = {6, 4, 5, 3, 7, 8, 9, 12, 13, 18}
8	Arr[] = {5, 4, 3, 6, 7, 8, 9, 12, 13, 18}
9	Arr[] = {4, 3, 5, 6, 7, 8, 9, 12, 13, 18}
10	Arr[] = {3, 4, 5, 6, 7, 8, 9, 12, 13, 18}

d) Cài đặt thuật toán

```
#include <iostream>
using namespace std;
void heapify(int arr[], int n, int i){ //tạo heap từ mảng Arr[]
    int largest = i;  // thiết lập node gốc
    int l = 2*i + 1;  // node con trái là  2*i + 1
    int r = 2*i + 2;  // node con trái là  2*i + 2
    if (l < n && arr[l] > arr[largest])//nếu node con trái lớn hơn gốc
            largest = l;
    if (r < n && arr[r] > arr[largest]) //nếu node con phải lớn hơn gốc
            largest = r;
    if (largest != i){//nếu node lớn nhất không phải là gốc
            swap(arr[i], arr[largest]); //đổi chỗ cho node gốc
            heapify(arr, n, largest);//đệ qui từ node largest
    }
}
void heapSort(int arr[], int n){
    for (int i = n / 2 - 1; i >= 0; i--) //xắp đặt lại mảng thành heap
            heapify(arr, n, i);
    for (int i=n-1; i>=0; i--){ //trích rút từng phần tử
```

```
            swap(arr[0], arr[i]);//luôn đổi chỗ cho phần tử đầu tiên
            heapify(arr, i, 0);//xây dựng heap đến phần tử cuối cùng
        }
    }
     void printArray(int arr[], int n) {//in kết quả
        cout<<"\n Dãy được sắp:";
        for (int i=0; i<n; ++i)
            cout << arr[i] <<setw(3);
    }
    int main(){
        int arr[] = {12, 11, 13, 5, 6, 7};
        int n = sizeof(arr)/sizeof(arr[0]);
        heapSort(arr, n);
        printArray(arr, n);
    }
```

3.6. Một số thuật toán tìm kiếm thông dụng

Tìm kiếm là lĩnh vực quan trọng của khoa học máy tính có mặt trong hầu hết các ứng dụng trên máy tính. Các thuật toán tìm kiếm được chia thành ba loại: tìm kiếm trên các đối tượng dữ liệu chưa được sắp xếp (tìm kiếm tuyến tính), tìm kiếm trên các đối tượng dữ liệu đã được sắp xếp (tìm kiếm nhị phân) và tìm kiếm xấp xỉ. Nội dung cụ thể của các phương pháp được thể hiện như dưới đây.

3.6.1. Thuật toán tìm kiếm tuyến tính (Sequential Serch)

Thuật toán tìm kiếm tuyến tính áp dụng cho tất cả các đối tượng dữ liệu chưa được sắp xếp. Để tìm vị trí của *x* trong dãy A[] gồm *n* phần tử, ta chỉ cần duyệt tuần tự trên dãy A[] từ phần tử đầu tiên đến phần tử cuối cùng. Nếu *x* = A[*i*] thì *i* chính là vị trí của *x* thuộc dãy A[]. Nếu duyệt đến phần tử cuối cùng vẫn chưa tìm thấy *x* ta kết luận *x* không có mặt trong dãy số A[]. Thuật toán được mô tả chi tiết trong Hình 3.9.

a) Biểu diễn thuật toán

```
int Sequential-Search( int A[], int n, int x) {
        for (int i = 0; i < n; i++ ) {//duyệt trên dãy số A[n]
            if ( x == A[i] ) //nếu điều này xảy ra
                return (i); //i chính là vị trí của x trong dãy A[]
        }
        return (-1);//kết luận x không có mặt trong dãy A[]
}
```

Hình 3.9. Thuật toán Sequential-Search.

b) Độ phức tạp thuật toán

Độ phức tạp thuật toán là O(n), với n là số lượng phần tử trong dãy A[].

c) Kiểm nghiệm thuật toán

Ví dụ ta cần tìm x = 9 trong dãy A[] = {56, 3, 249, 518, 7, 26, 94, 651, 23, 9 }. Khi đó quá trình tìm kiếm được thể hiện như dưới đây.

56	3	249	518	7	26	94	651	23	9
56	**3**	249	518	7	26	94	651	23	9
56	3	**249**	518	7	26	94	651	23	9
56	3	249	**518**	7	26	94	651	23	9
56	3	249	518	**7**	26	94	651	23	9
56	3	249	518	7	**26**	94	651	23	9
56	3	249	518	7	26	**94**	651	23	9
56	3	249	518	7	26	94	**651**	23	9
56	3	249	518	7	26	94	651	**23**	9
56	3	249	518	7	26	94	651	23	**9**

d) Cài đặt thuật toán

```
#include <iostream>
using namespace std;
int Sequential_Search( int A[], int n, int x){
        for(int i=0; i<n; i++){
                if ( x == A[i])
                  return i;
        }
        return -1;
}
int main(void){
        int A[] = {9, 7, 12, 8, 6, 5};
        int x = 15, n = sizeof(A)/sizeof(A[0]);
        int t = Sequential_Search(A,n,x);
        if(t>=0)cout<<"\n Vị trí của x:"<<t;
        else cout<<"\n Không tìm thấy x";
}
```

3.6.2. Thuật toán tìm kiếm nhị phân

Thuật toán tìm kiếm nhị phân là phương pháp định vị phần tử *x* trong một danh sách A[] gồm n phần tử đã được sắp xếp. Quá trình tìm kiếm bắt đầu bằng việc chia danh sách thành hai phần. Sau đó, so sách x với phần từ ở giữa. Khi đó có 3 trường hợp có thể xảy ra:

Trường hợp 1: nếu x bằng phần tử ở giữa A[mid], thì mid chính là vị trí của x trong danh sách A[].

Trường hợp 2: Nếu x lớn hơn phần tử ở giữa thì nếu x có mặt trọng dãy A[] thì ta chỉ cần tìm các phần tử từ mid+1 đến vị trí thứ n.

Trường hợp 3: Nếu x nhỏ hơn A[mid] thì x chỉ có thể ở dãy con bên trái của dãy A[].

Lặp lại quá trình trên cho đến khi cận dưới vượt cận trên của dãy A[] mà vẫn chưa tìm thấy *x* thì ta kết luận *x* không có mặt trong dãy A[]. Thuật toán được mô tả chi tiết trong Hình 3.10.

a) Biểu diễn thuật toán

```
int Binary-Search( int A[], int n, int x) {//tìm vị trí của x trong dãy A[]
        int  low = 0;//cận dưới của dãy khóa
        int  hight = n-1;//cận trên của dãy khóa
        int  mid = (low+hight)/2; //phần tử ở giữa
        while ( low <=hight) { //lặp trong khi cận dưới vẫn nhỏ hơn cận trên
                if ( x > A[mid] ) //nếu x lớn hơn phần tử ở giữa
                        low = mid + 1; //cận dưới được đặt lên vị trí mid +1
                els if ( x < A[i] )
                        hight = mid -1;//cận trên lùi về vị trí mid-1
                else
                        return(mide); //đây chính là vị trí của x
                mid = (low + hight)/2; //xác định lại phần tử ở giữa
        }
        return(-1); //không thấy x trong dãy khóa A[].
}
```

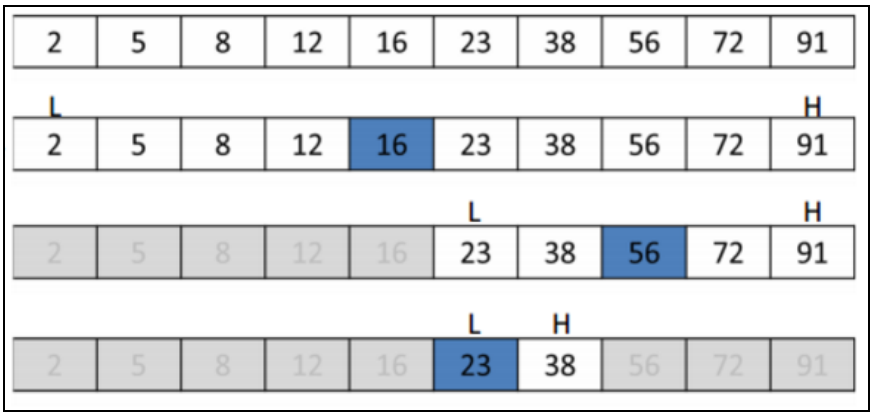
Hình 3.10. Thuật toán Binary-Search

b) Độ phức tạp thuật toán

Độ phức tạp thuật toán là O(log(n)), với n là số lượng phần tử của dãy A[].

c) Kiểm nghiệm thuật toán

Ví dụ ta cần tìm x = 23 trong dãy A[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91}. Khi đó quá trình được tiến hành như dưới đây.



d) Cài đặt thuật toán

```
#include <iostream>
using namespace std;
int Binary_Search( int A[], int n, int x) {//tìm vị trí của x trong dãy A[]
        int  low = 0;//cận dưới của dãy khóa
        int  hight = n-1; //cận trên của dãy khóa
        int  mid = (low+hight)/2; //vị trí phần tử ở giữa
        while ( low <=hight) { //lặp trong khi cận dưới nhỏ hơn cận trên
                if ( x > A[mid] ) //nếu x lớn hơn phần tử ở giữa
                        low = mid + 1; //cận dưới dịch lên vị  trí mid +1
                else if ( x < A[mid] ) //nếu x nhỏ hơn phần tử ở giữa
                        hight = mid -1; //cận trên dịch xuống vị  trí mid -1
                else
                        return(mid); //đây chính là vị trí của x
                mid = (low + hight)/2; //xác định lại vị trí ở giữa
        }
        return(-1); //không tìm thấy x trong A[].
}
int main(void){
        int A[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91};
        int x = 91, n = sizeof(A)/sizeof(A[0]);
        int t = Binary_Search(A,n,x);
        if(t>=0)cout<<"\n Vị trí của x:"<<t;
        else cout<<"\n không tìm thấy x";
}
```

3.6.3. Thuật toán tìm kiếm nội suy

Thuật toán tìm kiếm kiểu nội suy (interpolation search) là cải tiến của thuật toán tìm kiếm nhị phân. Thuật toán tìm kiếm nhị phân luôn thực hiện so sánh khóa với phần tử ở giữa. Trong đó, thuật toán tìm kiếm nội suy định vị giá trị so sánh tùy thuộc vào giá trị của khóa cần tìm. Bằng cách này, giá trị của khóa cần tìm kiếm dù ở đầu dãy, cuối dãy hay vị trí

bất kỳ thuật toán đều tìm được vị trí gần nhất để thực hiện so sánh. Thuật toán được mô tả chi tiết trong Hình 3.11.

a) Biểu diễn thuật toán

```
int Interpolation-Search(int A[], int x, int n){
        int low = 0,high = n – 1, mid;
        while ( A[low] <= x  &&  A[high] >= x){
                if (A[high] - A[low] == 0)   return (low + high)/2;
                mid = low + ((x - A[low]) * (high - low)) / (A[high] - A[low]);
                if (A[mid] < x) low = mid + 1;
                else if (A[mid] > x)  high = mid - 1;
                else    return mid;
        }
        if (A[low] == x)
                return low;
        return -1;
}
```

Hình 3.11. Thuật toán Interpolation-Search

b) Độ phức tạp thuật toán

Độ phức tạp trung bình của thuật toán tìm kiếm nội suy là O(log(n)), với n là số lượng phần tử của dãy A[]. Trong trường hợp xấu nhất, thuật toán có độ phức tạp là O(n).

c) Kiểm nghiệm thuật toán

Bạn đọc tự tìm hiểu phương pháp kiểm nghiệm thuật toán tìm kiếm nội suy trong các tài liệu liên quan.

d) Cài đặt thuật toán

```
#include <iostream>
using namespace std;
int interpolationSearch(int A[], int n, int x){//thuật toán tìm kiếm nội suy
        int L = 0, H = (n - 1);//cận dưới và cận trên của dãy
        if (x < A[L] || x > A[H])//nếu điều này xảy ra
                return -1; //chắc chắn x không có mặt trong dãy A[]
        while (L <= H){//lặp trong khi cận dưới bé hơn cận trên
                int pos = L + (((H-L) /(A[H]-A[L]))*(x - A[L])); //xác định vị trí
                if (A[pos] == x)//nếu vị trí đúng là x
                        return pos; //đây là vị trí cần tìm
                if (A[pos] < x)//nếu x lớn hơn A[pos]
                        L = pos + 1; //dịch cận dưới lên 1
                else //nếu x bé hơn A[pos]
                        H = pos - 1; //giảm cận trên đi 1
```

```
        }
        return -1; //kết luận không tìm thấy
}
int main(){
        int A[] = {10, 12, 13, 16, 31, 33, 35, 42, 47};
        int n = sizeof(A)/sizeof(A[0]);
        int x = 42; //phần tử cần tìm
        int index = interpolationSearch(A, n, x);
        if (index != -1)//nếu tìm thấy x
                cout<<"Vị trí:"<<index;
        else
                cout<<"Không tìm thấy x";
}
```

3.6.4. Thuật toán tìm kiếm Jumping

Thuật toán tìm kiếm Jumping được thực hiện bằng cách so sánh phần tử cần tìm với bước nhảy là một hàm mũ. Nếu khóa cần tìm lớn hơn phần tử tại bước nhảy ta nhảy tiếp một khoảng cũng là một hàm mũ. Trong trường hợp, khóa cần tìm nhỏ hơn phần tử tại bước nhảy, ta quay lại bước trước đó và thực hiện phép tìm kiếm tuyến tính thuần túy. Thuật toán được mô tả chi tiết trong Hình 3.12.

a) Biểu diễn thuật toán

```
int JumpSearch(int A[], int n, int x){ //thuật toán Jumping Search
        int step = (int) sqrt(n); //xác định bước nhảy
        int prev = 0; //giá trị khởi đầu bước nhảy trước
        int r = min(step,n)-1;//vị trí cần so sánh
        while (A[r]<x) {// lặp nếu x lớn hơn phần tử vị trí r
                prev = step; //lưu lại giá trị bước nhảy trước
                step += (int)sqrt(n);//tăng bước nhảy thêm khoảng căn bậc 2
                if (prev >= n) //nếu điều này xảy ra
                                return -1; //chắc chắn x không có trong dãy A[]
                r = min(step,n)-1; //tính toán lại vị trí cần so sánh
        }
        while (A[prev] < x){//thực hiện tìm kiếm tuyến tính thông thường
                prev++;
                if (prev == min(step, n))
                        return -1;
        }
        if (A[prev] == x) //nếu tìm thấy x
                        return prev;
        return -1;//không tìm thấy x
}
```

Hình 3.12. Thuật toán Jumping Search.

b) Độ phức tạp thuật toán

Độ phức tạp thuật toán trong trường hợp tốt, xấu nhất là $O(\sqrt{n})$. Trường hợp tốt nhất là O(log(n)), với n là số lượng phần tử của dãy A[].

c) Kiểm nghiệm thuật toán

Giả sử ta cần tìm vị trí của x = 55 trong dãy số A[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610}. Khi đó, thuật toán được thực hiện như sau:

Bước 1. Ta tìm được step=4. So sánh 53 với vị trí số A[3]=2<55.

Bước 2. Dịch chuyển step = 8, so sánh 53 với vị trí số A[7]=13<55.

Bước 3. Dịch chuyển step = 16, so sánh 53 với vị trí số A[15]=610>55.

Bước 4. Vì 610>55 nên ta trở về bước trước đó cộng thêm 1 là 9.

Bước 5. Tìm kiếm tuyến tính từ vị trí 9 đến 15 ta nhận được kết quả là 10.

d) Cài đặt thuật toán

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

```
int JumpSearch(int A[], int n, int x){ //thuật toán Jumping Search
        int step = (int) sqrt(n); //giá trị bước nhảy
        int prev = 0; //giá trị bước nhảy trước
        int r = min(step,n)-1;//vị trí cần so sánh
        while (A[r]<x) {//lặp trong khi A[r]<x
                prev = step; //lưu lại giá trị bước trước
                step += (int)sqrt(n);//tăng bước nhảy
                if (prev >= n) //nếu điều này xảy ra
                        return -1; //x chắc chắn không có trong A[]
                r = min(step,n)-1; //tính toán lại vị trí cần so sánh
        }
        while (A[prev] < x){//tìm kiếm tuyến tính thông thường
                prev++;
                if (prev == min(step, n))
                        return -1;
        }
        if (A[prev] == x) //nếu tìm thấy x
                 return prev;
        return -1;//không tìm thấy x
}
int main(void){
   int A[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21,
               34, 55, 89, 144, 233, 377, 610};
   int x = 233, n = sizeof(A)/sizeof(A[0]);
   int index = JumpSearch(A, n,x);
   if (index!=-1)       cout<<"\n Vị trí:"<<index;
   else cout<<"\n Không tìm thấy x";
}
```

BÀI TẬP

BÀI 1. SẮP XẾP ĐỔI CHỖ TRỰC TIẾP

Hãy thực hiện thuật toán sắp xếp đổi chỗ trực tiếp trên dãy N số nguyên. Ghi ra các bước thực hiện thuật toán. **Dữ liệu vào:** Dòng 1 ghi số N (không quá 100). Dòng 2 ghi N số nguyên dương (không quá 100). **Kết quả:** Ghi ra màn hình từng bước thực hiện thuật toán. Mỗi bước trên một dòng, các số trong dãy cách nhau đúng một khoảng trống.

Ví dụ:

Input	Output

4 5 7 3 2	Buoc 1: 2 7 5 3 Buoc 2: 2 3 7 5 Buoc 3: 2 3 5 7

BÀI 2. SẮP XẾP CHỌN

Hãy thực hiện thuật toán sắp xếp chọn trên dãy N số nguyên. Ghi ra các bước thực hiện thuật toán.

Dữ liệu vào: Dòng 1 ghi số N (không quá 100). Dòng 2 ghi N số nguyên dương (không quá 100).

Kết quả: Ghi ra màn hình từng bước thực hiện thuật toán. Mỗi bước trên một dòng, các số trong dãy cách nhau đúng một khoảng trống.

Ví dụ:

Input	Output
4 5 7 3 2	Buoc 1: 2 7 3 5 Buoc 2: 2 3 7 5 Buoc 3: 2 3 5 7

BÀI 3. SẮP XẾP CHÈN

Hãy thực hiện thuật toán sắp xếp chèn trên dãy N số nguyên. Ghi ra các bước thực hiện thuật toán.

Dữ liệu vào: Dòng 1 ghi số N (không quá 100). Dòng 2 ghi N số nguyên dương (không quá 100).

Kết quả: Ghi ra màn hình từng bước thực hiện thuật toán. Mỗi bước trên một dòng, các số trong dãy cách nhau đúng một khoảng trống.

Ví dụ:

Input	Output
4 5 7 3 2	Buoc 0: 5 Buoc 1: 5 7 Buoc 2: 3 5 7 Buoc 3: 2 3 5 7

BÀI 4. SẮP XẾP NỔI BỌT

Hãy thực hiện thuật toán sắp xếp nổi bọt trên dãy N số nguyên. Ghi ra các bước thực hiện thuật toán.

Dữ liệu vào: Dòng 1 ghi số N (không quá 100). Dòng 2 ghi N số nguyên dương (không quá 100).

Kết quả: Ghi ra màn hình từng bước thực hiện thuật toán. Mỗi bước trên một dòng, các số trong dãy cách nhau đúng một khoảng trống.

Ví dụ:

Input	Output
4 5 3 2 7	Buoc 1: 3 2 5 7 Buoc 2: 2 3 5 7

BÀI 5. MERGE SORT

Cho mảng A[] gồm N phần tử chưa được sắp xếp. Nhiệm vụ của bạn là sắp xếp các phần tử của mảng A[] theo thứ tự tăng dần bằng thuật toán Merge Sort.

Input:

- Dòng đầu tiên đưa vào số lượng bộ test T.
- Những dòng kế tiếp đưa vào các bộ test. Mỗi bộ test gồm hai phần: phần thứ nhất đưa vào số N tương ứng với số phần tử của mảng A[]; phần thứ 2 là N số của mảng A[]; các số được viết cách nhau một vài khoảng trống.
- T, N, A[i] thỏa mãn ràng buộc: $1 \le T \le 100$; $1 \le N, A[i] \le 10^6$.

Output:

- Đưa ra kết quả các test theo từng dòng.

Input	Output
2 5 4 1 3 9 7 10 10 9 8 7 6 5 4 3 2 1	1 3 4 7 9 1 2 3 4 5 6 7 8 9 10

BÀI 6. QUICK SORT

Cho mảng A[] gồm N phần tử chưa được sắp xếp. Nhiệm vụ của bạn là sắp xếp các phần tử của mảng A[] theo thứ tự tăng dần bằng thuật toán Quick Sort.

Input:

- Dòng đầu tiên đưa vào số lượng bộ test T.
- Những dòng kế tiếp đưa vào các bộ test. Mỗi bộ test gồm hai phần: phần thứ nhất đưa vào số N tương ứng với số phần tử của mảng A[]; phần thứ 2 là N số của mảng A[]; các số được viết cách nhau một vài khoảng trống.
- T, N, A[i] thỏa mãn ràng buộc: $1 \le T \le 100$; $1 \le N, A[i] \le 10^6$.

Output:

- Đưa ra kết quả các test theo từng dòng.

Input	Output
2 5 4 1 3 9 7 10 10 9 8 7 6 5 4 3 2 1	1 3 4 7 9 1 2 3 4 5 6 7 8 9 10

BÀI 7. TÌM KIẾM.

Cho mảng A[] gồm n phần tử đã được sắp xếp. Hãy đưa ra 1 nếu X có mặt trong mảng A[], ngược lại đưa ra -1.

Input:

- Dòng đầu tiên đưa vào số lượng bộ test T.
- Những dòng kế tiếp đưa vào các bộ test. Mỗi bộ test gồm hai dòng: dòng thứ nhất đưa vào n, X là số các phần tử của mảng A[] và số X cần tìm; dòng tiếp theo đưa vào n số A[i] ($1 \le i \le n$) các số được viết cách nhau một vài khoảng trống.
- T, n, A, X thỏa mãn ràng buộc: $1 \le T \le 100$; $1 \le N, X, A[i] \le 10^6$.

Output:

- Đưa ra kết quả mỗi test theo từng dòng.

Input:	Output:
2 5 16 2 4 7 9 16 7 98 1 22 37 47 54 88 96	1 -1

BÀI 8. TÌM SỐ CÒN THIẾU.

Cho mảng A[] gồm n-1 phần tử bao gồm các khác nhau từ 1, 2, .., n. Hãy tìm số không có mặt trong mảng A[].

Input:

- Dòng đầu tiên đưa vào số lượng bộ test T.
- Những dòng kế tiếp đưa vào các bộ test. Mỗi bộ test gồm hai dòng: dòng thứ nhất đưa vào n l; dòng tiếp theo đưa vào n-1 số A[i]; các số được viết cách nhau một vài khoảng trống.
- T, n, A thỏa mãn ràng buộc: $1 \le T \le 100$; $1 \le N, A[i] \le 10^7$.

Output:

- Đưa ra kết quả mỗi test theo từng dòng.

Input:	Output:
2 5 1 2 3 5 10 1 2 3 4 5 6 7 8 10	4 9

BÀI 9. TÌM KIẾM TRONG DÃY SẮP XẾP VÒNG.

Một mảng được sắp được chia thành hai đoạn tăng dần được gọi là mảng sắp xếp vòng. Ví dụ mảng A[] = { 5, 6, 7, 8, 9, 10, 1, 2, 3, 4} là mảng sắp xếp vòng. Cho mảng A[] gồm n phần tử, hãy tìm vị trí của phần tử x trong mảng A[] với thời gian log(n).

Input:

- Dòng đầu tiên đưa vào số lượng bộ test T.
- Những dòng kế tiếp đưa vào các bộ test. Mỗi bộ test gồm hai dòng: dòng thứ nhất đưa vào n và x; dòng tiếp theo đưa vào n số A[i]; các số được viết cách nhau một vài khoảng trống.
- T, n, A[i], x thỏa mãn ràng buộc: $1 \le T \le 100$; $1 \le N, x, A[i] \le 10^7$.

Output:

- Đưa ra kết quả mỗi test theo từng dòng.

Input:	Output:
2 10 3 5 6 7 8 9 10 1 2 3 4 10 3	9 3

1 2 3 4 5 6 7 8 9 10	

BÀI 10. SỐ NHỎ NHẤT VÀ SỐ NHỎ THỨ HAI.

Cho mảng A[] gồm n phần tử, hãy đưa ra số nhỏ nhất và số nhỏ thứ hai của mảng. Nếu không có số nhỏ thứ hai, hãy đưa ra -1.

Input:

- Dòng đầu tiên đưa vào số lượng bộ test T.
- Những dòng kế tiếp đưa vào các bộ test. Mỗi bộ test gồm hai dòng: dòng thứ nhất đưa vào n là số phần tử của mảng A[]; dòng tiếp theo đưa vào n số A[i]; các số được viết cách nhau một vài khoảng trống.
- T, n, A[i] thỏa mãn ràng buộc: $1 \leq T \leq 100$; $1 \leq N, A[i] \leq 10^7$.

Output:

- Đưa ra kết quả mỗi test theo từng dòng.

Input:	Output:
2 10 5 6 7 8 9 10 1 2 3 4 5 1 1 1 1 1	1 2 -1

CHƯƠNG 4. NGĂN XẾP, HÀNG ĐỢI, DANH SÁCH LIÊN KẾT

Như đã trình bày trong Chương 1, một kiểu dữ liệu trừu tượng (ADTs) được xác định khi ta xây dựng đầy đủ hai phần: cấu trúc dữ liệu cùng các phép toán trên cấu trúc dữ liệu đó. Nội dung của chương này trình bày ba kiểu dữ liệu trừu tượng quan trọng đó là danh sách liên kết, ngăn xếp và hàng đợi. Mỗi kiểu dữ liệu trừu tượng được xây dựng giải quyết lớp các vấn đề cụ thể của khoa học máy tính. Đối với người học, mỗi cấu trúc dữ liệu trừu tượng cần làm chủ được bốn điểm quan trọng sau:

- Định nghĩa cấu trúc dữ liệu ADTs.
- Biểu diễn cấu trúc dữ liệu ADTs.
- Thao tác (phép toán) trên cấu trúc dữ liệu ADTs.
- Ứng dụng của cấu trúc dữ liệu ADTs.

4.1. Danh sách liên kết đơn (Single Linked List)

Như ta đã biết mảng (*array*) là tập có thứ tự các phần tử có cùng chung một kiểu dữ liệu và được tổ chức liên tục nhau trong bộ nhớ. Ưu điểm lớn nhất của mảng là đơn giản và sử lý nhanh nhờ cơ chế tru cập phần tử trực tiếp vào các phần tử của mảng. Hạn chế lớn nhất của mảng là số lượng phần tử không thay đổi gây nên hiện tượng thừa bộ nhớ trong mọt số trường hợp và thiếu bộ nhớ trong một số trường hợp khác. Đối với một số bài toán có dữ liệu lớn, nhiều khi ta không đủ không gian nhớ tự do liên tục để cấp phát cho mảng. Để khắc phục hạn chế này ta có thể xây dựng kiểu dữ liệu danh sách liên kết đơn được định nghĩa, biểu diễn và thao tác như dưới đây.

4.1.1. Định nghĩa danh sách liên kết đơn

Tập hợp các node thông tin được tổ chức rời rạc trong bộ nhớ. Trong đó, mỗi node gồm coa hai thành phần:

- Thành phần dữ liệu (*data*): dùng để lưu trữ thông tin của node.
- Thành phần con trỏ (*pointer*): dùng để liên kết với node dữ liệu tiếp theo.

4.1.2. Biểu diễn danh sách liên kết đơn

Để biểu diễn danh sách liên kết đơn ta sử dụng phương pháp định nghĩa cấu trúc tự trỏ của các ngôn ngữ lập trình. Giả sử thành phần thông tin của mỗi node được định nghĩa như một cấu trúc Item như sau:

```
struct  Item {
        <Kiểu 1>    <Thành viên 1>;
        <Kiểu 2>    <Thành viên 2>;
        . . . . . ;
```

```
            <Kiểu N>    <Thành viên N>;
    };
```

Khi đó, danh sách liên kết đơn được định nghĩa như sau:

```
    struct node {
            Item    infor; //Thành phần thông tin của node;
            struct  node *next; //thành phần con trỏ của node
    } *Start; //Start là một danh sách liên kết đơn
```

Hình 3.1. Biểu diễn danh sách liên kết đơn

4.1.3. Thao tác trên danh sách liên kết đơn

Các thao tác trên danh sách liên kết đơn bao gồm:

- Tạo node rời rạc có giá trị value cho danh sách liên kết đơn
- Thêm một node vào đầu danh sách liên kết đơn.
- Thêm một node vào cuối danh sách liên kết đơn.
- Thêm node vào vị trí xác định trong danh sách liên kết đơn.
- Loại node trong sách liên kết đơn.
- Tìm node trong sách liên kết đơn.
- Sắp xếp node trong danh sách liên kết đơn.
- Sửa đổi nội dung node trong sách liên kết đơn.
- Đảo ngược các node trong danh sách liên kết đơn.
- Duyệt các node của danh sách liên kết đơn.

Để đơn giản, ta xem thành phần thông tin của node (Item) là một số nguyên. Khi đó, các thao tác trên danh sách liên kết đơn ta định nghĩa một lớp các thao tác như sau:

```
struct node { // biểu diễn node
        int info; //thành phần thông tin của node
        struct node *next; //thành phần con trỏ của node
}*start; // danh sách liên kết đơn: *start.
class single_linked_list { //biểu diễn lớp danh sách liên kết đơn
   public:
      node* create_node(int);//Tạo một node cho danh sách liên kết đơn
      void insert_begin(); //thêm node vào đầu DSLKĐ
      void insert_pos(); //thêm node tại vị trí cụ thể trên DSLKĐ
      void insert_last(); //thêm node vào cuối DSLKĐ
      void delete_pos(); //loại node tại vị trí cho trước trên DSLKĐ
      void sort(); //sắp xếp nội dung các node theo thứ tự tăng dần
```

```
        void search(); //tìm kiếm node trên DSLKĐ
        void update(); //sửa đổi thông tin của node trên DSLKĐ
        void reverse(); //đảo ngược danh sách liên kết đơn
        void display(); //hiển thị nội dung DSLKĐ
        single_linked_list(){//constructor của lớp single linked list.
                        start = NULL;//chú ý start là biến toàn cục
        }
};
```

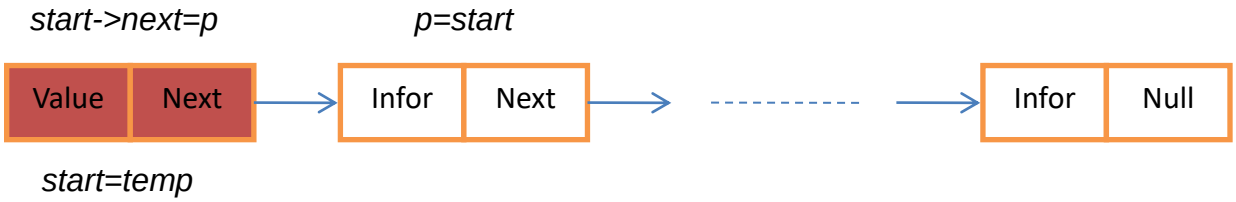
Thao tác: *tạo một node rời rạc có giá trị value cho DSLKĐ***.**

```
node *single_linked_list::create_node(int value){
        struct node *temp; //khai báo hai con trỏ node *temp
        temp = new(struct node); //cấp phát miền nhớ cho temp
        if (temp == NULL){ //nếu không đủ không gian nhớ
                cout<<"không đủ bộ nhớ để cấp phát"<<endl;
                return 0;
        }
        else {
                temp->info = value;//thiết lập thông tin cho node temp
                temp->next = NULL; //thiết lập liên kết cho node temp
                return temp;//trả lại node temp đã được thiết lập
        }
}
```

Thao tác: ***thêm node vào đầu DSLKĐ*****.**

```
void single_linked_list::insert_begin(){ //chèn node vào đầu DSLKĐ
        int value; cout<<"Nhập giá trị node:"; cin>>value; //giá trị node cần chèn
        struct node *temp, *p; //sử dụng hai con trỏ temp và p
        temp = create_node(value);//tạo một node rời rạc có giá trị value
        if (start == NULL){ //nếu danh sách liên kết rỗng
                start = temp; //danh sách liên kết chính là node temp
                start->next = NULL; //không có liêt kết với node khác
        }
        else { //nếu danh sách không rỗng
                p = start; //p trỏ đến node đầu của start
                start = temp; //node  temp trở thành node đầu tiên của start
                start->next = p;//các node còn lại chính là p
        }
}
```

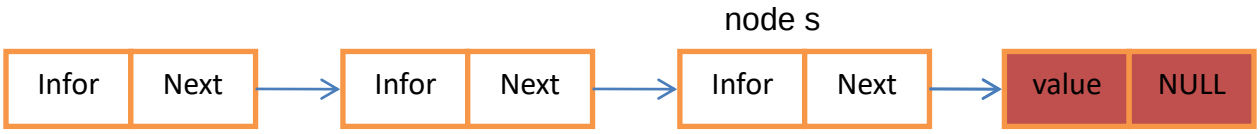
Hình 3.2. dưới đây mô tả phép thêm node vào đầu danh sách liên kết đơn.



Hình 3.2. Thêm node vào đầu danh sách liên kết đơn

Thao tác thêm node vào cuối danh sách liên kết đơn:

```
void single_linked_list::insert_last(){//thêm node vào cuối DSLKĐ
        int value;
        cout<<"Nhập giá trị cho node: ";cin>>value; //nhập giá trị node
        struct node *temp, *s; //sử dụng hai con trỏ temp và s
        temp = create_node(value);//tạo node rời rạc có giá trị value
        if(start==NULL) {//trường hợp DSLKĐ rỗng
                start = temp;
                temp->next=NULL;
        }
        s = start; //s trỏ đến node đầu danh sách
        while (s->next != NULL){ //di chuyển s đến node cuối cùng
                s = s->next;
        }
        temp->next = NULL; //temp không chỏ đi đâu nữa
        s->next = temp; //thiết lập liên kết cho s
        cout<<"Hoàn thành thêm node vào cuối"<<endl;
}
```

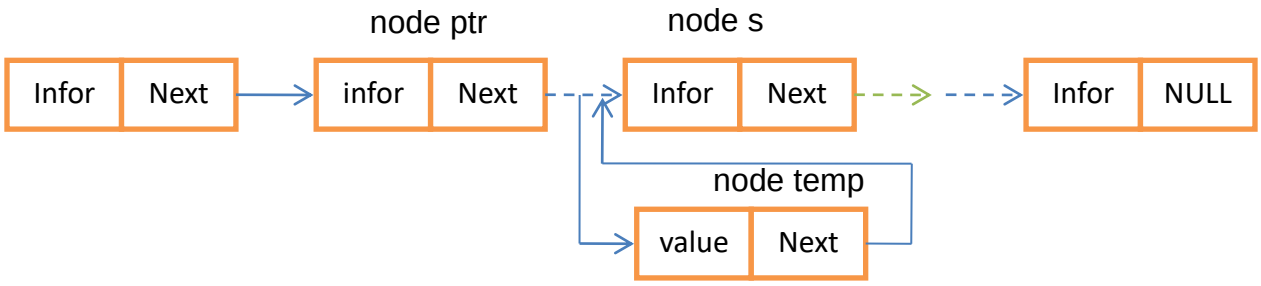


Hình 3.3. Thêm node vào cuối danh sách liên kết đơn

Thao tác thêm node vào vị trí pos của danh sách liên kết đơn:

```
void single_linked_list::insert_pos(){//thêm node vào vị trí pos
        int value, pos, counter = 0; cout<<"Nhap gia tri node:";cin>>value;
        struct node *temp, *s, *ptr; //sử dụng ba con trỏ node
        temp = create_node(value);//tạo node rời rạc có giá trị value
        cout<<"Nhập vị trí node cần thêm: ";cin>>pos;
        int i; s = start; //s trỏ đến node đầu tiên
        while (s != NULL){ //đếm số node của DSLKĐ
                s = s->next; counter++;
```

```
        }
        if (counter==0) {//trường hợp DSLK đơn rỗng
                cout<<"Danh sách rỗng"; return;
        }
        if (pos == 1){ //nếu pos là vị trí đầu tiên
                if (start == NULL){ //trường hợp DSLKĐ rỗng
                                start = temp; start->next = NULL;
                }
                else { //thêm node temp vào đầu DSLKĐ
                        ptr = start;  start = temp;  start->next = ptr;
                }
        }
        else if (pos > 1  && pos <= counter){ //trường hợp pos hợp lệ
                s = start; //s trỏ đến node đầu tiên
                for (i = 1; i < pos; i++){  //di chuyển đến node pos-1
                        ptr = s;  s = s->next;
                }
                ptr->next = temp;  temp->next = s; //thiết lập liên kết cho node
        }
        else { cout<<"Vượt quá giới hạn DSLKĐ"<<endl; }
}
```

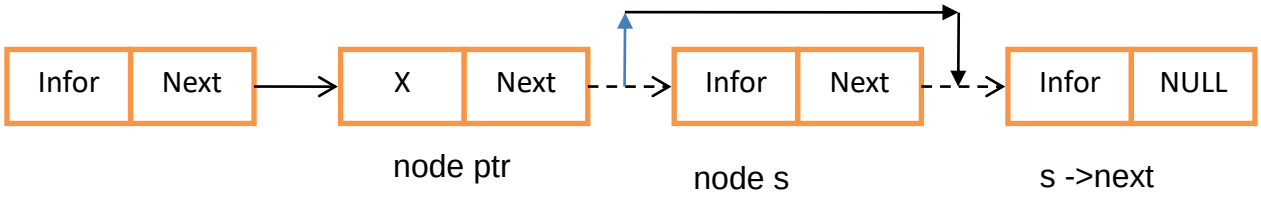


Hình 3.4. Thêm node vào vị trí pos

Thao tác loại node tại vị trí pos:

```
void single_linked_list::delete_pos(){//loại node ở vị trí pos
        int pos, i, counter = 0;
        if (start == NULL){  //nếu danh sách liê kết đơn rỗng
                cout<<"Không thực hiện được"<<endl; return;
        }
        cout<<"Vị trí cần loại bỏ:";cin>>pos;
        struct node *s, *ptr;  s = start; //s trỏ đến đầu danh sách
        if (pos == 1){//nếu vị trí loại bỏ là node đầu tiên
```

```
            start = s->next; s->next=NULL; free(s);
        }
        else {
            while (s != NULL) {  //đếm số node của DSLKĐ
                s = s->next; counter++;
            }

            if (pos > 0 && pos <= counter){ //nếu vị trí pos hợp lệ
                    s = start;//s trỏ đến node đầu của danh sách
                    for (i = 1;i < pos; i++){ //di chuyển đến vị trí pos-1
                        ptr = s;  s = s->next;
                    }
                    ptr->next = s->next; //thiết lập liên kết cho node
            }
            else { cout<<"Vi tri ngoai danh sach"<<endl; }
            free(s); //giải phóng s
            cout<<"Node da bi loai bo"<<endl;
        }
}
```



Hình 3.5. Thao tác loại node ở vị trí pos

Thao tác sửa đổi nội dung của node:

```
void single_linked_list::update(){//sửa đổi thông tin của node
        int value, pos, i;
        if (start == NULL){ //nếu danh sách LKĐ rỗng
            cout<<"Không thực hiện được"<<endl;  return;
        }
        cout<<"Nhập vị trí node cần sửa:";cin>>pos;
        cout<<"Giá trị mới của node:";cin>>value;
        struct node *s, *ptr; //sử dụng hai con trỏ s và ptr
        s = start; //s trỏ đến node đầu tiên
        if (pos == 1) { start->info = value;} //sửa luôn node đầu tiên
        else { //nếu pos không phải là node đầu tiên
            for (i = 0;i < pos - 1;i++){//chuyển s đến vị trí pos-1
```

```
                if (s == NULL){//Nếu s là node cuối cùng
                        cout<<"Vị trí "<<pos<<" không hợp lệ"; return;
                }
                s = s->next;
            }
            s->info = value;  //Sửa đổi thông tin cho node
        }
        cout<<"Hoàn thành việc sửa đổi"<<endl;
}
```

Thao tác duyệt danh sách liên kết đơn:

```
void single_linked_list::display(){//hiển thị nội dung DSLKĐ
        struct node *temp; //sử dụng một con trỏ temp
        if (start == NULL){ // nếu danh sách rỗng
                cout<<"Có gì đâu mà hiển thị"<<endl;
                return;
        }
        temp = start; //temp trỏ đến node đầu trong DSLKĐ
        cout<<"Nội dung DSLKĐ: "<<endl;
        while (temp != NULL) { //lặp cho đến node cuối cùng
                cout<<temp->info<<"->"; //hiển thị thông tin node
                temp = temp->next; //di chuyển đến node tiếp theo
        }
        cout<<"NULL"<<endl; //node cuối cùng là NULL
}
```

Thao tác tìm node trong danh sách liên kết đơn:

```
void single_linked_list::search(){//Tìm kiếm node
        int value, pos = 0; bool flag = false;
        if (start == NULL){//nếu danh sách rỗng
                cout<<"ta không có gì để tìm"<<endl;
                return;
        }
        cout<<"Nội dung node cần tìm:";cin>>value;
        struct node *s; s = start;//s trỏ đến đầu danh sách
        while (s != NULL){ pos++;
                if (s->info == value){//Nếu s->infor là value
                        flag = true;
                        cout<<"Tìm thấy "<<value<<" tại vị trí "<<pos<<endl;
```

```
            }
            s = s->next;
        }
        if (!flag) {//đến cuối vẫn không thấy
            cout<<"Giá trị"<<value<<"không tồn tại"<<endl;
        }
}
```

Thao tác sắp xếp các node trong danh sách liên kết đơn:

```
void single_linked_list::sort(){//sắp xếp theo nội dung các node
        struct node *ptr, *s; //sử dụng hai con trỏ ptr và s
        int value; //giá trị trung gian
        if (start == NULL){//nếu danh sách rỗng
            cout<<"không có gì để sắp xếp"<<endl;
            return;
        }
        ptr = start;//ptr trỏ đến node đầu danh sách
        while (ptr != NULL){ //lặp trong khi ptr khác rỗng
            for (s = ptr->next; s !=NULL; s = s->next){ //s là node tiếp theo
                if (ptr->info > s->info){//nếu điều này xảy ra
                    value = ptr->info;//tráo đổi nội dung hai node
                    ptr->info = s->info;
                    s->info = value;
                }
            }
            ptr = ptr->next;
        }
}
```

Thao tác đảo ngược các node của DSLKĐ:

```
void single_linked_list::reverse(){//đảo ngược danh sách
        struct node *ptr1, *ptr2, *ptr3; //sử dụng ba con trỏ node
        if (start == NULL) {//Nếu danh sách rỗng
            cout<<"ta không cần đảo"<<endl; return;
        }
        if (start->next == NULL){//Nếu danh sách chỉ có một node
             cout<<"đảo ngược là chính nó"<<endl; return;
        }
        ptr1 = start; //ptr1 trỏ đến node đầu tiên
```

```
        ptr2 = ptr1->next;//ptr2 trỏ đến node kế tiếp của ptr1
        ptr3 = ptr2->next;//ptr3 trỏ đến nod kế tiếp của ptr2
        ptr1->next = NULL;//Ngắt liên kết ptr1
        ptr2->next = ptr1;//node ptr2 bây giờ đứng trước node ptr1
        while (ptr3 != NULL){//Lặp nếu ptr3 khác rỗng
                ptr1 = ptr2;  //ptr1 lại bắt đầu tại vị trí ptr2
                ptr2 = ptr3;  //ptr2 bắt đầu tại vị trí ptr3
                ptr3 = ptr3->next; //ptr3 trỏ đến node kế tiếp
                ptr2->next = ptr1; //Thiết lập liên kết cho ptr2
        }
        start = ptr2; //node đầu tiên bây giờ là ptr2
}
```

//Chương trình cài đặt các thao tác trên danh sách liên kết đơn:

```
#include<iostream>
using namespace std;
struct node { // biểu diễn danh sách liên kết đơn
        int info; //thành phần thông tin
        struct node *next; //thành phần liên kết
}*start;
class single_linked_list { //biểu diễn lớp single_linked_list
   public:
      node* create_node(int);//tạo node rời rạc có giá trị value
      void insert_begin();//thêm node vào đầu danh sách liên kết đơn
      void insert_pos();//thêm node vào vị trí pos trong danh sách liên kết đơn
      void insert_last();//thêm node vào cuối danh sách liên kết đơn
      void delete_pos();//loại node tại vị trí pos của sách liên kết đơn
      void sort();//sắp xếp theo giá trị node cho danh sách liên kết đơn
      void search();//tìm node trong danh sách liên kết đơn
      void update(); //cập nhật thông tin cho node
      void reverse(); //đảo ngược các node trong danh sách liên kết đơn
      void display(); //duyệt danh sách liên kết đơn
      single_linked_list(){//constructor của lớp
                start = NULL;
      }
};
```

```
node *single_linked_list::create_node(int value){//tạo node rời rạc có giá trị value
        struct node *temp, *s; //sử dụng hai con trỏ node *temp, *s
        temp = new(struct node); //cấp phát không gian nhớ cho temp
        if (temp == NULL){ //nếu không đủ gian nhớ để cấp phát
                cout<<"Không đủ bộ nhớ"<<endl;
                return 0;
        }
        else {
                temp->info = value;//thiết lập thành phần thông tin
                temp->next = NULL; //thiết lập thành phần liên kết
                return temp;//trả lại con trỏ node
        }
}
void single_linked_list::insert_begin(){ //thêm node vào đầu
        int value; //giá trị node cần thêm
        cout<<"Nhập giá trị node:"; cin>>value;
        struct node *temp, *p;
        temp = create_node(value);//tạo node rời rạc có giá trị value
        if (start == NULL){//nếu danh sách liên kết đơn rỗng
                start = temp;//start chính là temp;
                start->next = NULL; //thiết lập thành phần liên kết cho start
        }
        else { //trường hợp danh sách liên kết không rỗng
                p = start; //con trỏ p trỏ đến start
                start = temp;//start trỏ đến temp
                start->next = p;//thiết lập lại liên kết cho start
        }
        cout<<"Hoàn thành thêm node vào đầu"<<endl;
}
void single_linked_list::insert_last(){//thêm node vào cuối danh sách liên kết đơn
        int value; //giá trị của node cần thêm
        cout<<"Nhap gia tri cho node: ";cin>>value;
        struct node *temp, *s; //sử dụng hai con trỏ node
        temp = create_node(value);//tạo node rời rạc có giá trị value
        if(start==NULL){ //nếu danh sách rỗng
                start = temp; temp->next=NULL;  return;
        }
```

```
        s = start; //s trỏ đến start
        while (s->next != NULL){ //di chuyển s đến node cuối cùng của DSLKĐ
                s = s->next;
        }
        temp->next = NULL; //temp trỏ đến null
        s->next = temp; //thiết lập liên kết cho s
        cout<<" Hoàn thành thêm node vào cuối "<<endl;
}
void single_linked_list::insert_pos(){//Them node vao vi tri cho truoc
        int value, pos, counter = 0;
        cout<<"Nhap gia tri node:";cin>>value;
        struct node *temp, *s, *ptr;
        temp = create_node(value);//tạo node rời rạc có giá trị value
        cout<<"Vi tri node can them: ";cin>>pos;
        int i; s = start; //s trỏ đến start
        while (s != NULL){//đếm số node của DSLKĐ là counter
                s = s->next; counter++;
        }
        if (pos == 1){ //nếu pos là node đầu tiên
                if (start == NULL){//trường hợp DSLKĐ rỗng
                        start = temp;//start trỏ đến temp
                        start->next = NULL; //thiết lập liên kết cho start là null
                }
                else { //trường hợp DSLKĐ không rỗng
                        ptr = start;//ptr trỏ đến start
                        start = temp;//start trỏ đến temp
                        start->next = ptr; //start liên kết với ptr
                }
        }
        else if (pos > 1  && pos <= counter){ //trường hợp vị trí pos hợp lệ
                s = start; //s trỏ đến start
                for (i = 1; i < pos; i++){//di chuyển ptr đến node sau vị trí pos
                        ptr = s;
                        s = s->next;
                }
                ptr->next = temp;//thiết lập liên kết cho ptr là temp
                temp->next = s; //thiết lập liên kết cho temp là s
```

```
        }
        else {//trường hợp vị trí pos không hợp lệ
                cout<<" Vị trí pos không hợp lệ "<<endl;
        }
}
void single_linked_list::sort(){//sắp xếp các node
        struct node *ptr, *s;
        int value;
        if (start == NULL){//trường hợp danh sách rỗng
                cout<<"Không phải làm gì"<<endl;
                return;
        }
        ptr = start;//ptr trỏ đến đầu danh sách
        while (ptr != NULL){ //bắt đầu sắp xếp
                for (s = ptr->next;s !=NULL;s = s->next){
                        if (ptr->info > s->info){
                                value = ptr->info;
                                ptr->info = s->info;
                                s->info = value;
                        }
                }
                ptr = ptr->next;
        }
}
void single_linked_list::delete_pos(){//loại node bất kỳ
        int pos, i, counter = 0;
        if (start == NULL){ //nếu danh sách rỗng
                cout<<"Danh sách rỗng"<<endl;
                return;
        }
        cout<<"Vị trí cần loại:";cin>>pos;
        struct node *s, *ptr;
        s = start; //s trỏ đến đầu danh sách
        if (pos == 1){ //nếu loại node đầu tiên
                start = s->next; free(s); return;
        }
```

```
        else { //trong trường hợp khác
                while (s != NULL) {//đếm số node
                        s = s->next;
                        counter++;
                }
                if (pos > 0 && pos <= counter){ //nếu vị trí hợp lệ
                        s = start;
                        for (i = 1;i < pos;i++){ //di chuyển s đến vị trí pos
                                ptr = s;
                                s = s->next;
                        }
                        ptr->next = s->next; //thiết lập liên kết cho ptr
                        free(s);
                }
                else {//nếu vị trí không hợp lệ
                        cout<<" nếu vị trí không hợp lệ "<<endl;
                }
        }
}
void single_linked_list::update(){//sửa đổi thông tin node
        int value, pos, i;
        if (start == NULL){//nếu danh sách rỗng
                cout<<"Danh sách rỗng"<<endl;
                return;
        }
        cout<<"Vị trí node cần cập nhật:";cin>>pos;
        cout<<"Giá trị mới của node:";cin>>value;
        struct node *s, *ptr;
        s = start; //s trỏ đến node đầu tiên
        if (pos == 1){//nếu là node đầu tiên
                start->info = value; //cập nhật lại giá trị node
        }
        else { //trong trường hợp khác
                for (i = 0;i < pos - 1;i++){//di chuyển s đến vị trí pos
                        if (s == NULL){//nếu s là rỗng
                                cout<<"Vị trí "<<pos<<" không hợp lệ"; return;
                        }
```

```
                s = s->next;
            }
            s->info = value;  //cập nhật lại giá trị cho node s
        }
        cout<<"Cập nhật thành công"<<endl;
}
void single_linked_list::search(){//Tim kiem node
        int value, pos = 0;
        bool flag = false;
        if (start == NULL){//nếu danh sách rỗng
            cout<<"Danh sách rỗng"<<endl;
            return;
        }
        cout<<"Giá trị cần tìm:";cin>>value;
        struct node *s; s = start;//s trỏ đến đầu danh sách
        while (s != NULL){ pos++;
            if (s->info == value){//nếu tìm thấy value
                flag = true;
                cout<<"Giá trị "<<value<<" tại vị trí "<<pos<<endl;
            }
            s = s->next;
        }
        if (!flag)
            cout<<"Giá trị "<<value<<" không tồn tại trong danh sách"<<endl;
}
void single_linked_list::reverse(){//đảo ngược danh sách
        struct node *ptr1, *ptr2, *ptr3;
        if (start == NULL) {//nếu danh sách rỗng
            cout<<"Danh sách rỗng"<<endl;
            return;
        }
        if (start->next == NULL){//nếu danh sách chỏ có một node
            return;
        }
        ptr1 = start; //ptr1 trỏ đến node đầu tiên
        ptr2 = ptr1->next;// ptr2 trỏ đến node tiếp theo
        ptr3 = ptr2->next;// ptr3 trỏ đến node tiếp theo
```

```
        ptr1->next = NULL;//ngắt liên kết của ptr1
        ptr2->next = ptr1;//ptr1 trở thành node kế tiếp của ptr2
        while (ptr3 != NULL){//lặp nếu ptr3 không rỗng
                ptr1 = ptr2;
                ptr2 = ptr3;
                ptr3 = ptr3->next;
                ptr2->next = ptr1;
        }
        start = ptr2;
}
void single_linked_list::display(){//duyệt danh sách
        struct node *temp;
        if (start == NULL){
                cout<<"Danh sách rỗng"<<endl;
                return;
        }
        temp = start;
        cout<<"Nội dung danh sách: "<<endl;
        while (temp != NULL) {
                cout<<temp->info<<"->";
                temp = temp->next;
        }
        cout<<"NULL"<<endl;
}
int main(){//chương trình chính
        int choice;
        single_linked_list X;//X là đối tượng DSLKĐ
        start = NULL; //khởi tạo start
        while (1){
                cout<<endl<<"---------------------------------"<<endl;
                cout<<endl<<"CÁC THAO TÁC TRÊN DSLKĐ"<<endl;
                cout<<endl<<"---------------------------------"<<endl;
                cout<<"1. Thêm node vào đầu danh sách"<<endl;
                cout<<"2. Thêm node vào cuối danh sách "<<endl;
                cout<<"3. Thêm node vào vị trí bất kỳ "<<endl;
                cout<<"4. Sắp xếp theo giá trị các ode"<<endl;
                cout<<"5. Loại node ở vị trí bất kỳ"<<endl;
```

```
cout<<"6. Cập nhật nội dung node"<<endl;
cout<<"7. Tìm kiếm node theo giá trị"<<endl;
cout<<"8. Duyệt danh sách"<<endl;
cout<<"9. Đảo ngược danh sách "<<endl;
cout<<"0.Thoát "<<endl;
cout<<"Lựa chọn chức năng: "; cin>>choice;
switch(choice){
        case 1:
                cout<<" Thêm node vào đầu danh sách: "<<endl;
                X.insert_begin();cout<<endl;     break;
        case 2:
                cout<<" Thêm node vào cuối danh sách: "<<endl;
                X.insert_last();cout<<endl; break;
        case 3:
                cout<<" Thêm node vào vị trí bất kỳ:"<<endl;
                X.insert_pos();cout<<endl;   break;
        case 4:
                cout<<"Sắp xếp nội dung node: "<<endl;
                X.sort();cout<<endl;    break;
        case 5:
                cout<<"Loại bỏ node: "<<endl;
                X.delete_pos();    break;
        case 6:
                cout<<"Cập nhật giá trị node:"<<endl;
                X.update();cout<<endl;    break;
        case 7:
                cout<<"Tìm kiếm node: "<<endl;
                X.search();cout<<endl;    break;
        case 8:
                cout<<"Duyệt danh sách:"<<endl;
                X.display();cout<<endl;   break;
        case 9:
                cout<<"Đảo ngược danh sách"<<endl;
                X.reverse();cout<<endl; break;
        case 0:
                cout<<"Thoát..."<<endl;
                exit(1);   break;
```

```
                default:
                        cout<<"Lựa chọn sai"<<endl;    break;
                }
        }
}
```

4.1.4. Ứng dụng của danh sách liên kết đơn

Ta có thể sử dụng danh sách liên đơn để giải quyết những vấn đề sau:

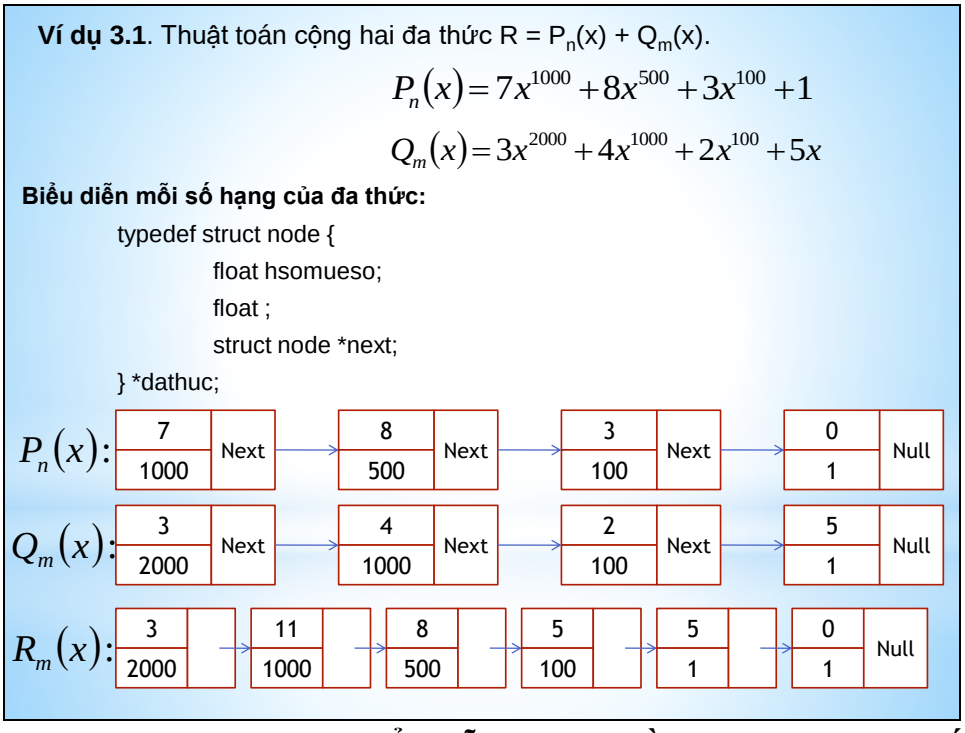
- Sử dụng danh sách liên kết đơn trong việc xây dựng các lược đồ quản lý bộ nhớ.
- Sử dụng danh sách liên kết đơn trong việc xây dựng ngăn xếp.
- Sử dụng danh sách liên kết đơn trong việc xây dựng hàng đợi.
- Sử dụng danh sách liên kết đơn biểu diễn cây.
- Sử dụng danh sách liên kết đơn biểu diễn đồ thị.
- Sử dụng danh sách liên kết đơn trong biểu diễn tính toán.

Ví dụ 3.1. Xây dựng phép cộng giữa hai đa thức.

Input : đa thức $P_n(x)$ bậc n và đa thức $Q_m(x)$ bậc m.

Output: đa thức $R = P_n(x) + Q_m(x)$.

Lời giải. Ta có thể sử dụng danh sách liên kết đơn để biểu diễn và xây dựng thuật toán cộng hai đa thức như sau.



Hình 3.6. Biểu diễn đa thức bằng danh sách liên kết đơn

```
Thuật toán Cong_Dathuc ( Dathuc *P, Dathuc *Q ):
    Bước 1 (Khởi tạo): R = ∅;
    Bước 2 (lặp ):
        while ( P≠∅ && Q≠∅ ) {
            if( P→Somu > Q → Somu ) {
                R = R →P; P = P →next;
            }
            else if (P→Somu < Q → Somu ) {
                R = R →Q; Q = Q →next;
            }
            else {  P → heso = P → heso + Q → heso;
                R = R →P; P = P →next; Q = Q →next;
            }
        }
    Bước 3 (Hoàn chỉnh đa thức):
        if (P ≠∅ ) R = R = R →P;
        if (Q ≠∅ ) R = R = R →Q;
    Bước 4 (Trả lại kết quả):
        Return (R);
```

Hình 3.7. Thuật toán cộng hai đa thức

4.2. Danh sách liên kết kép (double linked list)

Hạn chế lớn nhất đối với danh sách liên kết đơn là vấn đề tìm kiếm. Phương pháp tìm kiếm duy nhất ta có thể cài đặt được trên danh sách liên kết đơn là tìm kiếm tuyến tính. Từ một node bất kỳ, hoặc ta quay lại từ node đầu tiên hoặc chỉ được phép tìm theo hướng con trỏ next. Để hạn chế điều này ta có thể xây dựng kiểu dữ liệu danh sách liên kết kép như sau.

4.2.1. Định nghĩa

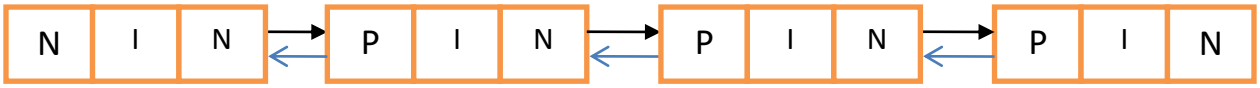
Danh sách liên kết kép là tập các node dữ liệu được tổ chức rời rạc nhau trong bộ nhớ, mỗi node gồm có ba thành phần:

- Thành phần thông tin (infor): dùng để lưu trữ thông tin của node.
- Thành phần liên kết trước (next): dùng để liên kết với node dữ liệu phía trước.
- Thành phần liên kết sau (previous): dùng để liên kết với node dữ liệu phía sau.

4.2.2. Biểu diễn

Sử dụng phương pháp biểu diễn đệ qui của các cấu trúc tự trỏ ta định nghĩa danh sách liên kết kép như sau:

```
struct node { //định nghĩa node
        Item   Infor; //thành phần dữ liệu của node
        struct  node *next; //thành phần con trỏ trước
        struct  node *prev; //thành phần con trỏ sau
}*start; //đây là danh sách liên kết kép
```



Hình 3.8. Biểu diễn danh sách liên kết kép.

4.2.3. Các thao tác trên danh sách liên kết kép

Các thao tác trên danh sách liên kết kép bao gồm:

- Tạo node rời rạc có giá trị value cho danh sách liên kết kép.
- Thêm node vào đầu danh sách liên kết kép.
- Thêm node vào cuối danh sách liên kết kép.
- Thêm node vào vị trí pos trong danh sách liên kết kép.
- Loại node tại vị trí pos của danh sách liên kết kép.
- Sắp xếp nội dung các node của danh sách liên kết kép.
- Tìm kiếm node có giá trị value trên danh sách liên kết kép.
- Duyệt trái danh sách liên kết kép.
- Duyệt phải danh sách liên kết kép.
- Đảo ngược các node trong danh sách liên kết kép.

Nội dung cụ thể các thao tác được thể hiện như chương trình dưới đây:

```
#include<iostream>
using namespace std;
//Biểu diễn danh sách liên kết kép:
struct node { //biểu diễn node
        int info; //thành phần thông tin của node
        struct node *next; //thành phần con trỏ đến node trước
        struct node *prev; //thành phần con trỏ đến node sau
}*start; //đây là danh sách liên kết kép
//Lớp biểu diễn các thao tác trên danh sách liên kết kép:
class double_linked_list { //Bieu dien lop
   public:
      node* create_node(int);//tạo node rời rạc có giá trị value
      void insert_begin(); //thêm node vào đầu DSLKK
      void insert_pos();//thêm node vào vị trí pos trong DSLKK
```

```
        void insert_last();//thêm node vào cuối DSLKK
        void delete_pos();//loại node tại vị trí pos của DSLKK
        void sort();//sắp xếp nội dung các node
        void search(); //tìm kiếm node có giá trị value
        void update();//cập nhật thông tin cho node
        void right_travel(); //duyệt phải DSLKK từ node đầu tiên
        void left_travel();//duyệt trái DSLKK từ node cuối cùng
        double_linked_list(){//Constructor của lớp
           start = NULL;
        }
};
```

//Thao tác tạo node rời rạc có giá trị value:

```
node *double_linked_list::create_node(int value){
        struct node *temp; //khai báo con trỏ node temp
        temp = new(struct node); //cấp phát miền nhớ cho temp
        if (temp == NULL){ //nếu không đủ bộ nhớ
                cout<<"Không đủ bộ nhớ để cấp phát"<<endl;
                return 0;
        }

        else {//tạo node rời rạc có giá trị value
                temp->info = value;//thiết lập thành phần thông tin
                temp->next = NULL; //thiết lập liên kết trước cho node
                temp->prev = NULL; //thiết lập liên kết sau cho node
                return temp;//trả lại con trỏ node
        }
}
```

//Thêm node vào đầu danh sách liên kết kép:

```
void double_linked_list::insert_begin(){ //thêm node vào đầu DSLKK
        int value; //giá trị của node là value
        cout<<"Nhap gia tri node:"; cin>>value;
        struct node *temp, *p;
        temp = create_node(value);// tạo node rời rạc có giá trị value
        if (start == NULL){//trường hợp danh sách rỗng
                start = temp; //start chính làm temp
                start->next = NULL;  //thiết lập liên kết trước
                start->prev = NULL; //thiết lập liên kết sau
```

```
        }
        else {//trường hợp danh sách không rỗng
                p = start; //p trỏ đến node đầu tiên của danh sách
                start = temp; //start trỏ đến temp
                start->next = p; //thiết lập liên kết trước cho start
                p->prev = start; //thiết lập liên kết sau cho start
        }
        cout<<"Hoàn thành thêm node vào đầu"<<endl;
}
```

//Thêm node vào cuối DSLKK

```
void double_linked_list::insert_last(){//thêm node vào cuối DSLKK
        int value; //giá trị node cần thêm
        cout<<"Nhập giá trị cho node: ";cin>>value;
        struct node *temp, *s; //sử dụng hai con trỏ node
        temp = create_node(value);//tạo node temp rời rạc có giá trị value
        s = start; //s trỏ đến start
        while (s->next != NULL){ //di chuyển đến node cuối cùng
                s = s->next;
        }

        temp->next = NULL; //thiết lập liên kết trước cho temp
        s->next = temp; //thiết lập liên kết trước cho s
        temp->prev = s; //thiết lập liên kết sau cho temp
        cout<<"Hoàn thành việc thêm node vào cuối"<<endl;
}
```

//**Thêm node vào vị trí bất kỳ:**

```
void double_linked_list::insert_pos(){//thêm node và vị trí bất kỳ
        int value, pos, counter = 0;
        cout<<"Giá trị node cần thêm:";cin>>value;
        struct node *temp, *s, *ptr;
        temp = create_node(value);//tạo node rời rạc có giá trị value
        cout<<"Vị trí node cần thêm: ";cin>>pos;
        int i; s = start;  //s trỏ đến node đầu trong danh sách
        while (s != NULL){//xác định counter là số node trong danh sách
                s = s->next; counter++;
        }
        if (pos == 1){ //nếu ta thêm vào node đầu tiên
```

```
            if (start == NULL){//trường hợp danh sách rỗng
                start = temp; //start lấy luôn là temp
                start->next = NULL; //thiết lập liên kết trước
                start->prev = NULL; //thiết lập liên kết sau
            }
            else { //trường hợp danh sách không rỗng
                ptr = start; //ptr trỏ đến start
                start = temp; //start lấy bằng temp
                start->next = ptr; //thiết lập liên kết trước cho start
                ptr->prev = start; //thiết lập liên kết sau cho ptr
            }
        }
        else if (pos > 1  && pos <= counter){ //trường hợp vị trí pos hợp lệ
            s = start; //s trỏ đến node đầu tiên
            for (i = 1; i < pos; i++){//di chuyển s đến vị trí pos
                ptr = s;
                s = s->next;
            }
            ptr->next = temp;//thiết lập liên kết trước cho ptr là temp
            temp->prev =ptr; //thiết lập liên kết sau cho temp là ptr
            temp->next = s; //thiết lập liên kết trước cho temp là s
            s->prev = temp; //thiết lập liên kết sau cho s là temp
        }
        else {
            cout<<"Vị trí không hợp lệ"<<endl;
        }
}
//Sắp xếp nội dung các node
void double_linked_list::sort(){//sắp xếp nội dung các node
        struct node *ptr, *s; //sử dụng hai con trỏ node
        int value;
        if (start == NULL){//trường hợp danh sách rỗng
            cout<<"Không có gì để sắp xếp"<<endl;
            return;
        }
        ptr = start;//ptr trỏ đến node đầu tiên
        while (ptr != NULL){ //lặp cho đến node cuối cùng
```

```
            for (s = ptr->next;s !=NULL;s = s->next){ //duyệt từ node tiếp theo
                if (ptr->info > s->info){//nếu xảy ra điều này
                    value = ptr->info;//đổi nội dung hai node
                    ptr->info = s->info;
                    s->info = value;
                }
            }
            ptr = ptr->next; //duyệt đến node tiếp theo
        }
}
```

//Loại node ở vị trí pos:

```
void double_linked_list::delete_pos(){//loại node ở vị trí bất kỳ
        int pos, i, counter = 0;
        if (start == NULL){ //trường hợp danh sách rỗng
            cout<<"Không có gì để loại bỏ"<<endl;
            return;
        }
        cout<<"Vị trí node cần loại:";cin>>pos;
        struct node *s, *ptr; //sử dụng hai con trỏ node
        s = start; //s trỏ đến start
        if (pos == 1){ //nếu loại node đầu tiên
            start = s->next; //di chuyển start lên một node
            free(s); //giải phóng s là xong
        }
        else {//nếu không phải là node đầu tiên
            while (s != NULL) {//xác định counter là số node trong DSLKK
                s = s->next;
                counter++;
            }
            if (pos > 0 && pos <= counter){ //nếu vị trí hợp lệ
                s = start; //s trỏ đến node đầu tiên
                for (i = 1;i < pos;i++){ //di chuyển s đến vị trí pos
                    ptr = s;
                    s = s->next;
                }
                if(s->next==NULL){//nếu s lại là node cuối cùng
                    ptr->next = NULL; //thiết lập liên kết trước cho ptr
```

```
                        s ->prev = NULL; //thiết lập liên kết sau cho s
                        free(s);//giải phóng s
                    }
                    else {
                        ptr->next = s->next; //thiết lập liên kết trước cho ptr
                        (s->next)->prev = ptr; //thiết lập liên kết sau cho s-next
                        free(s); //giải phóng s
                    }
                }
                else {
                    cout<<"Vị trí không hợp lệ"<<endl;
                }
        }
}
```

//Sửa đổi thông tin cho node

```
void double_linked_list::update(){//sửa đổi thông tin cho node
        int value, pos, i;
        if (start == NULL){ //nếu danh sách rỗng
                cout<<"Không có gì để sửa"<<endl;  return;
        }
        cout<<"Vị trí node cần cập nhật:";cin>>pos;
        cout<<"Giá trị mới của node:";cin>>value;
        struct node *s, *ptr; //sử dụng hai con trỏ node
        s = start; //s trỏ đến start
        if (pos == 1){//nếu cập nhật node đầu tiên
                start->info = value; //ta thực hiện được ngay
        }
        else {//trường hợp không phải node đầu tiên
                for (i = 0;i < pos - 1;i++){//chuyển s đến vị trí pos
                        if (s == NULL){//nếu s là node rỗng
                                cout<<"Vị trí "<<pos<<" không hợp lệ";
                                return;
                        }
                        s = s->next;
                }
                s->info = value;  //cập nhật lại thông tin cho s
        }
```

```
        cout<<"Sửa đổi thành công"<<endl;
}
//Tìm kiếm node
void double_linked_list::search(){//tìm kiếm node có giá trị value
        int value, pos = 0;     bool flag = false;
        if (start == NULL){//trường hợp danh sách rỗng
                cout<<"Không có gì để tìm"<<endl;  return;
        }
        cout<<"Giá trị node cần tìm:";cin>>value;
        struct node *s; s = start;//s trỏ đến start
        while (s != NULL){ pos++;
                if (s->info == value){//nếu tìm thấy node s
                        flag = true;
                        cout<<"Giá trị "<<value<<" tại vị trí "<<pos<<endl;
                }
                s = s->next;
        }
        if (!flag)
                cout<<"Giá trị "<<value<<" không tồn tại"<<endl;
}
//Duyệt phải các node
void double_linked_list::right_travel(){//duyệt từ node đầu tiên
        struct node *temp;
        if (start == NULL){
                cout<<"Danh sách rỗng"<<endl;
                return;
        }
        temp = start;
        cout<<"Nội dung các node: "<<endl;
        while (temp != NULL) { //lặp đến node cuối cùng
                cout<<temp->info<<"->"; //đưa ra nội dung node
                temp = temp->next; //di chuyển đến node tiếp theo
        }
        cout<<"NULL"<<endl; //node cuối cùng là null
}
//Duyệt trái từ node cuối cùng
void double_linked_list::left_travel(){//duyệt từ node cuối cùng
```

```
        struct node *temp;
        if (start == NULL){
                cout<<"Danh sách rỗng"<<endl;
                return;
        }
        temp = start;
        while(temp->next!=NULL) //di chuyển đến node cuối cùng
                temp = temp->next;
        cout<<"Nội dung các node: "<<endl;
        while (temp != NULL) {//lặp cho đến node đầu tiên
                cout<<temp->info<<"->";//nội dung node hiện tại
                temp = temp->prev;//lùi lại node phía sau
        }
        cout<<"NULL"<<endl; //node cuối cùng là null
}
int main() {//chương trình chính
        int choice;
        double_linked_list X;//X là đối tượng double_linked_list
        start = NULL; //start được khởi đầu là null

        while (1){
                cout<<endl<<"---------------------------------"<<endl;
                cout<<endl<<"THAO TÁC TRÊN DSLKK "<<endl;
                cout<<endl<<"---------------------------------"<<endl;
                cout<<"1. Thêm node vào đầu danh sách"<<endl;
                cout<<"2. Thêm node vào cuối danh sách "<<endl;
                cout<<"3. Thêm node vào vị trí bất kỳ "<<endl;
                cout<<"4. Sắp xếp nội dung các node"<<endl;
                cout<<"5. Loại bỏ node bất kỳ"<<endl;
                cout<<"6. Cập nhật nội dung node"<<endl;
                cout<<"7. Tìm kiếm node theo giá trị"<<endl;
                cout<<"8. Duyệt từ trái qua phải"<<endl;
                cout<<"9. Duyệt từ phải qua trái "<<endl;
                cout<<"0. Thoát "<<endl;
                cout<<"Lựa chọn chức năng: "; cin>>choice;
                switch(choice){
                        case 1:
```

```
                cout<<"Thêm node vào đầu: "<<endl;
                X.insert_begin();cout<<endl;   break;
            case 2:
                cout<<"Thêm node vào cuối: "<<endl;
                X.insert_last();cout<<endl;  break;
            case 3:
                cout<<"Thêm node vào vị trí bất kỳ:"<<endl;
                X.insert_pos();cout<<endl;  break;
            case 4:
                cout<<"Sắp xếp nội dung các node: "<<endl;
                X.sort();cout<<endl;  break;
            case 5:
                cout<<"Loại bỏ node bất kỳ: "<<endl;
                X.delete_pos();  break;
            case 6:
                cout<<"Cập nhật nội dung node:"<<endl;
                X.update();cout<<endl;  break;
            case 7:
                cout<<"Tìm node theo giá trị: "<<endl;
                X.search();cout<<endl;  break;
            case 8:
                cout<<"Duyệt từ node bên trái:"<<endl;
                X.left_travel();cout<<endl;  break;
            case 9:
                cout<<"Duyệt từ node bên phải:"<<endl;
                X.right_travel();cout<<endl;  break;
            case 0:
                cout<<"Thoat..."<<endl;  exit(1);  break;
            default:
                cout<<"Lua chon sai"<<endl;
        }
    }
}
```

4.2.4. Xây dựng danh sách liên kết kép bằng STL

Các ngôn ngữ lập trình đã xây dựng API cho danh sách liên kết kép. Dưới đây là một cách cài đặt sử dụng STL của C++.

```
#include <iostream>
```

```
#include <list>
using namespace std;
int main(){
	int myints[] = {5, 6, 3, 2, 7};
	list<int> l, l1 (myints, myints + sizeof(myints) / sizeof(int));
	list<int>::iterator it;
	int choice, item;
	do {
		cout<<"\n----------------------"<<endl;
		cout<<"CAI DAT DANH SACH LK KEP BANG STL"<<endl;
		cout<<"\n----------------------"<<endl;
		cout<<"1.Them phan tu vao dau"<<endl;
		cout<<"2.Them phan tu vao cuoi"<<endl;
		cout<<"3.Loai bo phan tu o dau"<<endl;
		cout<<"4.Loai bo phan tu o cuoi"<<endl;
		cout<<"5.Phan tu o dau danh sach"<<endl;
		cout<<"6.Phan tu o cuoi danh sach"<<endl;
		cout<<"7.Kich co cua danh sach"<<endl;
		cout<<"8.Thay doi kich co danh sach"<<endl;
		cout<<"9.Loai bo phan tu co gia tri Values"<<endl;
		cout<<"10.Loai bo phan tu trung lap"<<endl;
		cout<<"11.Dao nguoc cac phan tu"<<endl;
		cout<<"12.Sap xep cac phan tu"<<endl;
		cout<<"13.Hoa nhap cac phan tu"<<endl;
		cout<<"14.Hien thi cac phan tu"<<endl;
		cout<<"0. Thoat"<<endl;
		cout<<"Dua vao lua chon: ";cin>>choice;
		switch(choice){
			case 1:
				cout<<"Node can them vao dau: ";cin>>item;
				l.push_front(item); break;
			case 2:
				cout<<"Node can them vao cuoi: ";cin>>item;
				l.push_back(item);break;
			case 3:
				item = l.front(); l.pop_front();
				cout<<"Node "<<item<<" da bi loai"<<endl;
```

```
            break;
    case 4:
            item = l.back();l.pop_back();
            cout<<"Node "<<item<<" da bi loai"<<endl; break;
    case 5:
            cout<<"Phan tu dau danh sach: ";
            cout<<l.front()<<endl; break;
    case 6:
            cout<<"Phan tu cuoi danh sach: : ";
            cout<<l.back()<<endl; break;
    case 7:
            cout<<"Kich co cua danh sach: "<<l.size()<<endl;
            break;
    case 8:
            cout<<"Kich co moi cua danh sach: ";
            cin>>item;
            if (item <= l.size())
                    l.resize(item);
            else
                    l.resize(item, 0);
            break;
    case 9:
            cout<<"Noi dung node bi loai: ";cin>>item;
            l.remove(item); break;
    case 10:
            l.unique();
            cout<<"Phan tu giong nhau bi loai"<<endl;
            break;
    case 11:
            l.reverse();
            cout<<"Danh sach da duoc dao"<<endl;
            break;
    case 12:
            l.sort();
            cout<<"Danh sach da duoc sap"<<endl;
            break;
    case 13:
```

```
                l1.sort();
                l.sort();
                l.merge(l1);
                cout<<"Cac danh sach da hoa nhap"<<endl;
                break;
            case 14:
                cout<<"Duyet danh sach:  ";
                for (it = l.begin(); it != l.end(); it++)
                    cout<<*it<<"  ";
                cout<<endl;
                break;
            case 0:
                break;
        }
    }while(choice!=0);
}
```

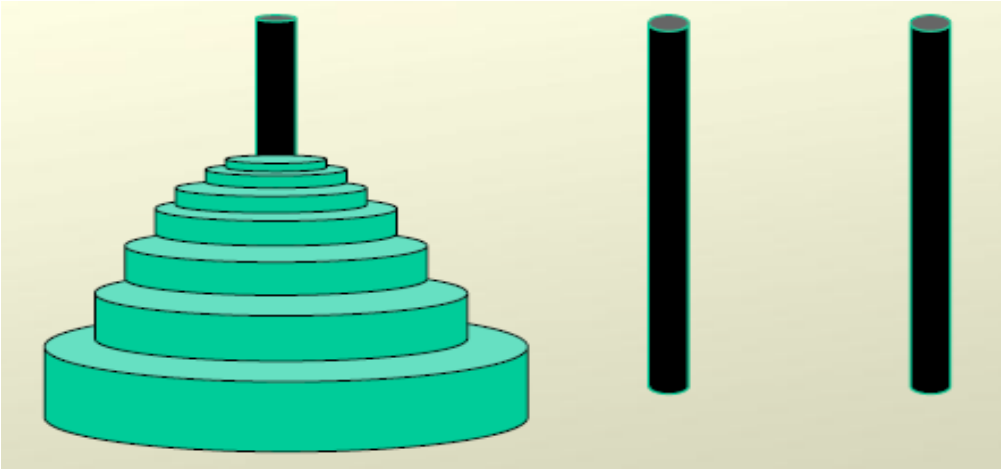
4.3. Ngăn xếp (Stack)

Ngăn xếp được hiểu như một cái ngăn để xếp. Ngăn xếp được ứng dụng trong nhiều lĩnh vực quan trọng khác nhau của khoa học máy tính. Trong mục này, ta sẽ xem xét phương pháp xây dựng kiểu dữ liệu ngăn xếp và ứng dụng của ngăn xếp.

4.3.1. Định nghĩa ngăn xếp

Tập hợp các node thông tin được tổ chức liên tục hoặc rời rạc nhau trong bộ nhớ và thực hiện theo cơ chế vào trước ra sau FILO (First – In – Last – Out).

Ta có thể hình dung stack như chồng đĩa trong bài toán "Tháp Hà Nội". Đĩa có đường kính lớn nhất được xếp trước nhất, đĩa có đường kính nhỏ nhất được xếp sau cùng vào chồng đĩa. Tuy nhiên, khi lấy ra các đĩa có đường kính nhỏ nhất được lấy ra trước nhất, đĩa có đường lớn nhất sẽ được lấy ra sau cùng.



Hình 3.10. Mô tả ngăn xếp

4.3.2. Biểu diễn ngăn xếp

Có hai phương pháp biểu diễn ngăn xếp: biểu diễn liên tục và biểu diễn rời rạc.

- **Biểu diễn liên tục**: các phần tử dữ liệu của ngăn xếp được lưu trữ liên tục nhau trong bộ nhớ.
- **Biểu diễn rời rạc**: các phần tử dữ liệu của ngăn xếp được lưu trữ rời rạc nhau trong bộ nhớ (Danh sách liên kết).

Trong trường hợp biểu diễn liên tục, ta sử dụng mảng để biểu diễn các node dữ liệu của ngăn xếp như sau:

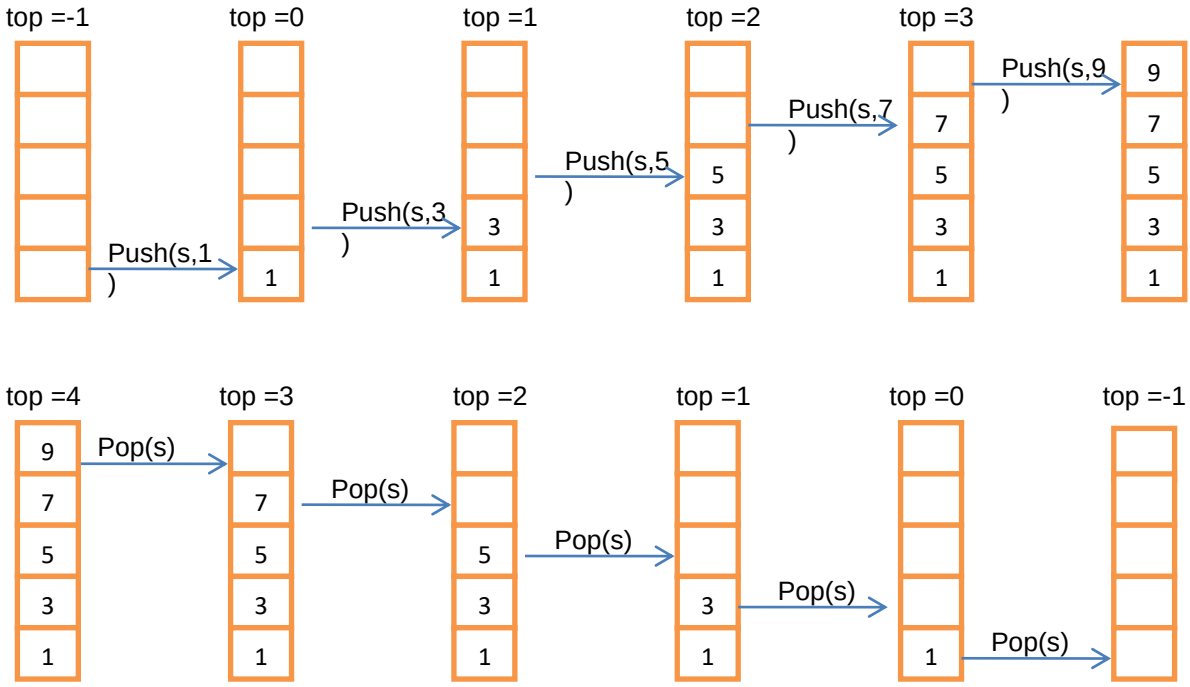
```
typedef struct Stack{
        int     top; //Đỉnh đầu của stack nơi diễn ra mọi thao tác
        int     node[MAX]; //Dữ liệu lưu trữ trong stack gồm MAX phần tử
};
```

Trong trường hợp biểu diễn rời rạc, ta sử dụng danh sách liên kết để biểu diễn rời rạc các node dữ liệu của ngăn xếp như sau:

```
typedef struct node{
        int     infor; //thông tin của node
        struct node *next; //con trỏ trỏ đến node tiếp theo.
};
```

4.3.3. Các thao tác trên ngăn xếp

Có hai thao tác cơ bản để tạo nên cơ chế vào sau ra trước (LIFO) của stack đó là đưa phần tử vào ngăn xếp (push) và lấy phần tử ra khỏi ngăn xếp. Hai thao tác push và pop đều thực hiện chung tại một vị trí trên ngăn xếp sẽ tạo nên cơ chế LIFO. Hình 3.11 dưới đây sẽ mô tả cơ chế LIFO của ngăn xếp.



Hình 3.11. Cơ chế vào trước ra sau của ngăn xếp

Xây dựng ngăn xếp dựa vào mảng: chương trình dưới đây thể hiện phương pháp xây dựng ngăn xếp dựa vào mảng.

```
#include <iostream>
#include <iomanip>
#define MAXSIZE 5
using namespace std;
typedef struct STK{ //biểu diễn stack dùng mảng
        int top; //vị trí diễn ra các thao tác
        int node[MAXSIZE];//các node dữ liệu của ngăn xếp
};


class STACK { //xây dựng lớp thao tác cho stack
    public:
        STK s; //định nghĩa s là một stack gồm MAX node
        void push (){ //lấy phần tử ra khỏi ngăn xếp
            int value;//giá trị của node
            if (s.top == (MAXSIZE - 1)){//nếu top ở vị trí MAX-1
                cout<<"Tràn Stack"<<endl;
                return;//ta không đưa được phần tử vào ngăn xếp
            }
```

```
            else { //nếu stack chưa tràn
                cout<<"Nhập phần tử:";cin>>value;
                s.top = s.top + 1; //dịch chuyển top lên một node
                s.node[s.top] = value; //tại vị trí top ta lưu trữ value
            }
        }
        int pop (){//lấy phần tử ra khỏi ngăn xếp
            int value;
            if (s.top == - 1){//nếu stack rỗng
                cout<<"Stack rỗng"<<endl;
                return (INT_MIN);
            }
            else {//nếu stack không rỗng
                value = s.node[s.top]; //giá trị node cần lấy ra
                cout<<"Phần tử bị loại:"<<s.node[s.top]<<endl;
                s.top = s.top - 1; //giảm s.top đi 1
            }
            return(value);//giá trị node bị lấy ra
        }
        void display (){//duyệt stack
            if (s.top == -1) { //nếu stack rỗng
                cout<<"Ta không có gì để duyệt"<<endl;  return;
            }
            else {
                cout<<"\n Nội dung stack: "<<endl;
                for (int i = s.top; i >= 0; i--)cout<<s.node[i]<<setw(3);
            }
        }
        STACK(void){ //constructor của lớp
            s.top=-1; //thiết lập s.top là -1
        }
};
int main (){ //chương trình chính
    STACK X; //định nghĩa đối tượng X
    int choice;
    cout<<"CÁC THAO TÁC TRÊN STACK\n";
    do {
```

```
            cout<<"----------------------------------------\n";
            cout<<"    1   -->   PUSH          \n";
            cout<<"    2   -->   POP         \n";
            cout<<"    3   -->   DISPLAY          \n";
            cout<<"    0   -->   EXIT        \n";
            cout<<"----------------------------------------\n";
            cout<<"Đưa vào lựa chọn:";cin>>choice;
            switch (choice){
                  case 1: X.push(); break;
                  case 2: X.pop(); break;
                  case 3: X.display(); break;
                  default:
                        cout<<"Lựa chọn sai"; break;
            }
      }while(choice!=0);
}
```

Xây dựng ngăn xếp dựa vào danh sách liên kết: Khi xây dựng ngăn xếp dựa vào danh sách liên kết ta chỉ cần định nghĩa một danh sách liên kết. Trên danh sách liên kết ta chỉ cần trang bị tao tác thêm node vào đầu và loại node ở đầu hoặc thêm node vào cuối và loại bỏ node ở cuối sẽ tạo nên ngăn xếp. Chương trình dưới đây thể hiện phương pháp xây dựng ngăn xếp dựa vào danh sách liên kết.

```
      #include<iostream>
      using namespace std;
      struct node{//biểu diễn stack như một danh sách liên kết đơn
            int info; //thông tin của node
            struct node *link; //thành phần liên kết
      }*Stack; //danh sách liên kết đơn stack
      class stack_list{ //xây dựng lớp stack_list
            public:
                  node *push(node *, int); //đưa phần tử vào ngăn xếp
                  node *pop(node *);//lấy phần tử ra khỏi ngăn xếp
                  void traverse(node *); //duyệt ngăn xếp
                  stack_list(){//constructor của lớp
                        Stack = NULL; //stack đầu tiên thiết lập lá null
                  }
      };
      node *stack_list::push(node *Stack, int item){ //thêm node item vào đầu stack
```

```
        node *tmp; //sử dụng con trỏ tmp
        tmp = new (struct node); //cấp phát miền nhớ cho node tmp
        tmp->info = item; //thiết lập thành phần thông tin
        tmp->link = Stack;//thiết lập liên kết cho tmp
        Stack = tmp; //tmp là node đầu danh sách
        return Stack; //kết quả danh sách mới
}
node *stack_list::pop(node *Stack){//đưa node vào ngăn xếp
        node *tmp;//sử dụng con trỏ temp
        if (Stack == NULL) //nếu danh sách rỗng
                cout<<"Stack rỗng"<<endl;
        else    {//nếu danh sách không rỗng
                tmp = Stack; //tmp trỏ đến node đầu stack
                cout<<"Phần tử bị loại bỏ: "<<tmp->info<<endl;
                Stack = Stack->link; //stack trỏ đến node tiếp theo
                free(tmp);//đây là node bị loại
        }
        return Stack;//trả lại stack mới
}
void stack_list::traverse(node *Stack){//duyệt stack
        node *ptr;  ptr = Stack;
        if (Stack == NULL) //nếu stack rỗng
                cout<<"Stack rỗng"<<endl;
        else    {//nếu stack không rỗng
                cout<<"Nội dung Stack :";
                while (ptr != NULL)    {
                        cout<<ptr->info<<endl;
                        ptr = ptr->link;
                }
        }
}
int main(){
        int choice, item;
        stack_list X;
        do    {
                cout<<"\n----------------------------------------"<<endl;
                cout<<"CÁC THAO TÁC TRÊN STACK"<<endl;
```

```
                cout<<"\n----------------------------------------"<<endl;
                cout<<"1. Đưa phần tử vào stack"<<endl;
                cout<<"2. Lấy phần tử ra khỏi stack"<<endl;
                cout<<"3. Duyệt stack"<<endl;
                cout<<"0. Thoát"<<endl;
                cout<<"Dua vao lua chon: ";cin>>choice;
                switch(choice)      {
                        case 1:
                                cout<<"Nhap gia tri: "; cin>>item;
                                Stack = X.push(Stack, item);break;
                        case 2:
                                Stack = X.pop(Stack);break;
                        case 3:
                                X.traverse(Stack);break;
                        default:
                                cout<<"Lựa chọn sai"<<endl;break;
                }
        } while(choice!=0);
}
```

Xây dựng ngăn xếp dựa vàoSTL:

```
#include <iostream>
#include <stack>
using namespace std;
int main(){
        stack<int> st;
        int choice, item;
        do {
                cout<<"\n---------------------"<<endl;
                cout<<"CAI DAT STECK BANG STL"<<endl;
                cout<<"\n---------------------"<<endl;
                cout<<"1.Them phan tu vao stack"<<endl;
                cout<<"2.Loai phan tu khoi Stack"<<endl;
                cout<<"3.Kich thuoc cua Stack"<<endl;
                cout<<"4.Phan tu dau cua Stack"<<endl;
                cout<<"5.Kiem tra tinh rong cua Stack"<<endl;
                cout<<"0.Thoat"<<endl;
```

```
            cout<<"Dua vao lua chon: ";   cin>>choice;
            switch(choice){
                case 1:
                    cout<<"Phan tu can them vao stack: ";
                    cin>>item;st.push(item);break;
                case 2:
                    if (!st.empty()){
                        item = st.top(); st.pop();
                        cout<<"Phan tu "<<item<<" da bi loai"<<endl;
                    }
                    else cout<<"Stack rong"<<endl;
                    break;
                case 3:
                    cout<<"Kich co stack: ";
                    cout<<st.size()<<endl; break;
                case 4:
                    if(st.empty()){
                        cout<<"Phan tu dau Stack: ";
                        cout<<st.top()<<endl;
                    }
                    else cout<<"Stack rong"<<endl;
                    break;
                case 5:
                    cout<<"Trang thai Stack: "<<st.empty();
                    break;
                case 0: break;
            }
        }while(choice!=0);
}
```

4.3.4. Ứng dụng của ngăn xếp

Ngăn xếp được ứng dụng để giải quyết những vấn đề sau:

- Xây dựng các giải thuật đệ qui: tất cả các giải thuật đệ qui được xây dựng dựa trên cơ chế FIFO của ngăn xếp.
- Khử bỏ các giải thuật đệ qui.
- Biểu diễn tính toán.
- Duyệt cây, duyệt đồ thị…

Ví dụ 3.1. Chuyển đổi biểu thức trung tố về biểu thức hậu tố. Ta vẫn hay làm quen và thực hiện tính toán trên các biểu thức số học trung tố. Ví dụ biểu thức trung tố P = (a+b*c)-(a/b+c). Trong cách viết này các phép toán bao giờ cũng đứng giữa hai toán hạng. Phương pháp tính toán được thực hiện theo thứ tự ưu tiên các phép toán số học. Biểu thức số học hậu tố là phương pháp biểu diễn các phép toán hai ngôi +, -, *, /, ^ đứng sau các toán hạng. Phương pháp biểu diễn được thực hiện theo nguyên tắc như sau:

- Nếu a + b là biểu thức trung tố thì biểu thức hậu tố tương ứng là a b +
- Nếu a - b là biểu thức trung tố thì biểu thức hậu tố tương ứng là a b -
- Nếu a * b là biểu thức trung tố thì biểu thức hậu tố tương ứng là a b *
- Nếu a / b là biểu thức trung tố thì biểu thức hậu tố tương ứng là a b /
- Nếu a ^ b là biểu thức trung tố thì biểu thức hậu tố tương ứng là a b ^ (phép ^ được ký hiệu cho phép lấy lũy thừa)
- Nếu (P) trong đó P là hậu tố thì biểu thức hậu tố tương ứng P.

Biểu thức trung tố	Biểu thức hậu tố tương đương
a + b	ab+
a - b	ab-
a * b	ab*
a / b	ab/
a ^ b	ab^
(P)	P

Ví dụ với biểu thức P = (a+b*c)-(a/b+c) sẽ được biến đổi thành biểu thức hậu tố tương đương như dưới đây:

$$\begin{aligned}(a+b*c)-(a/b+c)&=\\&=(a+bc*)-(ab/+c)\\&=(abc*+)-(ab/c+)\\&=abc*+-ab/c+\\&=abc*+ab/c+-\end{aligned}$$

Đối với biểu thức số học hậu tố, không còn các phép toán ‘(‘ , ‘)’, không còn thứ tự ưu tiên các phép toán. Bài toán đặt ra là cho biểu thức trung tố P hãy chuyển đổi P thành biểu diễn hậu tố tương đương với P. Sau khi đã có biểu thức trung tố P, hãy xây dựng thuật toán tính toán giá trị biểu thức hậu tố P. Thuật toán được xây dựng dựa vào ngăn xếp được thể hiện trong Hình 3.12 như dưới đây.

Thuật toán infix-to-postfix (P):

Bước 1 (Khởi tạo)**:**

stack = ∅; *//stack dùng để lưu trữ các phép toán*

Out = ∅;*// out dùng lưu trữ biểu thức hậu tố*

Bước 2 (Lặp) :

For each x∈P do *// duyệt từ trái qua phải biểu thức trung tố P*

2.1. Nếu x =‘ (‘ : Push(stack, x);

2.2. Nếu x là toán hạng: x⇒Out;

2.3. Nếu x ∈ { +, -, *, / ,^}

y = get(stack); *//lấy phép toán ở đầu ngăn xếp*

a) Nếu priority(x)≥priority(y): Push(stack, x);

b) Nếu priority(x)<priority(y):

y = Po(stack); y⇒Out; Push(stack, x);

c) Nếu stack = ∅: Push(stack, x);

2.4. Nếu x =’)’:

y = Pop(stack);

While (y!=‘(’) do

y⇒Out; y = Pop(stack);

EndWhile;

EndFor;

Bước 3(Hoàn chỉnh biểu thức hậu tố)*:*

While (stack ≠ ∅) do

y = Pop(stack); y⇒Out;

EndWhile;

Bước 4(Trả lại kết quả)*:*

Return(Out).

Hình 3.12. Thuật toán chuyển đổi biểu thức trung tố thành biểu thức hậu tố

Kiểm nghiệm thuật toán với P = (a + b*c) – (a/b + c):

x∈P	Bước	Stack	Out
x =‘(‘	2.1	(	∅
x =a	2.2	(	a
x =+	2.3.a	(+	a
x =b	2.2	(+	a b
x =*	2.3.a	(+ *	a b
x = c	2.2	(+ *	a b c
x =‘)‘	2.3	∅	a b c * +
x =-	2.2.c	-	a b c * +
x =‘(‘	2.1	- (	a b c * +
x =a	2.2	- (	a b c * + a
x =/	2.2.a	- (/	a b c * + a
x =b	2.2	- (/	a b c * + a b
x =+	2.3.b	- (+	a b c * + a b/
x =c	2.2	- (+	a b c * + a b/c
x =‘)‘	2.4	∅	a b c * + a b/c + -
P = a b c * + a b/c + -			

Hình 3.13. Kiểm nghiệm thuật toán

Thuật toán tính toán giá trị biểu thức hậu tố:

Bước 1 (Khởi tạo)**:**

stack = ∅;

Bước 2 (Lặp) :

For each x∈P do

2.1. Nếu x là toán hạng:

Push(stack, x);

*2.2. Nếu x ∈ { +, -, *, / }*

a) TH2 = Pop(stack, x);

b) TH1 = Pop(stack, x);

c) KQ = TH1 ⊗ TH2;

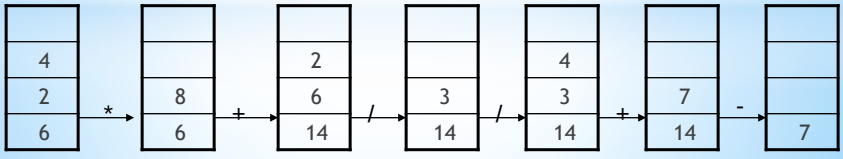
d) Push (stack, KQ);

EndFor;

Bước 4(Trả lại kết quả)*:*

Return(Pop(stack)).

Ví dụ: P = 6 2 4 * + 6 2 / 4 + -



Hình 3.14. Tính toán biểu thức hậu tố

4.4. Hàng đợi (Queue)

Hàng đợi được hiểu là một hàng để đợi. Hàng đợi trong máy tính cũng giống như hàng đợi trong thực tế: hàng đợi mua vé tàu, vé xe, vé máy bay. Hàng đợi ứng dụng trong nhiều lĩnh vực khác nhau của khoa học máy tính. Trong mục này ta sẽ xem xét phương pháp xây dựng hàng đợi cùng với ứng dụng của hàng đợi.

4.4.1. Định nghĩa hàng đợi

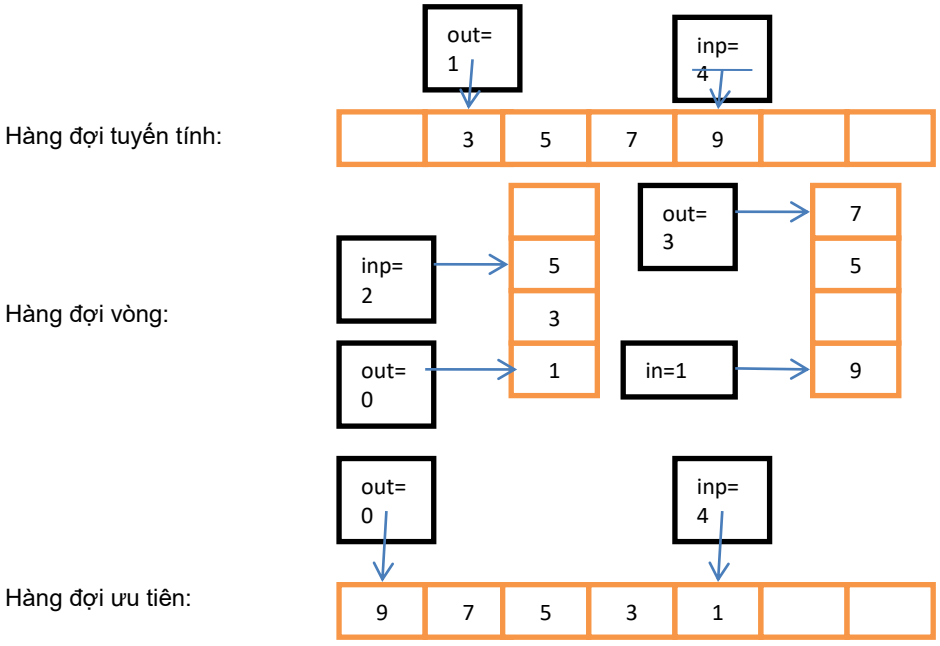
Hàng đợi (queue) là tập các node thông tin được tổ chức liên tục hoặc rời rạc nhau trong bộ nhớ và thực hiện theo cơ chế FIFO (First-In-First-Out).

Hàng đợi tuyến tính (sequntial queue): hàng đợi liên tục được xây dựng theo nguyên tắc có điểm vào (inp) luôn lớn hơn điểm ra (out). Không gian nhớ sẽ không được tái sử dụng sau mỗi phép lấy phần tử ra khỏi hàng đợi.

Hàng đợi vòng (curcular queue): hàng đợi liên tục được xây dựng theo nguyên tắc không gian nhớ sẽ được tái sử dụng sau mỗi phép lấy phần tử ra khỏi hàng đợi.

Hàng đợi ưu tiên (priority queue): hàng đợi được xây dựng theo nguyên tắc phép đưa phần tử vào hàng đợi được xếp ứng với thứ tự ưu tiên của nó.

Hàng đợi hai điểm cuối (double ended queue): hàng đợi được xây dựng theo nguyên tắc phép đưa phần tử vào và lấy phần tử ra khỏi hàng đợi được thực hiện ở hai điểm cuối.



Hình 3.15. Các loại hàng đợi

4.4.2. Biểu diễn hàng đợi

Có hai phương pháp biểu diễn hàng đợi: biểu diễn liên tục và biểu diễn rời rạc. Trong trường hợp biểu diễn liên tục ta sử dụng mảng, trong trường hợp biểu diễn rời rạc ta sử dụng danh sách liên kết.

Biểu diễn liên tục:

```
typedef struct {
        int     inp; // dùng để đưa phần tử vào hàng đợi
        int     out; // dùng để lấy phần tử ra khỏi hàng đợi
        int     node[MAX]; //các node thông tin của hàng đợi
} queue;
```

Biểu diễn rời rạc:

```
struct node { //định nghĩa cấu trúc node
        int infor;//thành phần dữ liệu
        struct node *next; //thành phần liên kết
} *queue;
```

4.4.3. Thao tác trên hàng đợi

Hàng đợi được xây dựng dựa vào hai thao tác cơ bản: đưa phần tử vào hàng đợi (push) và lấy phần tử ra khỏi hàng đợi (pop). Hai thao tác push và pop phối hợp với nhau để tạo nên cơ chế FIFO của hàng đợi.

Cài đặt hàng đợi tuyến tính dựa vào mảng:

```
#include <iostream>
#include <iomanip>
#define MAX 100
using namespace std;
class Queue { //định nghĩa lớp queue
  public:
        int node[MAX]; //các node của queue
        int inp; //dùng để đưa phần tử vào hàng đợi
        int out; //dùng để lấy phần tử ra khỏi hàng đợi
        void Push(){ //đưa phần tử vào hàng đợi
                int value;//giá trị node
                if (inp == MAX - 1) //nếu hàng đợi tràn
                   cout<<"Tràn hàng đợi "<<endl;
                else { //nếu hàng đợi chưa đầy
                        if (out == - 1) //trường hợp hàng đợi rỗng
```

```
                out = 0; //lấy vị trí out = 0
                cout<<"Giá trị node : ";cin>>value;
                inp = inp + 1; //tăng con trỏ inp
                node[inp] = value;//lưu trữ nội dung node
            }
        }
        void Pop(){//lấy phần tử ra khỏi hàng đợi
            if (out == - 1 || out > inp){ //trường hợp hàng đợi rỗng
                cout<<"Queue rỗng"<<endl;
                return ;
            }
            else{//trường hợp hàng đợi không rỗng
                cout<<" Node được lấy ra : "<<node[out]<<endl;
                out = out + 1;
            }
        }
        void Display(){//duyệt các node trong hàng đợi
            if (out == - 1)//trường hợp hàng đợi rỗng
                cout<<"Queue is empty "<<endl;
            else{
                cout<<"Nội dung hàng đợi : ";
                for (int i = out; i <= inp; i++)
                    cout<<node[i]<<setw(3);
                cout<<endl;
            }
        }
        Queue (void) { //constructor của lớp
            inp = - 1;//điểm vào thiết lập -1
            out = - 1; //điểm ra thiết lập -1
        }
};
int main(void) {
        int choice; Queue X;//định nghĩa đối tượng X là queue
        do {
            cout<<"1. Đưa phần tử vào hàng đợi"<<endl;
            cout<<"2. Lấy phần tử ra khỏi hàng đợi"<<endl;
            cout<<"3. Duyệt các node của hàng đợi "<<endl;
```

```
                cout<<"0. Thoát "<<endl;
                cout<<"Đưa vào lựa chọn : ";  cin>>choice;
                switch (choice){
                        case 1: X.Push(); break;
                        case 2: X.Pop(); break;
                        case 3: X.Display();break;
                        default:
                                cout<<"Lựa chọn sai"<<endl; break;
                }
        }while(choice!=0);
}
```

Cài đặt hàng đợi dựa vào danh sách liên kết: ta chỉ cần xây dựng một danh sách liên kết, sau đó trang bị hai thao tác thêm node vào một đầu và loại bỏ node ở đầu còn lại. Dưới đây là phương pháp xây dựng hàng đợi dựa vào danh sách liên kết.

```
#include<iostream>
#include<iomanip>
using namespace std;
typedef struct node{//định nghĩa một node
        int data; // thành phần dữ liệu
        node *next;//thành phần liên kết
};
class Queue {//xây dựng lớp queue
public:
        node *start; //đây là con trỏ đến node đầu tiên
        node *end ;// /đây là con trỏ đến node cuối cùng
        node *np ;// /sử dụng một con trỏ trung gian
        void Push(void){//đưa phần tử vào hàng đợi
                int value; //giá trị node cần thêm
                cout<<"Giá trị node:"; cin>>value;
                np = new node;//cấp phát miền nhớ cho node
                np->data = value;//thiết lập dữ liệu cho node
                np->next = NULL;//thiết lập liên kết cho node
                if(start == NULL){//nếu danh sách rỗng
                        start = end = np;//node đầu và node cuối là một
                        end->next = NULL;//cuối cùng là NULL
                }
                else {//trường hợp danh sách rỗng
```

```
                end->next = np;//node cuối cùng là np
                end = np;//đây là node cuối mới
                end->next = NULL;//thiết lập liên kết cho end
            }
        }
        int Pop(){//lấy phần tử ra khỏi hàng đợi
            int x;
            if(start == NULL){//nếu danh sách rỗng
                cout<<"Hàng đợi rỗng"<<endl;
            }
            else {//trường hợp hàng đợi không rỗng
                np = start;//np trỏ đến node đầu danh sách
                x = np->data;//đây là nội dung node loại bỏ
                start = start->next;//dịch chuyển lên một node
                delete(np);//giải phóng node đầu tiên
                return(x);
            }
        }
        void Display(void){//duyệt hàng đợi
            np = start;
            if(np==NULL)
                cout<<"Hàng đợi rỗng"<<endl;
            else {
                cout<<"Nội dung hàng đợi:";
                while(np!=NULL){
                    cout<<np->data<<setw(3);
                    np=np->next;
                }
                cout<<endl;
            }
        }
        Queue(void){//constructor của lớp
            start = NULL;
            end = NULL;
            np = NULL;
        }
};
```

```
int main(void) {
	int choice; Queue X;//X là đối tượng Queue
	do {
		cout<<"1.Thêm phần tử "<<endl;
		cout<<"2.Loại phần tử "<<endl;
		cout<<"3.Duyệt hàng đợi"<<endl;
		cout<<"0.Thoát "<<endl;
		cout<<"Đưa vào lựa chọn : ";
		cin>>choice;
		switch (choice){
			case 1: X.Push(); break;
			case 2: X.Pop(); break;
			case 3: X.Display();break;
			default: cout<<"Lựa chọn sai "<<endl; break;
		} /*End of switch*/
	}while(choice!=0); /*End of while*/
} /*End of main()*/
```

Cài đặt hàng đợi dựa vào STL:

```
#include <iostream>
#include <queue>
using namespace std;
int main(){
	queue<int> q;
	int choice, item;
	do {
		cout<<"\n----------------------"<<endl;
		cout<<"CAI DAT HANG DOI BANG STL"<<endl;
		cout<<"\n----------------------"<<endl;
		cout<<"1.Them phan tu vao Queue"<<endl;
		cout<<"2.Loai phan tu khoi Queue"<<endl;
		cout<<"3.Kich thuoc cua Queue"<<endl;
		cout<<"4.Phan tu dau tien cua Queue"<<endl;
		cout<<"5.Phan tu cuoi cung cua Queue"<<endl;
		cout<<"6.Trang thai cua Queue"<<endl;
		cout<<"0.Thoat"<<endl;
		cout<<"Dua vao lua chon: ";cin>>choice;
```

```
            switch(choice){
                case 1:
                    cout<<"Phan tu can them: ";
                    cin>>item;q.push(item); break;
                case 2:
                    if(!q.empty()){
                        item = q.front();q.pop();
                        cout<<"Phan tu "<<item<<" da bi loai"<<endl;
                    }
                    else cout<<"Queue rong"<<endl;
                    break;
                case 3:
                    cout<<"Kich co cua Queue: "<<q.size()<<endl;
                    break;
                case 4:
                    if(!q.empty()){
                        cout<<"Phan tu dau hang doi: "<<q.front()<<endl;
                    }
                    break;
                case 5:
                    if(!q.empty())
                        cout<<"Phan tu cuoi Queue:"<<q.back()<<endl;
                    break;
                case 6:
                    cout<<"Trang thai Queue:"<<q.empty()<<endl;
                    break;
                case 0:  break;
            }
        }while(choice!=0);
}
```

Cài đặt hàng đợi ưu tiên dựa vào danh sách liên kết:

```
#include <iostream>
#include <iomanip>
using namespace std;
struct node{//biểu diễn node của hàng đợi ưu tiên
    int priority;//mức độ ưu tiên của node
```

```
        int data;//thành phần thông tin của node
        struct node *next;//thành phần liên kết của node
};
class Priority_Queue{//định nghĩa lớp Priority_Queue
        private:node *start;//start là danh sách liên kết đơn
        public:
                Priority_Queue(){//Constructor của lớp
                        start = NULL;//thiết lập lúc đầu là rỗng
                }
                void Push(int item, int priority);//thêm node vào hàng đợi ưu tiên
                void Pop(void) ; //loại node khỏi hàng đợi ưu tiên
                void Display();//duyệt các node
};
void Priority_Queue::Push(int item, int priority) {//thêm node với độ ưu tiên
        node *tmp, *q;
        tmp = new node;//cấp phát miền nhớ cho node
        tmp->data = item;//thiết lập thông tin của node
        tmp->priority = priority;//thiết lập chế độ ưu tiên của node
        if (start == NULL || priority < start->priority){//trường hợp danh sách rỗng
                tmp->next = start;//tmp chỉ việc đặt ở đầu
                start = tmp;//start bắt đầu từ đây
        }
        else{//nếu start không rỗng
                q = start;// q trỏ đến node đầu tiên
                //tìm vị trí phù hợp với độ ưu tiên của tmp
                while (q->next != NULL && q->next->priority <= priority)
                        q=q->next;
                tmp->next = q->next;//thiết lập liên kết cho tmp
                q->next = tmp;//thiết lập liên kết cho q
        }
}
void Priority_Queue::Pop(){
        node *tmp;
        if(start == NULL)//nếu hàng đợi rỗng
                cout<<"Hàng đợi rỗng”<<endl;
        else{
                tmp = start; //tmp trỏ đến starrt
```

```
            cout<<"Node bị loại: "<<tmp->data<<endl;
            start = start->next; //di chuyển start lên một node
            free(tmp);//giải phóng tmp
        }
}
void Priority_Queue::Display(){//Hien thi priority queue
        node *ptr;
        ptr = start; //ptr trỏ đến start
        if (start == NULL)
            cout<<"Hàng đợi rỗng"<<endl;
        else{
            cout<<"Nội dung hàng đợi"<<endl;
            cout<<"Priority      Item "<<endl;
            while(ptr != NULL){
                cout<<ptr->priority<<setw(10)<<ptr->data<<endl;
                ptr = ptr->next;
            }
        }
}
int main(){
        int choice, item, priority;
        Priority_Queue pq;
        do  {
            cout<<"1.Push\n";
            cout<<"2.Delete\n";
            cout<<"3.Display\n";
            cout<<"0.Quit\n";
            cout<<"Enter your choice : ";  cin>>choice;
            switch(choice) {
                case 1:
                    cout<<"Giá trị node : ";  cin>>item;
                    cout<<"Mức độ ưu tiên : ";   cin>>priority;
                    pq.Push(item, priority);   break;
                case 2:
                    pq.Pop();  break;
                case 3:
                    pq.Display();  break;
```

```
                default :
                    cout<<"Lựa chọn sai"<<endl;break;
            }
        } while(choice != 0);
    }
```

Cài đặt hàng đợi ưu tiên bằng STL:

```
#include <iostream>
#include <queue>
using namespace std;
int main(){
    priority_queue<int> pq;
    int choice, item;
    do{
        cout<<"\n----------------------"<<endl;
        cout<<"XAY DUNG QUEUE PRIORUTY BANG STL"<<endl;
        cout<<"\n----------------------"<<endl;
        cout<<"1.Them phan tu vao hang doi uu tien"<<endl;
        cout<<"2.Loai phan tu khoi hang doi uu tien"<<endl;
        cout<<"3.Kich co hang doi uu tien"<<endl;
        cout<<"4.Phan tu dau hang doi uu tien"<<endl;
        cout<<"5.Trang thai hang doi uu tien"<<endl;
        cout<<"0.Thoat"<<endl;
        cout<<"Dua vao lua chon: ";cin>>choice;
        switch(choice){
            case 1:
                cout<<"Phan tu can them: ";cin>>item;
                pq.push(item); break;
            case 2:
                if (!pq.empty()){
                    item = pq.top();pq.pop();
                    cout<<"Phan tu "<<item<<" da bi loai"<<endl;
                }
                else{
                    cout<<"Hang doi uu tien rong"<<endl;
                }
                break;
```

```
                case 3:
                        cout<<"Kich co hang doi uu tien: "<<pq.size()<<endl;
                        break;
                case 4:
                        if(!pq.empry())
                                cout<<"Phan tu dau tien : "<<pq.top()<<endl;
                        break;
                case 5:
                        cout<<"Trang thai hang doi: "<<pq.empty()<<endl;
                        break;
                case 0:
                        break;
        }
    }while(choice!=0);
}
```

4.4.4. Ứng dụng của hàng đợi

Hàng đợi được sử dụng trong các ứng dụng sau:

- Dùng để xây dựng các hệ thống lập lịch.
- Dùng để xây dựng các thuật toán duyệt cây.
- Dùng để xây dựng các thuật toán duyệt đồ thị.
- Dùng trong việc biểu diễn tính toán.

Ví dụ 3.2. **Bài toán n-ropes**. Cho n dây với chiều dài khác nhau. Ta cần phải nối các dây lại với nhau thành một dây. Chi phí nối hai dây lại với nhau được tính bằng tổng độ dài hai dây. Nhiệm vụ của bài toán là tìm cách nối các dây lại với nhau thành một dây sao cho chi phí nối các dây lại với nhau là ít nhất.

Input:

- Số lượng dây: 4
- Độ dài dây L[]= { 4, 3, 2, 6}

Output: Chi phí nối dây nhỏ nhất.

OPT = 29

Chi phí nhỏ nhất được thực hiện như sau: lấy dây số 3 nối với dây số 2 để được tập 3 dây với độ dài 4, 5, 6. Lấy dây độ dài 4 nối với dây độ dài 5 ta nhận được tập 2 dây với độ dài 6, 9. Cuối cùng nối hai dây còn lại ta nhận được tập một dây với chi phí là 6+9 =15. Như vậy, tổng chi phí nhỏ nhất của ba lần nối dây là 5 + 9 + 15 = 29.

Ta không thể có cách nối dây khác với chi phí nhỏ hơn 29. Ví dụ lấy dây 1 nối dây 2 ta nhận được 3 dây với độ dài {7, 2, 6}. Lấy dây 3 nối dây 4 ta nhận được tập hai dây

với độ dài {7, 8}, nối hai dây cuối cùng ta nhận được 1 dây với độ dài 15. Tuy vậy, tổng chi phí là 7 + 8 + 15 = 30.

Lời giải. Sử dụng thuật toán tham lam dựa vào hàng đợi ưu tiên như trong Hình 3.16.

Thuật toán. *Sử dụng phương pháp tham lam dựa vào hàng đợi ưu tiên*.
Thuật toán Greedy-N-Ropes(int L[], int n):
Input:
- n : số lượng dây.
- L[] : chi phí nối dây.

Output:
- Chi phí nối dây nhỏ nhất.

Actions:

Bước 1 . Tạo pq là hàng đợi ưu tiên lưu trữ độ dài n dây.
Bước 2 (Lặp).

```
OPT = 0; // Chi phí nhỏ nhất.
While (pq.size>1) {
        First = pq.top; pq.pop(); //Lấy và loại phần tử đầu tiên trong pq.
        Second = pq.top; pq.pop(); //Lấy và loại phần tử kế tiếp trong pq.
        OPT = First + Second; //Giá trị nhỏ nhất để nối hai dây
        Pq.push(First + Second); //Đưa lại giá trị First + Second vào pq.
}
```

Bước 3(Trả lại kết quả).

```
Return(OPT);
```

EndActions.

Hình 3.16. Thuật toán tham giải bài toán n-ropes

Kiểm nghiệm Thuật toán Greedy-N-Ropes(int L[], int n):
Input:
- Số lượng dây n = 8.
- Chi phí nối dây L[] = { 9, 7, 12, 8, 6, 5, 14, 4}.

Output:
- Chi phí nối dây nhỏ nhất.

Phương pháp tiến hành::

Bước	Giá trị First, Second	OPT=?	Trạng thái hàng đợi ưu tiên.
		0	4, 5, 6, 7, 8, 9, 12, 14
1	First=4; Second=5	9	6, 7, 8, 9, 9, 12, 14
2	First=6; Second=7	22	8, 9, 9, 12, 13, 14
3	First=8; Second=9	39	9, 12, 13, 14, 17
4	First=9; Second=12	60	13, 14, 17, 21
5	First=13; Second=14	87	17, 21, 27
6	First=17; Second=21	125	27, 38
7	First=27; Second=38	190	65
OPT = 190			

Hình 3.17. Kiểm nghiệm thuật toán tham giải bài toán n-rope

BÀI TẬP

BÀI 1. TỔNG ĐA THỨC

Cho hai đa thức có bậc không quá 10000 (chỉ viết ra các phần tử có hệ số khác 0). Hãy sử dụng danh sách liên kết đơn để viết chương trình tính tổng hai đa thức đó.

Dữ liệu vào: Dòng đầu ghi số bộ test. Mỗi bộ test có hai dòng, mỗi dòng ghi một đa thức theo mẫu như trong ví dụ. Số phần tử của đa thức không quá 20.

Chú ý: Bậc của các hạng tử luôn theo thứ tự giảm dần, trong đa thức chỉ có phép cộng và luôn được viết đầy đủ hệ số + số mũ (kể cả mũ 0).

Kết quả: Ghi ra một dòng đa thức tổng tính được (theo mẫu như ví dụ)

Ví dụ:

Input	Output
1 3*x^8 + 7*x^2 + 4*x^0 11*x^6 + 9*x^2 + 2*x^1 + 3*x^0	3*x^8 + 11*x^6 + 16*x^2 + 2*x^1 + 7*x^0

BÀI 2. ĐẢO TỪ

Cho một xâu ký tự str bao gồm nhiều từ trong xâu. Hãy đảo ngược từng từ trong xâu?

Input:

- Dòng đầu tiên đưa vào số lượng bộ test T;
- Những dòng tiếp theo mỗi dòng đưa vào một bộ test. Mỗi bộ test là một dòng ghi lại nhiều từ trong xâu str.

Output:

- Đưa ra kết quả mỗi test theo từng dòng.

Ràng buộc:

- T, str thỏa mãn ràng buộc: $1 \le T \le 100$; $2 \le length(str) \le 10^6$.

Ví dụ:

Input	Output
2 ABC DEF 123 456	CBA FED 321 654

BÀI 3. KIỂM TRA DÃY NGOẶC ĐÚNG

Cho một xâu chỉ gồm các kí tự '(', ')', '[', ']', '{', '}'. Một dãy ngoặc đúng được định nghĩa như sau:

- Xâu rỗng là 1 dãy ngoặc đúng.
- Nếu A là 1 dãy ngoặc đúng thì (A), [A], {A} là 1 dãy ngoặc đúng.
- Nếu A và B là 2 dãy ngoặc đúng thì AB là 1 dãy ngoặc đúng.

Cho một xâu S. Nhiệm vụ của bạn là xác định xâu S có là dãy ngoặc đúng hay không?

Input:

Dòng đầu tiên là số lượng bộ test T ($T \le 20$).

Mỗi test gồm 1 xâu S có độ dài không vượt quá 100 000.

Output:

Với mỗi test, in ra "YES" nếu như S là dãy ngoặc đúng, in ra "NO" trong trường hợp ngược lại.

Ví dụ:

Input:	**Output**
2 [()]{}{[()()]()} [(])	YES NO

BÀI 4. BIẾN ĐỔI TRUNG TỐ - HẬU TỐ

Hãy viết chương trình chuyển đổi biểu thức biểu diễn dưới dạng trung tố về dạng hậu tố.

Input:

- Dòng đầu tiên đưa vào số lượng bộ test T;
- Những dòng tiếp theo mỗi dòng đưa vào một bộ test. Mỗi bộ test là một biểu thức tiền tố exp.

Output:

- Đưa ra kết quả mỗi test theo từng dòng.

Ràng buộc:

- T, exp thỏa mãn ràng buộc: 1≤T≤100; 2≤length(exp)≤10.

Ví dụ:

Input	Output
2 (A+(B+C) ((A*B)+C)	ABC++ AB*C+

BÀI 5. TÍNH GIÁ TRỊ BIỂU THỨC HẬU TỐ

Hãy viết chương trình chuyển tính toán giá trị của biểu thức hậu tố.

Input:

- Dòng đầu tiên đưa vào số lượng bộ test T;
- Những dòng tiếp theo mỗi dòng đưa vào một bộ test. Mỗi bộ test là một biểu thức hậu tố exp. Các số xuất hiện trong biểu thức là các số đơn có 1 chữ số.

Output:

- Đưa ra kết quả mỗi test theo từng dòng, chỉ lấy giá trị phần nguyên.

Ràng buộc:

- T, exp thỏa mãn ràng buộc: 1≤T≤100; 2≤length(exp)≤20.

Ví dụ:

Input	**Output**
2 231*+9- 875*+9-	-4 34

BÀI 6. PHẦN TỬ BÊN PHẢI ĐẦU TIÊN LỚN HƠN

Cho dãy số A[] gồm N phần tử. Với mỗi A[i], bạn cần tìm phần tử bên phải đầu tiên lớn hơn nó. Nếu không tồn tại, in ra -1.

Input:

Dòng đầu tiên là số lượng bộ test T ($T \le 20$).
Mỗi test bắt đầu bởi số nguyên N ($1 \le N \le 100000$).
Dòng tiếp theo gồm N số nguyên A[i] ($0 \le A[i] \le 10^9$).

Output:

Với mỗi test, in ra trên một dòng N số R[i], với R[i] là giá trị phần tử đầu tiên lớn hơn A[i].

Ví dụ

Input	Output
3 4 4 5 2 25 3 2 2 2 4 4 4 5 5	5 25 25 -1 -1 -1 -1 5 5 -1 -1

BÀI 7. SỐ NHỊ PHÂN TỪ 1 ĐẾN N

Cho số tự nhiên n. Hãy in ra tất cả các số nhị phân từ 1 đến n.

Input:

- Dòng đầu tiên ghi lại số lượng test T (T≤100).
- Mỗi test là một số tự nhiên n được ghi trên một dòng (n≤10000).

Output:

- Đưa ra kết quả mỗi test trên một dòng.

Ví dụ:

Input	Output
2 2 5	1 10 1 10 11 100 101

BÀI 8. BIẾN ĐỔI S – T

Cho hai số nguyên dương S và T (S, T<10000) và hai thao tác (a), (b) dưới đây:

Thao tác (a): Trừ S đi 1 (S = S-1) ;

Thao tác (b): Nhân S với 2 (S = S*2);

Hãy dịch chuyển S thành T sao cho số lần thực hiện các thao tác (a), (b) là ít nhất. Ví dụ với S =2, T=5 thì số các bước ít nhất để dịch chuyển S thành T thông qua 4 thao tác sau:

Thao tác (a): 2*2 = 4;

Thao tác (b): 4-1 = 3;

Thao tác (a): 3*2 = 6;

Thao tác (b): 6-1 = 5;

Input:

- Dòng đầu tiên ghi lại số tự nhiên T là số lượng Test;
- T dòng kế tiếp mỗi dòng ghi lại một bộ Test. Mỗi test là một bộ đôi S và T.

Output: Đưa ra kết quả mỗi test theo từng dòng.

Ví dụ:

Input	Output
3 2 5 3 7 7 4	4 4 3

BÀI 9. BIẾN ĐỔI SỐ NGUYÊN TỐ

Cho cặp số S và T là các số nguyên tố có 4 chữ số (Ví dụ S = 1033, T = 8197 là các số nguyên tố có 4 chữ số). Hãy viết chương trình tìm cách dịch chuyển S thành T thỏa mãn đồng thời những điều kiện dưới đây:

a) Mỗi phép dịch chuyển chỉ được phép thay đổi một chữ số của số ở bước trước đó (ví dụ nếu S=1033 thì phép dịch chuyển S thành 1733 là hợp lệ);

b) Số nhận được cũng là một số nguyên tố có 4 chữ số (ví dụ nếu S=1033 thì phép dịch chuyển S thành 1833 là không hợp lệ, và S dịch chuyển thành 1733 là hợp lệ);

c) Số các bước dịch chuyển là ít nhất.

Ví dụ số các phép dịch chuyển ít nhất để S = 1033 thành T = 8179 là 6 bao gồm các phép dịch chuyển như sau:

8179 ← 8779← 3779← 3739← 3733← 1733← 1033.

Input:

- Dòng đầu tiên đưa vào số lượng test T (T≤100)
- Những dòng kế tiếp mỗi dòng đưa vào một test. Mỗi test là một bộ đôi S, T.

Output:

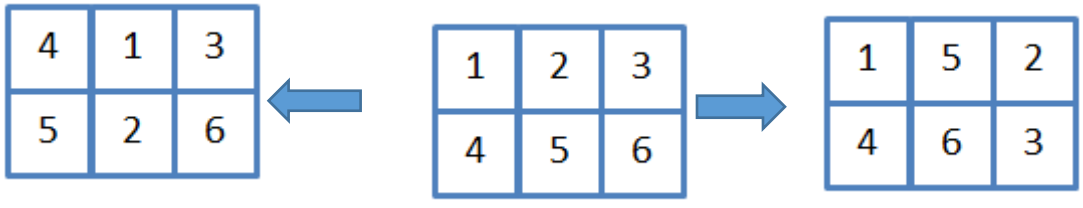
- Đưa ra kết quả mỗi test theo từng dòng.

Ví dụ:

Input	Output
2 1033 8179 1033 8779	6 5

BÀI 10. QUAY HÌNH VUÔNG

Có một chiếc bảng hình chữ nhật với 6 miếng ghép, trên mỗi miếng ghép được điền một số nguyên trong khoảng từ 1 đến 6. Tại mỗi bước, chọn một hình vuông (bên trái hoặc bên phải), rồi quay theo chiều kim đồng hồ.



Yêu cầu: Cho một trạng thái của bảng, hãy tính số phép biến đổi ít nhất để đưa bảng đến trạng thái đích.

Input:

- Dòng đầu ghi số bộ test (không quá 10). Mỗi bộ test gồm hai dòng:
    - Dòng đầu tiên chứa 6 số là trạng thái bảng ban đầu (thứ tự từ trái qua phải, dòng 1 tới dòng 2).
    - Dòng thứ hai chứa 6 số là trạng thái bảng đích (thứ tự từ trái qua phải, dòng 1 tới dòng 2).

Output:

- Với mỗi test, in ra một số nguyên là đáp số của bài toán.

Ví dụ:

Input	Output
1 1 2 3 4 5 6 4 1 2 6 5 3	2

CHƯƠNG 5. CÂY NHỊ PHÂN (BINARY TREE)

Như đã được đề cập ở trên, hạn chế lớn nhất của mảng là luôn đòi hỏi một không gian nhớ liên tục. Điều này sẽ gặp phải khó khăn khi xử lý các đối tượng dữ liệu lớn. Hàng đợi không đòi hỏi một không gian nhớ liên tục nhưng gặp phải vấn đề trong tìm kiếm. Trong chương này ta sẽ xem xét một cấu trúc dữ liệu rời rạc đó là cây nhị phân. Các node trên cây được tổ chức và truy cập không theo thứ tự. Cây nhị phân cho phép ta tổ chức và xử lý được các ứng dụng có dữ liệu rất lớn. Tìm kiếm trên cây nhị phân không nhanh như tìm kiếm trên mảng nhưng tốt hơn rất nhiều so với danh sách liên kết. Nội dung đề cập của chương bao gồm:

- Định nghĩa và khái niệm.
- Biểu diễn cây nhị phân.
- Các thao tác trên cây nhị phân.
- Ứng dụng của cây nhị phân.
- Cây nhị phân tìm kiếm.
- Cây nhị phân tìm kiếm cân bằng.
- Cây nhiều nhánh.

5.1. Định nghĩa và khái niệm

Đối với cấu trúc dữ liệu cây ta có hai phương pháp tiếp cận: tiếp cận bằng cây nhị phân và tiếp cận cây bằng lý thuyết đồ thị. Cây nhị phân được xem là cây đơn giản nhất trong cấu trúc cây. Tuy nhiên, một kết quả quan trọng trong khi nghiên cứu về cây nhị phân là mọi cây tổng quát đều có thể dịch chuyển về một cây nhị phân tương đương. Điều này có nghĩa mọi kết quả phát biểu trên cây nhị phân cũng đúng cho cây tổng quát. Dưới đây là một số khái niệm trên cây nhị phân.

4.1.1. Định nghĩa

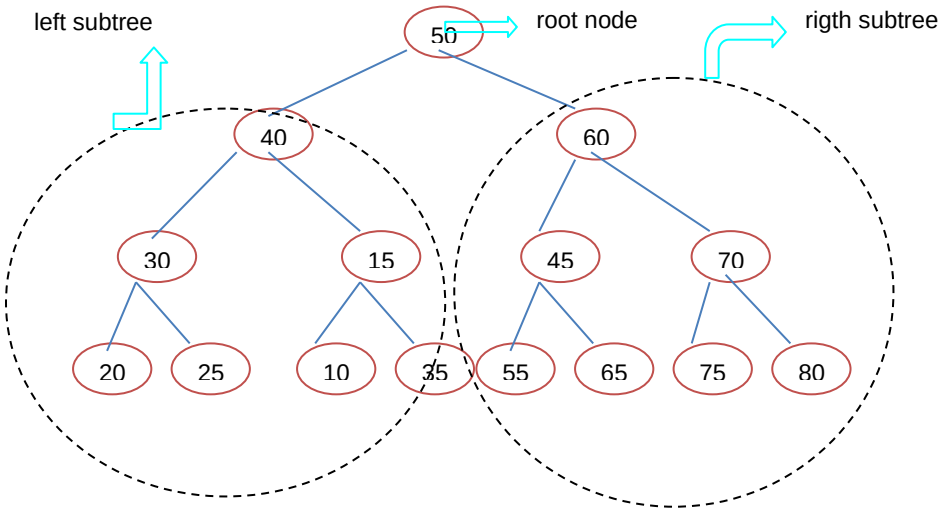
Tập hợp hữu hạn các node có cùng kiểu dữ liệu (có thể là tập ∅) được phân thành 3 tập:

- Tập thứ nhất có thể là ∅ hoặc chỉ có một node gọi là node gốc (root).
- Hai tập con còn lại tự hình thành hai cây con bên trái (left subtree) và cây con bên phải (right subtree) của node gốc (hai tập con này cũng có thể là tập ∅).

Một số khái niệm trên cây:

- **Node gốc (Root)** là node đầu tiên định hình cây.

- **Node cha** (Father): node A là node cha của node B nếu B hoặc là node con bên trái của node A (left son) hoặc B là node con bên phải của node B (right son).
- **Node lá (Leaf):** node không có node con trái, không có node con phải.
- **Node trung gian (Internal Node):** node hoặc có node con trái, hoặc có node con phải, hoặc cả hai hoặc cả hai.
- **Node trước (Ancestor)**: node A gọi là node trước của node B nếu cây con node gốc là A chứa node B.
- **Node sau trái (left descendent)**: node B là node sau bên trái của node A nếu cây con bên trái của node A chứa node B.
- **Node sau phải (right descendent):** node B là node sau bên phải của node A nếu cây con bên phải của node A chứa node B.
- **Node anh em (brother)**: A và B là anh em nếu cả A và B là node con trái và node con phải của cùng một node cha.
- **Bậc của node (degree of node)**: Số cây con tối đa của node.
- **Mức của node (level of node)**: mức node gốc có bậc là 1, mức của các node khác trên cây bằng mức của node cha cộng thêm 1.
- **Chiều sâu của cây (depth of tree)**: mức lớn nhất của node lá trong cây. Như vậy, độ sâu của cây bằng đúng độ dài đường đi dài nhất từ node gốc đến node lá.



Hình 4.1. Cây nhị phân.

5.1.2. Một số tính chất của cây nhị phân

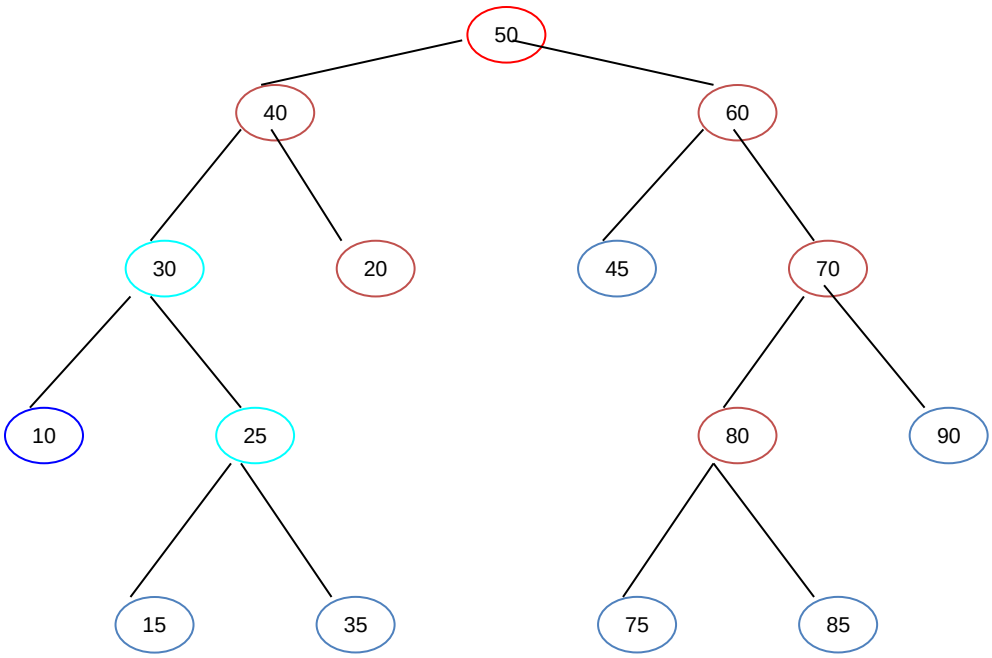
Cây nhị phân có một số tính chất dưới đây:

- Số lượng lớn nhất các node ở mức h là 2^{h-1}. Ví dụ, số lượng mức của node gốc là $2^{1-1} = 1$, mức 2 là $2^{2-1} = 2$…
- Số lượng node lớn nhất của cây nhị phân có chiều cao h là 2^h-1. Điều này có thể suy luận trực tiếp từ tính chất 1 vì $N \leq 2^0 + 2^1 + \ldots + 2^{h-1} = 2^h - 1$.
- Đối với cây nhị phân có N node thì chiều cao tối thiểu hay mức tối thiểu của cây là $\lceil log_2(N+1) \rceil$.
- Một cây nhị phân có L node lá có chiều cao tối đa là $\lceil log_2(L) \rceil + 1$.
- Trong cây nhị phân số lượng các node lá luôn nhiều hơn các node có hai node con 1 đơn vị.

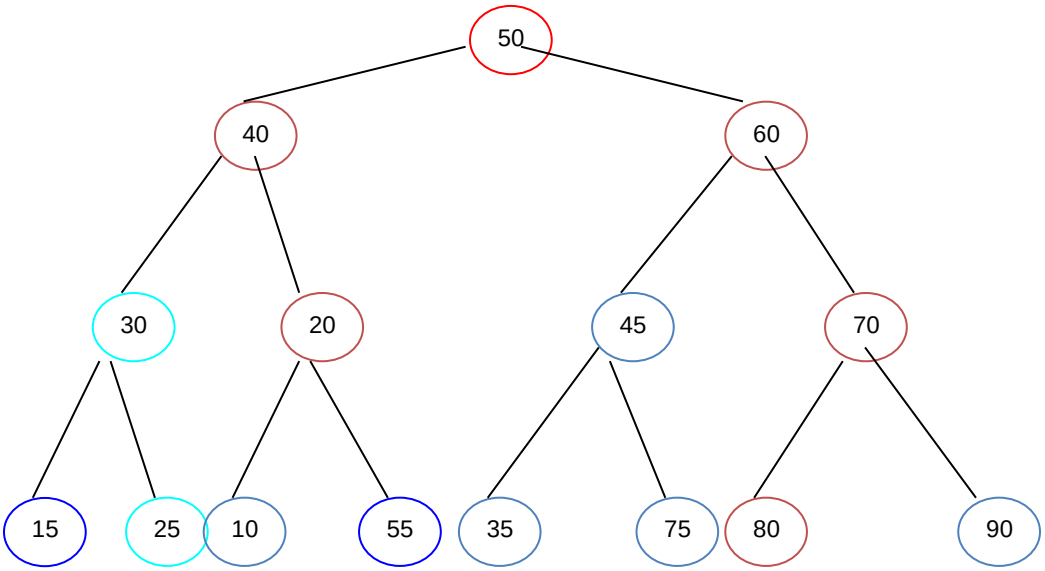
5.1.3. Các loại cây nhị phân

Dưới đây là một số loại cây nhị phân đặc biệt:

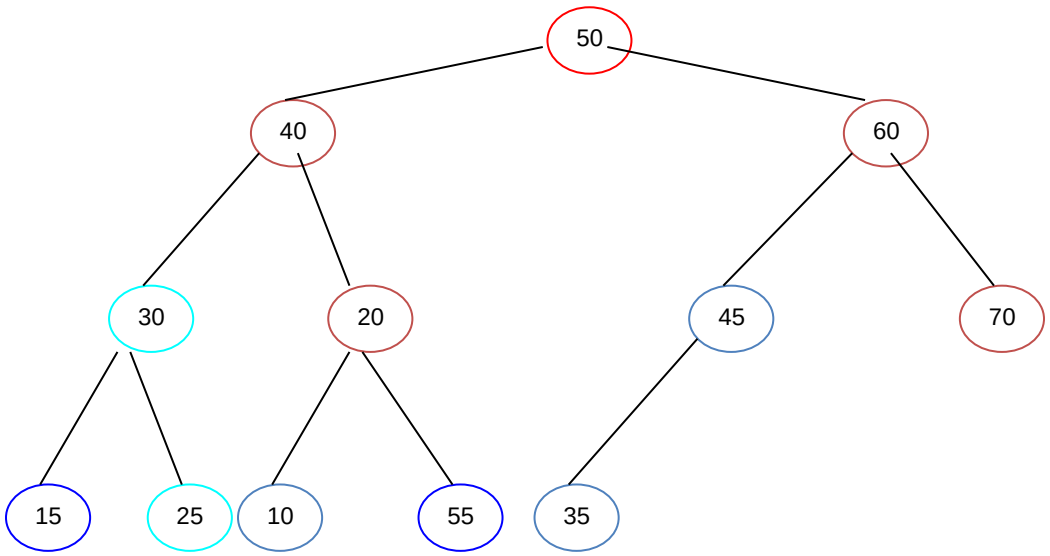
- **Cây lệch trái:** cây nhị phân chỉ có node con bên trái.
- **Cây lệch phải**: cây chỉ có node con bên phải.
- **Cây nhị phân đúng (strickly binary tree):** node gốc và tất cả các node trung gian đều có đúng hai node con (Hình 4.2).
- **Cây nhị phân đầy đủ(complete binary tree)**: cây nhị phân đúng và tất cả node lá đều có mức là d (Hình 4.3).
- **Cây nhị phân gần đầy (almost complete binary tree)**:Cây nhị phân gần đầy có chiều sâu d là cây nhị phân thỏa mãn (Hình 4.4):
  - Tất cả node con có mức không nhỏ hơn d-1 đều có hai node con (cây nhị phân đầy đủ mức d-1).
  - Các node ở mức d đầy từ trái qua phải.
- **Cây nhị phân hoàn toàn cân bằng.**Cây nhị phân có số node thuộc nhánh cây con trái và số node thuộc nhánh cây con phải chênh lệch nhau không quá 1 (Hình 4.5).
- **Cây nhị phân tìm kiếm**. Cây nhị phân thỏa mãn điều kiện (Hình 4.6):
  - Hoặc là rỗng hoặc có một node gốc.
  - Mỗi node gốc có tối đa hai cây con. Nội dung node gốc lớn hơn nội dung node con bên trái và nhỏ hơn nội dung node con bên phải.
  - Hai cây con bên trái và bên phải cũng hình thành nên hai cây tìm kiếm.
- **Cây nhị phân tìm kiếm hoàn toàn cân bằng.** Cây nhị phân tìm kiếm có chiều sâu cây con bên trái và chiều sâu cây con bên phải chênh lệch nhau không quá 1 (Hình 4.7).



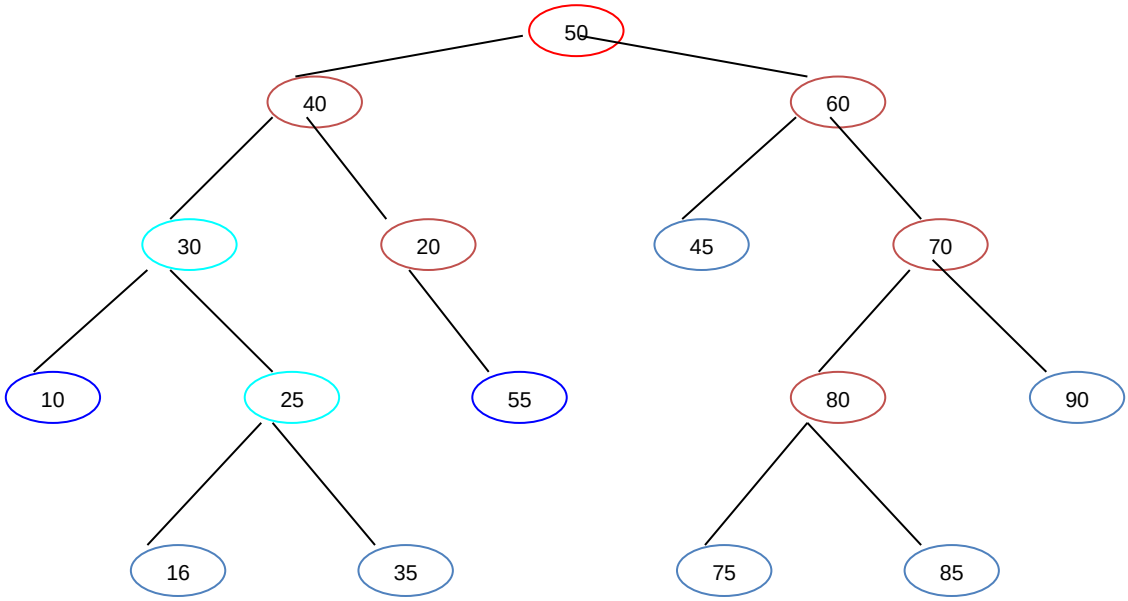
Hình 4.2. Cây nhị phân đúng



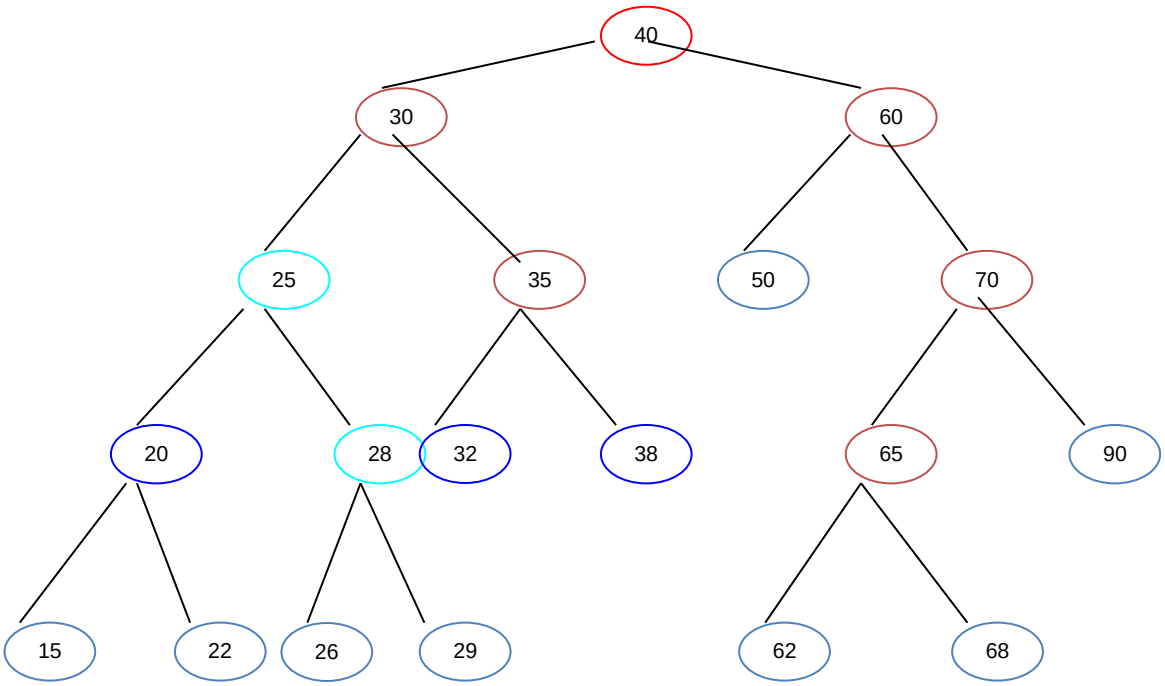
Hình 4.3. Cây nhị phân đầy đủ



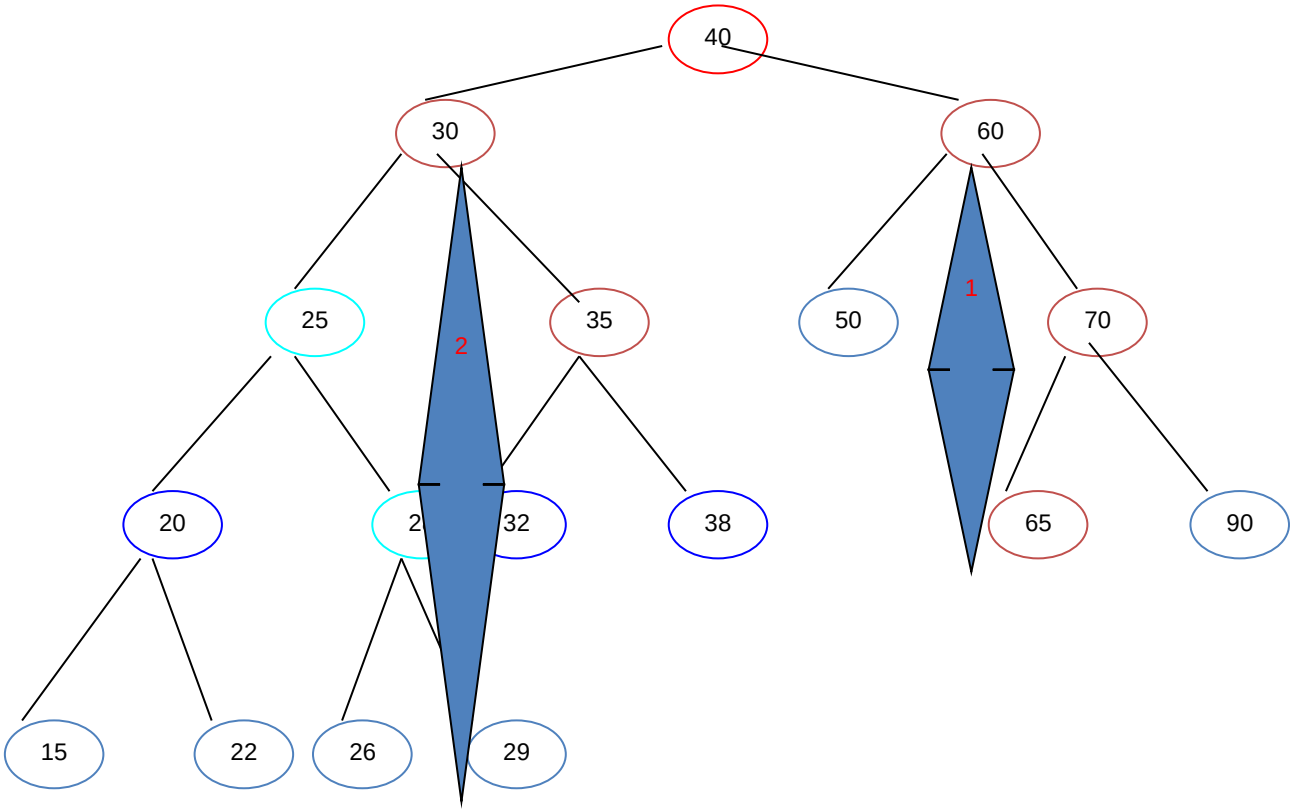
Hình 4.4. Cây hịn phân gần đầy



Hình 4.5. Cây nhị phân hoàn toàn cân bằng



Hình 4.6. Cây nhị phân tìm kiếm



Hình 4.7. Cây nhị phân tìm kiếm hoàn toàn cân bằng

5.2. Biểu diễn cây nhị phân

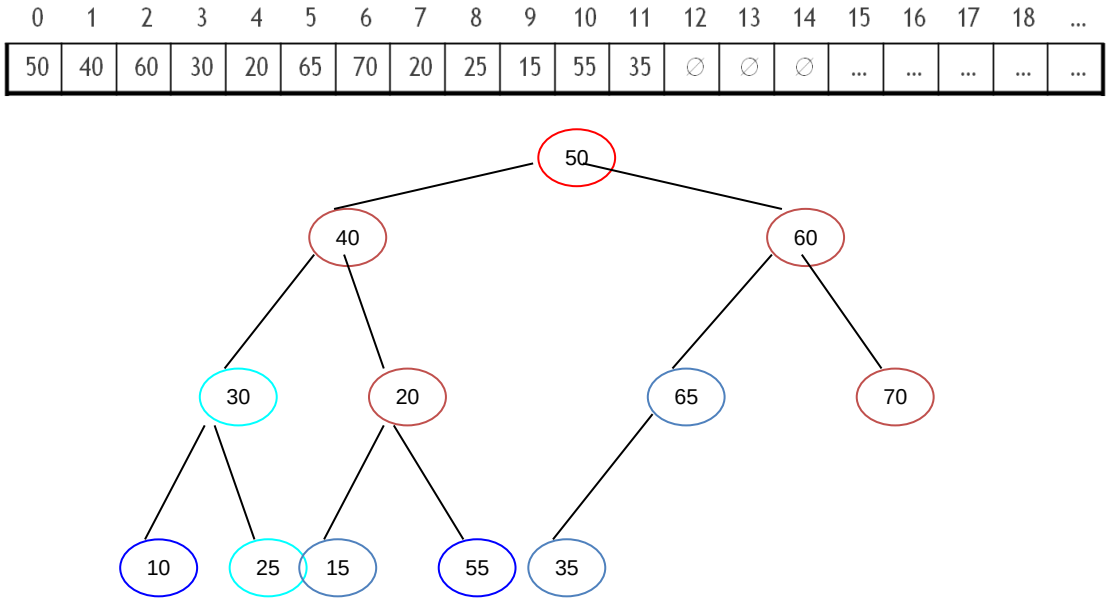
Có hai phương pháp biểu diễn cây nhị phân: biểu diễn liên tục và biểu diễn rời rạc. Trong trường hợp biểu diễn liên tục ta sử dụng mảng. Trong trường hợp biểu diễn rời rạc ta sử dụng danh sách liên kết. Phương pháp biểu diễn được thực hiện như dưới đây.

5.2.1. Biểu diễn cây nhị phân bằng mảng

Tổ chức lưu trữ các node của cây bằng một mảng node[MAX]. Trong đó:

- Node gốc được lưu trữ tại vị trí node[0].
- Node con trái và node con phải của node[p] ở các vị trí tương ứng là node[2p+1] và node[2p+2].

Hình 4.8 dưới đây mô tả chi tiết phương pháp biểu diễn cây nhị phân dựa vào mảng.



Hình 4.8. Biểu diễn cây nhị phân bằng mảng

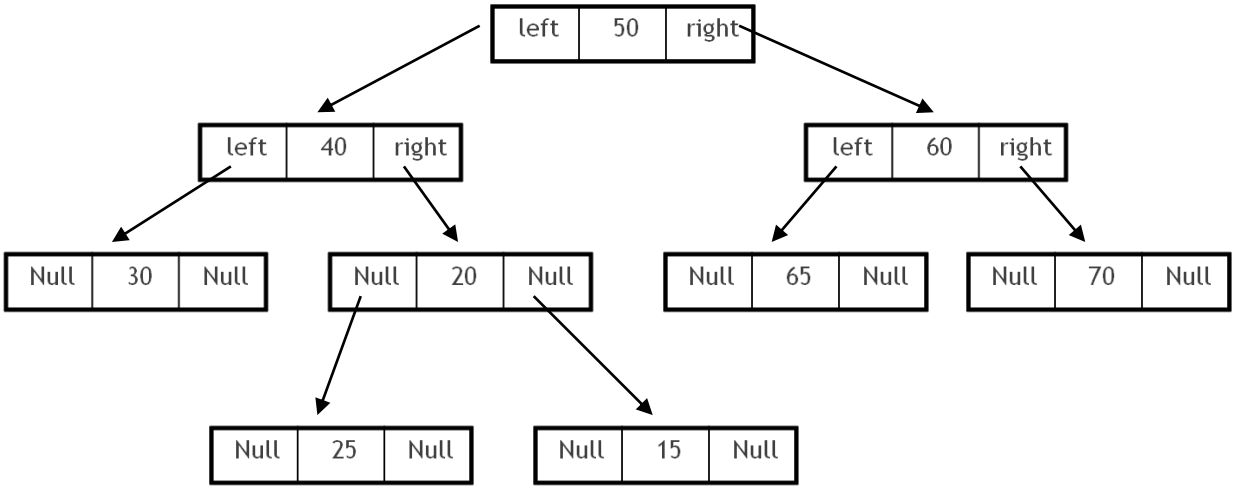
5.2.2. Biểu diễn cây nhị phân bằng danh sách liên kết

Định nghĩa mỗi node của cây như một node của danh sách liên kết kép. Mỗi node gồm có ba thành phần:

- Thành phần dữ liệu (data): dùng để lưu trữ thông tin của node.
- Thành phần con trỏ left: dùng để liên kết với các node con bên trái của node.
- Thành phần con trỏ right: dùng để liên kết với các node con bên phải của node.

```
struct node { //định nghĩa node
        int   data; //thành phần dữ liệu của node
        struct  node *left; //liên kết với node con bên trái
        struct  node *right; // liên kết với node con bên phải
}*Tree; //đây là một cây
```

Ví dụ với cây trong Hình 4.8 sẽ được biểu diễn như Hình 4.9 bằng danh sách liên kết.



Hình 4.9. Biểu diễn cây bằng danh sách liên kết

5.3. Các thao tác trên cây nhị phân

Các thao tác trên cây nhị phân bao gồm:

- Tạo node cho cây.
- Tạo node gốc cho cây.
- Thêm vào node lá bên trái node p.
- Thêm vào node lá bên phải node p.
- Loại bỏ node lá bên trái node p.
- Loại bỏ node lá bên phải node p.
- Loại bỏ cả cây.
- Duyệt cây theo thứ tự trước.
- Duyệt cây theo thứ tự giữa.
- Duyệt cây theo thứ tự sau.

5.3.1. Định nghĩa và khai báo cây nhị phân

Sử dụng phương pháp biểu diễn cây bằng danh sách liên kết, ta định nghĩa cây và lớp các thao tác trên cây như sau:

```
struct node { //định nghĩa cấu trúc node
        int data; //thành phần dữ liệu của node
        node *left;//thành phần liên kết với node con trái
```

```
        node *right; //thành phần liên kết với node con phải
    }*T; //cây nhị phân T


class Tree { //định nghĩa lớp các thao tác trên cây T
    public:
    Tree(void){ //constructor của lớp
        T=NULL; //cây ban đầu được khởi tạo là NULL
    }
    node *Make_Node(void);//tạo node rời rạc cóa giá trị value
    void Make_Root(void); //tạo node gốc cho cây
    void Insert_Left(node *root, int value); //tạo node lá trái cho node
    void Insert_Right(node *root, int value);//tạo node lá phải cho node
    void Remove_Left(node *root, int value); //loại node lá trái của node
    void Remove_Right(node *root, int value); //loại node lá phải của node
    void Clear_Tree(node *root); //loại bỏ cả cây
    void NLR(node *root); //duyệt cây theo thứ tự trước
    void LNR(node *root); //duyệt cây theo thứ tự giữa
    void LRN(node *root); //duyệt cây theo thứ tự sau
    void Function(void); //hàm kiểm tra các thao tác trên cây
};
```

5.3.2. Các thao tác thêm node vào cây nhị phân

Đối với cây nhị phân tổng quát, node đóng vai trò quan trọng nhất là node gốc (root) của cây (Make-Root()). Sau khi cây có node gốc, toàn bộ cây sẽ được định hình xung quanh node gốc. Quá trình định hình cây được thực hiện bởi các phép thêm node lá bên trái (Insert-Left())cho một node hoặc thêm node lá bên phải cho một node (Insert_Right()). Các thao tác thêm node cụ thể được tiến hành như sau:

Thao tác tạo node rời rạc có giá trị value:

```
node *Tree::Make_Node(void){ //tạo node rời rạc có giá trị value
    node *tmp; //sử dụng con trỏ node tmp
    int value;//giá trị node cần tạo ra
    tmp = new node; //cấp phát miền nhớ cho node
    cout<<"Giá trị node:"; cin>>value;
    tmp->data = value; //thiết lập thành phần dữ liệu của node
    tmp->left = NULL; //thiết lập liên kết trái cho node
    tmp->right = NULL; //thiết lập liên kết phải cho node
    return tmp; // node rời rạc được tạo ra
```

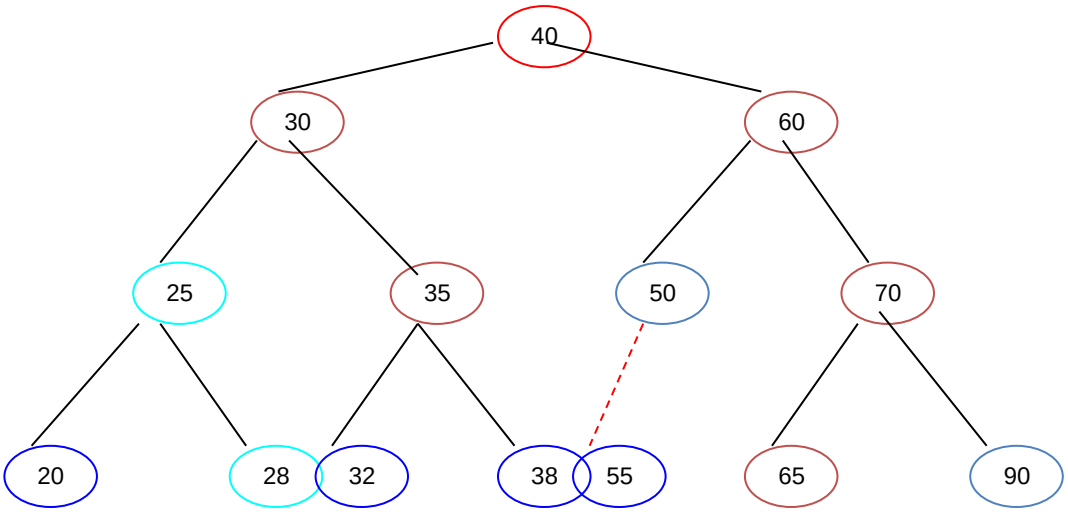
```
}
```

Thao tác tạo node gốc cho cây:

```
void Tree::Make_Root(void){ //tạo node gốc cho cây
    if (T!=NULL) {//nếu cây không rỗng
        cout<<"Cây đã có gốc"<<endl;
    }

    else {//nếu cây rỗng
        node *tmp=Make_Node(); //tạo node rời rạc có giá trị value
        T = tmp; // node gốc của cây là tmp
        tmp->left=NULL; // thiết lập liên kết trái cho node gốc
        tmp->right=NULL; // thiết lập liên kết phải cho node gốc
    }
}
```

Thao tác thêm node lá trái cho node có giá trị value: để thêm node lá bên trái cho node có giá trị value ta cần xem xét đầy đủ các trường hợp:

- **Trường hợp 1**: nếu node có giá trị value không có thực trên cây thì ta không thể thực hiện được. Để xác định điều này ta chỉ cần tìm node trên cây có giá trị value. Nếu node không có trên cây ta sẽ khống chế được trường hợp 1.
- **Trường hợp 2**: nếu node có giá trị value có thực trên cây nhưng node này đã có node con trái thì ta cũng không thể thực hiện được. Ví dụ ta không thể thêm vào node con trái cho node có giá trị value = 60 trong Hình 4.10. Điều này cũng được xác định bằng cách tìm node trên cây có giá trị value và kiểm tra sự tồn tại của node con bên trái..
- **Trường hợp 3**: nếu node có giá trị value chưa có node con trái thì ta tiến hành thiết lập liên kết của node đến node trái cần tạo ra. Ví dụ ta thêm vào node con trái cho node có giá trị value = 50 trong Hình 4.10. Điều này cũng được xác định bằng cách tìm node trên cây có giá trị value và kiểm tra sự tồn tại của node con bên trái.



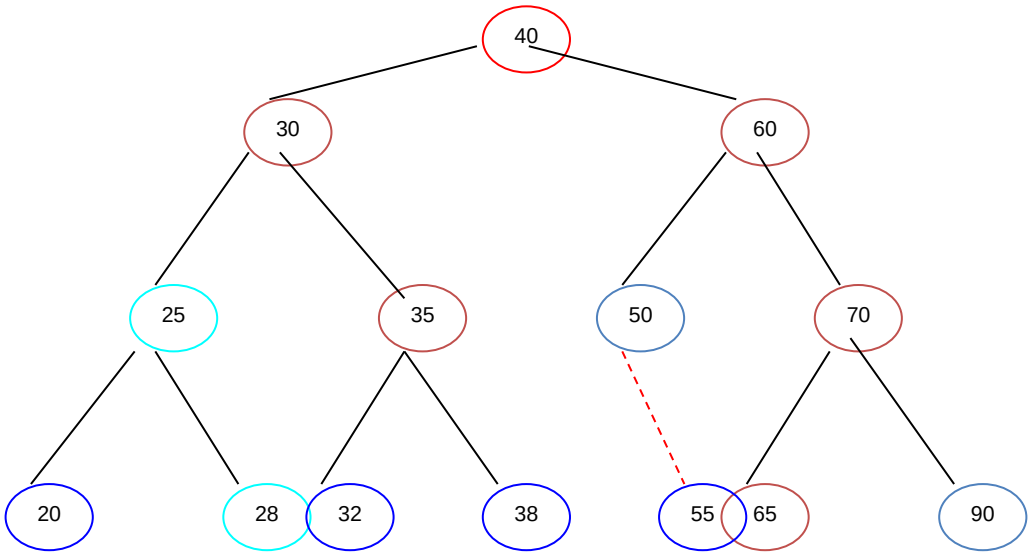
Hình 4.10. Thêm node lá bên trái cho node có giá trị value.

```
void Tree::Insert_Left(node *root, int value){//thêm node lá trái cho node có gí trị value
        if(root!=NULL){//nếu cây không rỗng
                if( root->data==value){//nếu tìm thấy node có giá trị value
                        if(root->left!=NULL){ //nếu node đã có node con trái
                                cout<<"Node "<<value<<" đã có node con trái"<<endl;
                                return;
                        }
                        else{//nếu node chưa có node con trái
                                root->left=Make_Node();//tạo node lá cho node
                                return;
                        }
                }
                Insert_Left(root->left,value);//tìm node sang nhánh cây con trái
                Insert_Left(root->right,value); //tìm node sang nhánh cây con phải
        }
}
```

Thao tác thêm node lá phải cho node có giá trị value: để thêm node lá bên phải cho node có giá trị value ta cần xem xét đầy đủ 3 trường hợp:

- **Trường hợp 1**: nếu node có giá trị value không có thực trên cây thì ta không thể thực hiện được. Để xác định điều này ta chỉ cần tìm node trên cây có giá trị value. Nếu node không có trên cây ta sẽ khống chế được trường hợp 1.

- **Trường hợp 2**: nếu node có giá trị value có thực trên cây nhưng node này đã có node con phải thì ta cũng không thể thực hiện được. Ví dụ ta không thể thêm vào node con phải cho node có giá trị value = 60 trong Hình 4.11. Điều này cũng được xác định bằng cách tìm node trên cây có giá trị value và kiểm tra sự tồn tại của node con bên phải.
- **Trường hợp 3**: nếu node có giá trị value chưa có node con phải thì ta tiến hành thiết lập liên kết của node đến node phải cần tạo ra. Ví dụ ta thêm vào node con phải cho node có giá trị value = 50 trong Hình 4.11. Điều này cũng được xác định bằng cách tìm node trên cây có giá trị value và kiểm tra sự tồn tại của node con bên phải.



Hình 4.11. Thêm node lá bên phải cho node có giá trị value.

```
void Tree::Insert_Right(node *root, int value){ //thêm node lá trái cho node
        if(root!=NULL){ //nếu cây không rỗng
                if( root->data==value){ //nếu tìm thấy node có giá trị value
                        if(root->right!=NULL){ //nếu node đã có node con phải
                                cout<<"Node "<<value<<" đã có node con phải"<<endl;
                                return;
                        }
                        else{//nếu node chưa có node conphải
                                root->right=Make_Node();//tạo node lá cho node
                                return;
                        }
                }
```

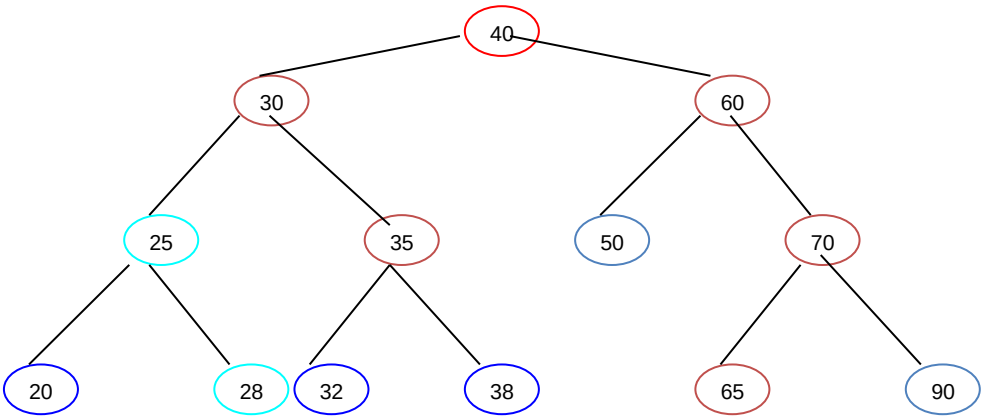
```
            Insert_Right(root->left,value); //tìm node sang nhánh cây con trái
            Insert_Right(root->right,value); //tìm node sang nhánh cây con phải
        }
}
```

5.3.3. Các thao tác loại node khỏi cây nhị phân

Các thao tác loại bỏ node khỏi cây nhị phân được thực hiện tương tự như người chặt cây: ngắt lá, ngắt cành cây, ngắt cả cây. Các thao tác loại bỏ node trên cây bao gồm: loại bỏ node lá bên trái cho node (Remove_Left()), loại bỏ node lá bên phải cho node (Remove_Right()), loại bỏ cây con bắt đầu tại một node (Clear_Tree()). Các thao tác được thực hiện như sau:

Thao tác loại bỏ node lá bên trái cho node có giá trị value: để thực hiện được thao tác này ta cần xem xét đầy đủ 4 khả năng có thể xảy ra:

- **Trường hợp 1**: nếu node có giá trị value không có thực trên cây thì ta không thể thực hiện được. Để xác định điều này ta chỉ cần tìm node có giá trị value trên cây. Nếu node không có trên cây ta sẽ khống chế được trường hợp 1.
- **Trường hợp 2**: nếu node có giá trị value có thực trên cây nhưng node này không có node lá bên trái thì ta cũng không thể thực hiện được. Ví dụ ta không thể tloại node lá trái của node có giá trị value = 38 trong Hình 4.12. Điều này cũng được thực hiện bằng cách tìm node trên cây có giá trị value và kiểm tra sự tồn tại của node lá bên trái.
- **Trường hợp 3**: nếu node có giá trị value có cả cây con trái thì ta cũng không thể thực hiện được. Ví dụ ta không thể loại node lá trái của node có giá trị value = 30 trong Hình 4.12. Điều này cũng được thực hiện bằng cách tìm node trên cây có giá trị value và kiểm tra sự tồn tại cây con trái của node.
- **Trường hợp 4**: nếu node có giá trị value có node lá bên trái thì ta tiến hành ngắt liên kết của node đến node lá bên trái. Ví dụ ta cần loại bỏ node lá bên trái cho node có giá trị value = 35 trong Hình 4.12. Điều này cũng được xác định bằng cách tìm node trên cây có giá trị value và kiểm tra sự tồn tại của node lá bên trái.

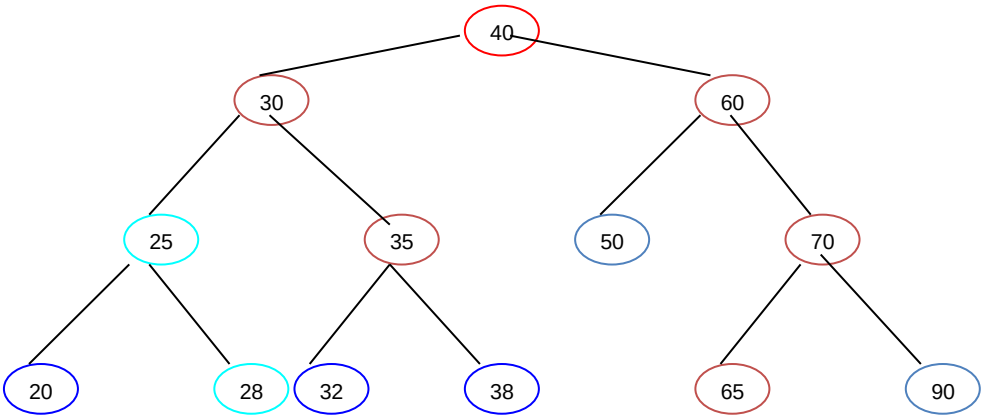


Hình 4.12. Loại bỏ node lá trái trên cây

```
void Tree::Remove_Left(node *root, int value){ //loại bỏ node lá bên trái cho node
        if(root!=NULL){//nếu cây không rỗng
                if( root->data==value){//nếu tìm thấy node có giá trị value
                        if(root->left==NULL){ //nếu node không có node con trái
                                cout<<"Node "<<value<<" không có lá trái"<<endl;
                                return;
                        }
                        else if((root->left)->left!=NULL ||(root->left)->right!=NULL ){
                                //nếu node có cây con trái
                                cout<<"Node "<<value<<" có cây con trái"<<endl;
                                return;
                        }
                        else if ((root->left)->left==NULL &&(root->left)->right==NULL ){
                                //nếu node có lá con trái
                                node *rp = root->left; //rp nhận node lá trái
                                root->left=NULL; //ngắt liên kết trái của node
                                delete(rp); //giải phóng node
                                return;
                        }
                }
                Remove_Left(root->left,value); //tìm node sang cây con trái
                Remove_Left(root->right,value); //tìm node sang cây con phải
        }
}
```

Thao tác loại bỏ node lá bên phải cho node có giá trị value: để thực hiện được thao tác này ta cần xem xét đầy đủ 4 khả năng có thể xảy ra:

- **Trường hợp 1**: nếu node có giá trị value không có thực trên cây thì ta không thể thực hiện được. Để xác định điều này ta chỉ cần tìm node có giá trị value trên cây. Nếu node không có trên cây ta sẽ khống chế được trường hợp 1.
- **Trường hợp 2**: nếu node có giá trị value có thực trên cây nhưng node này không có node lá bên phải thì ta cũng không thể thực hiện được. Ví dụ ta không thể tloại node lá phải của node có giá trị value = 38 trong Hình 4.13. Điều này cũng được thực hiện bằng cách tìm node trên cây có giá trị value và kiểm tra sự tồn tại của node lá bên phải.
- **Trường hợp 3**: nếu node có giá trị value có cả cây con phải thì ta cũng không thể thực hiện được. Ví dụ ta không thể loại node lá phải của node có giá trị value = 30 trong Hình 4.13. Điều này cũng được thực hiện bằng cách tìm node trên cây có giá trị value và kiểm tra sự tồn tại cây con phải của node.
- **Trường hợp 4**: nếu node có giá trị value có node lá bên phải thì ta tiến hành ngắt liên kết của node đến node lá bên phải. Ví dụ ta cần loại bỏ node lá bên trái cho node có giá trị value = 35 trong Hình 4.12. Điều này cũng được xác định bằng cách tìm node trên cây có giá trị value và kiểm tra sự tồn tại của node lá bên phải.



Hình 4.13. Loại bỏ node lá phải trên cây

```
void Tree::Remove_Right(node *root, int value){ //loại bỏ node lá bên phải cho node
    if(root!=NULL){ //nếu cây không rỗng
        if( root->data==value){ //nếu tìm thấy node có giá trị value
            if(root->right==NULL){ //nếu node không có node con phải
                cout<<"Node trai cua "<<value<<" khong co la phai"<<endl;
                return;
            }
            else if((root->right)->left!=NULL ||(root->right)->right!=NULL ){
```

```
                    //nếu node có cây con phải
                    cout<<"Node "<<value<<" có cây con phải"<<endl;
                    return;
                }
                else if ((root->right)->left==NULL&&(root->right)->right==NULL ){
                    //nếu node có node lá phải
                    node *rp = root->right; //rp nhận node lá bên phải
                    root->right=NULL; //ngắt liên kết phải cho node
                    delete(rp); //giải phóng node lá phải
                    return;
                }
            }
            Remove_Right(root->left,value); //tìm node sang cây con trái
            Remove_Right(root->right,value); //tìm node sang cây con phải
        }
}
```

Thao tác loại bỏ cây con gốc root: nguyên tắc loại bỏ cây con gốc root được thực hiện bằng cách giải phóng từng node lá trên cây cho đến khi giải phóng đến node gốc. Thao tác được tiến hành như sau:

```
    void Tree::Clear_Tree(node *root){ //loại bỏ cả cây gốc root
        if(root!=NULL){ //nếu cây rỗng
            Clear_Tree(root->left); //loại loại cây con bên trái
            Clear_Tree(root->right); //loại loại cây con bên phải
            cout<<"Node được giải phóng:"<<root->data<<endl;
            delete(root);//giải phóng node
        }
    }
```

5.3.4. Ba phép duyệt cây nhị phân

Có ba phép duyệt cơ bản trên cây nhị phân: duyệt theo thứ tự trước (inoder), duyệt theo thứ tự giữa (preoder), và duyệt theo thứ tự sau (postoder). Điểm khác biệt duy nhất giữa các phép duyệt cây là node được thăm theo thứ tự trước, giữa hay sau. Chú ý, ba phép duyệt cây cũng được sử dụng như ba phép tìm kiếm node trên cây. Dưới đây là ba phép duyệt cây.

Phép duyệt theo thứ tự trước:

```
        void Tree::NLR(node *root){ //duyệt cây theo thứ tự trước
            if(root!=NULL){//nếu cây không rỗng
```

```
            cout<<root->data<<setw(3); //thăm node
            NLR(root->left);//duyệt theo thứ tự trước cây con trái
            NLR(root->right); //duyệt theo thứ tự trước cây con phải
        }
    }
```

Phép duyệt theo thứ tự giữa:

```
    void Tree::LNR(node *root){ //duyệt cây theo thứ tự giữa
        if(root!=NULL){ //nếu cây không rỗng
            LNR(root->left); //duyệt theo thứ tự giữa cây con trái
            cout<<root->data<<setw(3); //thăm node
            LNR(root->right); //duyệt theo thứ tự giữa cây con phải
        }
    }
```

Phép duyệt theo thứ tự sau:

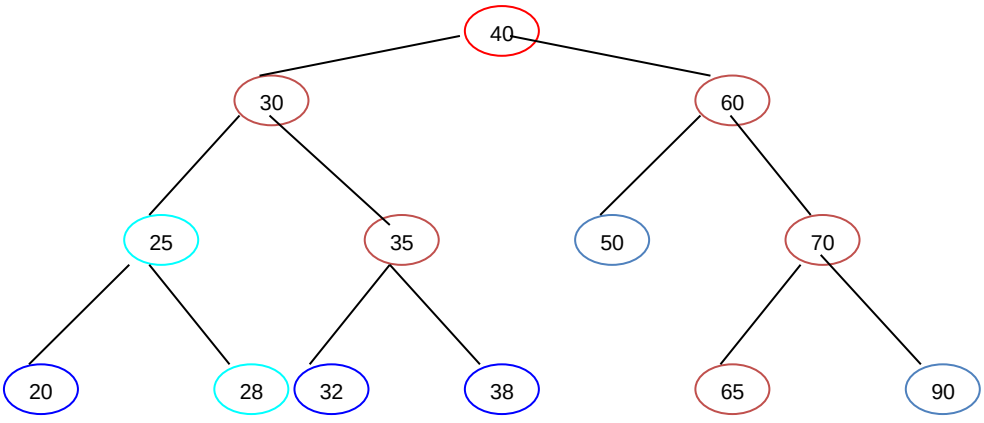
```
    void Tree::LRN(node *root){ //duyệt cây theo thứ tự sau
        if(root!=NULL){ //nếu cây không rỗng
            LRN(root->left); //duyệt theo thứ tự sau cây con trái

            LNR(root->right); //duyệt theo thứ tự sau cây con phải
            cout<<root->data<<setw(3); //thăm node
        }
    }
```

Ví dụ với cây Hình 4.14 sẽ cho ta kết quả của ba phép duyệt như sau:

NLR(T) = 40, 30, 25, 20, 28, 35, 32, 38, 60, 50, 70, 65, 90.

LNR(T) = 20, 25, 28, 30, 32, 35, 38, 40, 50, 60, 65, 70, 90.

LRN (T) = 20, 28, 25, 32, 38, 35, 30, 50, 65, 90, 70, 60, 40.



Hình 4.14. Loại bỏ node lá phải trên cây

5.3.5. Chương trình cài đặt các thao tác trên cây nhị phân

```
#include <iostream>
#include <iomanip>
using namespace std;
struct node { //định nghĩa cấu trúc node
        int data; //thành phần dữ liệu
        node *left;//thành phần liên kết với node con trái
        node *right; //thành phần liên kết với node con phải
}*T; //cây nhị phân T

class Tree { //định nghĩa lớp các thao tác trên cây T
  public:
        Tree(void){ //constructor của lớp
                T=NULL; //cây ban đầu được khởi tạo là NULL
        }
        node *Make_Node(void);//tạo node rời rạc cho cây
        void Make_Root(void); //tạo node gốc cho cây
        void Insert_Left(node *root, int value); //tạo node lá trái cho node
        void Insert_Right(node *root, int value);//tạo node lá phải cho node
        void Remove_Left(node *root, int value); //loại node lá trái của node
        void Remove_Right(node *root, int value); //loại node lá phải của node
        void Clear_Tree(node *root); //loại bỏ cả cây
        void NLR(node *root); //duyệt cây theo thứ tự trước
        void LNR(node *root); //duyệt cây theo thứ tự giữa
        void LRN(node *root); //duyệt cây theo thứ tự sau
        void Function(void); //hàm kiểm tra các thao tác trên cây
};
node *Tree::Make_Node(void){ //tạo node rời rạc có giá trị value
        node *tmp; int value;
        tmp = new node;
        cout<<"Giá trị node:"; cin>>value;
        tmp->data = value;
        tmp->left = NULL;
        tmp->right = NULL;
        return tmp;
}
```

```
void Tree::Make_Root(void){ //tạo node gốc cho cây
        if(T==NULL){//nếu cây rỗng
                node *tmp=Make_Node();
                T = tmp; tmp->left=NULL;tmp->right=NULL;
        }
        else {//nếu cây không rỗng
                cout<<"Cây đã có gốc"<<endl;
        }
}
void Tree::Insert_Left(node *root, int value){//thêm node lá trái cho node có gí trị value
        if(root!=NULL){//nếu cây không rỗng
                if( root->data==value){//nếu tìm thấy node có giá trị value
                        if(root->left!=NULL){ //nếu node đã có node con trái
                                cout<<"Node "<<value<<" đã có node con trái"<<endl;
                                return;
                        }
                        else{//nếu node chưa có node con trái
                                root->left=Make_Node();//tạo node lá cho node
                                return;
                        }
                }
                Insert_Left(root->left,value);//tìm node sang nhánh cây con trái
                Insert_Left(root->right,value); //tìm node sang nhánh cây con phải
        }
}
void Tree::Insert_Right(node *root, int value){ //thêm node lá trái cho node
        if(root!=NULL){ //nếu cây không rỗng
                if( root->data==value){ //nếu tìm thấy node có giá trị value
                        if(root->right!=NULL){ //nếu node đã có node con phải
                                cout<<"Node "<<value<<" đã có node con phải"<<endl;
                                return;
                        }
                        else{//nếu node chưa có node conphải
                                root->right=Make_Node();//tạo node lá cho node
                                return;
                        }
                }
```

```
            Insert_Right(root->left,value); //tìm node sang nhánh cây con trái
            Insert_Right(root->right,value); //tìm node sang nhánh cây con phải
        }
}
void Tree::Remove_Left(node *root, int value){ //loại bỏ node lá bên trái cho node
        if(root!=NULL){//nếu cây không rỗng
            if( root->data==value){//nếu tìm thấy node có giá trị value
                if(root->left==NULL){ //nếu node không có node con trái
                    cout<<"Node "<<value<<" không có lá trái"<<endl;
                    return;
                }
                else if((root->left)->left!=NULL ||(root->left)->right!=NULL ){
                    //nếu node có cây con trái
                    cout<<"Node "<<value<<" có cây con trái"<<endl;
                    return;
                }
                else if ((root->left)->left==NULL &&(root->left)->right==NULL ){
                    //nếu node có lá con trái
                    node *rp = root->left; //rp nhận node lá trái
                    root->left=NULL; //ngắt liên kết trái của node
                    delete(rp); //giải phóng node
                    return;
                }
            }
            Remove_Left(root->left,value); //tìm node sang cây con trái
            Remove_Left(root->right,value); //tìm node sang cây con phải
        }
}
void Tree::Remove_Right(node *root, int value){ //loại bỏ node lá bên phải cho node
        if(root!=NULL){ //nếu cây không rỗng
            if( root->data==value){ //nếu tìm thấy node có giá trị value
                if(root->right==NULL){ //nếu node không có node con phải
                    cout<<"Node trai cua "<<value<<" khong co la phai"<<endl;
                    return;
                }
                else if((root->right)->left!=NULL ||(root->right)->right!=NULL ){
                    //nếu node có cây con phải
```

```
                        cout<<"Node "<<value<<" có cây con phải"<<endl;
                        return;
                    }
                    else if ((root->right)->left==NULL&&(root->right)->right==NULL ){
                        //nếu node có node lá phải
                        node *rp = root->right; //rp nhận node lá bên phải
                        root->right=NULL; //ngắt liên kết phải cho node
                        delete(rp); //giải phóng node lá phải
                        return;
                    }
                }
                Remove_Right(root->left,value); //tìm node sang cây con trái
                Remove_Right(root->right,value); //tìm node sang cây con phải
        }
}
void Tree::Clear_Tree(node *root){ //loại bỏ cả cây gốc root
        if(root!=NULL){ //nếu cây rỗng
                Clear_Tree(root->left); //loại loại cây con bên trái
                Clear_Tree(root->right); //loại loại cây con bên phải
                cout<<"Node được giải phóng:"<<root->data<<endl;
                delete(root);//giải phóng node
        }
}
void Tree::NLR(node *root){ //duyệt cây theo thứ tự trước
        if(root!=NULL){//nếu cây không rỗng
                cout<<root->data<<setw(3); //thăm node
                NLR(root->left);//duyệt theo thứ tự trước cây con trái
                NLR(root->right); //duyệt theo thứ tự trước cây con phải
        }
}
void Tree::LNR(node *root){ //duyệt cây theo thứ tự giữa
        if(root!=NULL){ //nếu cây không rỗng
                LNR(root->left); //duyệt theo thứ tự giữa cây con trái
                cout<<root->data<<setw(3); //thăm node
                LNR(root->right); //duyệt theo thứ tự giữa cây con phải
        }
}
```

```
void Tree::LRN(node *root){ //duyệt cây theo thứ tự sau
        if(root!=NULL){ //nếu cây không rỗng
                LRN(root->left); //duyệt theo thứ tự sau cây con trái
                LNR(root->right); //duyệt theo thứ tự sau cây con phải
                cout<<root->data<<setw(3); //thăm node
        }
}
void Tree:: Function(void){
        int choice;int value;
        node *tmp, *root;
        do {
                cout<<"\n CÁC THAO TÁC TRÊN CÂY"<<endl;
                cout<<"1. Tạo node gốc"<<endl;
                cout<<"2. Tạo node lá trái"<<endl;
                cout<<"3. Tạo node lá phải "<<endl;
                cout<<"4. Loại bỏ node lá trái "<<endl;
                cout<<"5. Loại bỏ node lá phải "<<endl;
                cout<<"6. Loại bỏ cả cây "<<endl;
                cout<<"7. Duyệt Node Left Right"<<endl;
                cout<<"8. Duyệt Left Node Right"<<endl;
                cout<<"9. Duyệt Left Right Node"<<endl;
                cout<<"0. Thoát "<<endl;
                cout<<"Đưa vào lựa chọn:"; cin>>choice;
                switch(choice){
                        case 1: Make_Root(); break;
                        case 2:
                                root = T; cout<<"Giá trị node lá trái:"; cin>>value;
                                Insert_Left(root, value);break;
                        case 3:
                                root = T; cout<<" Giá trị node lá phải:"; cin>>value;
                                Insert_Right(T, value);break;
                        case 4:
                                root =T; cout<<" Giá trị node:"; cin>>value;
                                Remove_Left(root, value);break;
                        case 5:
                                root =T; cout<<" Giá trị node:"; cin>>value;
                                Remove_Right(root, value);break;
```

```
                case 6:
                        Clear_Tree(T); T=NULL; break;
                case 7:
                        cout<<"Duyệt NLR:"; root = T; NLR(root);
                        cout<<endl; break;
                case 8:
                        cout<<"Duyệt LNR:";root = T; LNR(root);
                        cout<<endl; break;
                case 9:
                        cout<<"Duyệt LRN:";root = T; LRN(root);
                        cout<<endl; break;
            }
        }while(choice!=0);
}
int main(void){
        Tree X; X.Function();
}
```

5.4. Ứng dụng của cây nhị phân

Cây nhị phân được ứng dụng trong các lĩnh vực sau:

- Tổ chức lưu trữ dữ liệu lớn: cây tìm kiếm, cây từ điển, cây top-down, cây tìm kiếm nhiều nhánh.
- Tìm kiếm và truy xuất thông tin trên các ứng dụng dữ liệu lớn.
- Biểu diễn tính toán: cây biểu thức, cây quyết định.

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

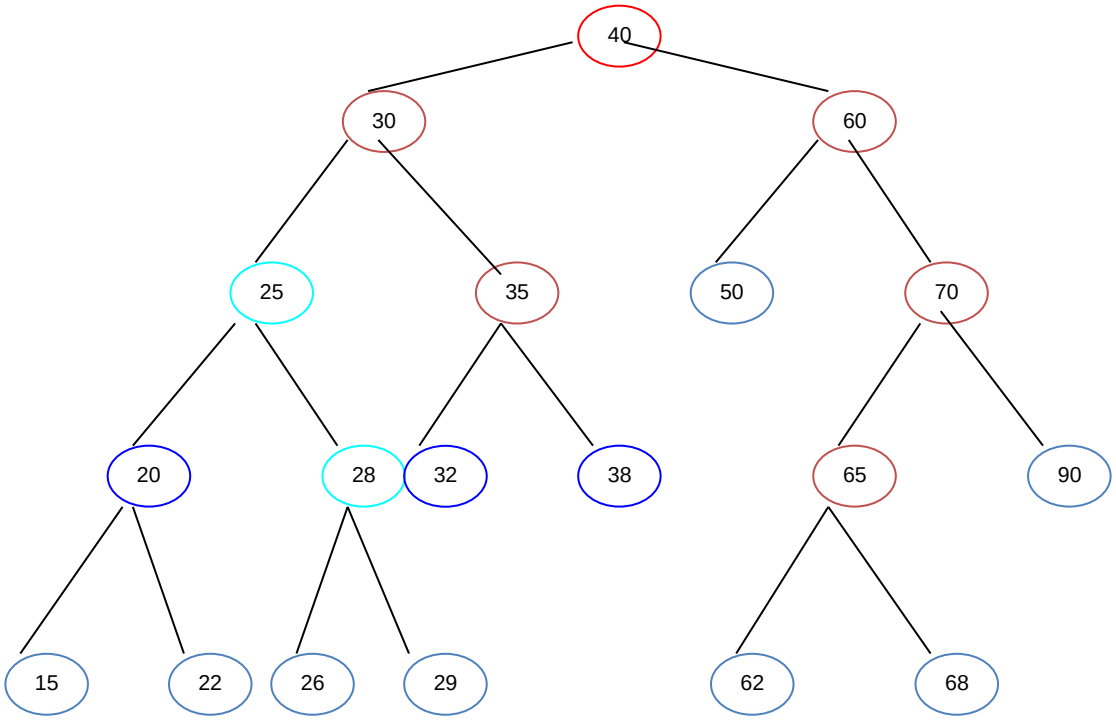
Đối với cây nhị phân tổng quát ta chỉ mới xem xét đến khía cạnh tổ chức lưu trữ dữ liệu. Phép tìm kiếm và truy xuất dữ liệu trên cây tổng quát không tốt hơn các cấu trúc dữ liệu mảng hay danh sách liên kết. Thậm chí, cây nhị phân tổng quát còn phải sử dụng nhiều không gian nhớ hơn cho các con trỏ NULL để xác định các node lá. Mục này ta sẽ xem xét cây nhị phân tìm kiếm là cây có vai trò đặc biệt quan trọng trong tổ chức lưu trữ và tìm kiếm dữ liệu.

5.5.1. Định nghĩa cây nhị phân tìm kiếm

Cây nhị phân tìm kiếm là một cây nhị phân thỏa mãn hai điều kiện:

- Hoặc là rỗng hoặc có một node gốc. Mỗi node gốc có tối đa hai cây con.

- Nội dung node gốc lớn hơn nội dung node con bên trái và nhỏ hơn nội dung node con bên phải. Hai cây con bên trái và bên phải cũng hình thành nên hai cây nhị phân tìm kiếm.



Hình 4.15. Ví dụ về cây nhị phân tìm kiếm.

5.5.2. Biểu diễn cây nhị phân tìm kiếm

Biểu diễn cây nhị phân tìm kiếm cũng giống như biểu diễn của các cây nhị phân thông thường. Trong trường hợp biểu diễn liên tục ta sử dụng mảng. Trong trường hợp biểu diễn rời rạc ta sử dụng danh sách liên kết.

```
struct node {
        int data; //thông tin của node
        struct node *left; //con trỏ trỏ đến cây con trái
        struct node *right; // con trỏ trỏ đến cây con phải
} *BST_Tree;
```

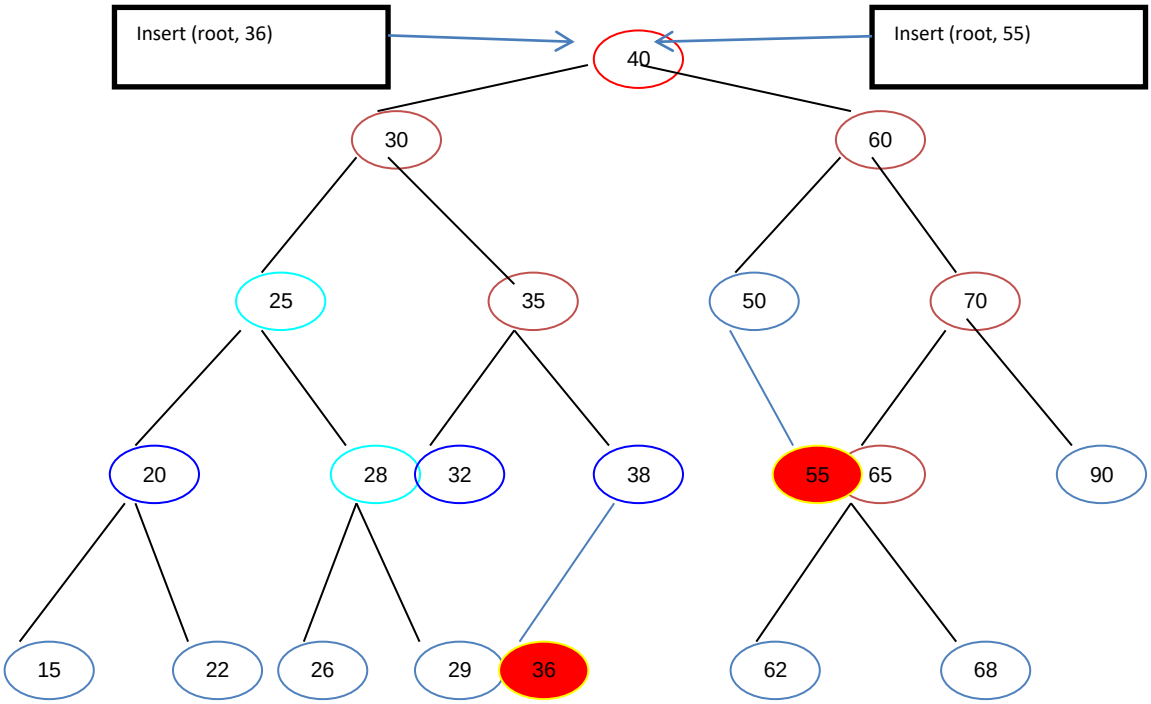
5.5.3. Các thao tác trên cây nhị phân tìm kiếm

Khác với cây nhị phân thông thường, phép thêm node vào cây tìm kiếm được thực hiện tự động sao cho sau khi thêm node ta vẫn nhận được một cây nhị phân tìm kiếm. Phép loại bỏ node được thực hiện trên tất cả các node kể cả node trung gian. Sau khi loại bỏ node ta phải tìm một node thích hợp trên cây để thay thế cho node vừa loại bỏ sao cho ta vẫn nhận được một cây nhị phân tìm kiếm. Phép tìm node trên cây cũng được thực hiện tốt hơn so với cây nhị phân thông thường. Các phép duyệt trên cây cũng được thực

hiện giống như cây nhị phân thông thường. Các thao tác cụ thể trên cây nhị phân tìm kiếm bao gồm:

- Thêm vào node vào cây nhị phân tìm kiếm.
- Loại bỏ node trên cây nhị phân tìm kiếm.
- Tìm kiếm node trên cây nhị phân tìm kiếm .
- Duyệt cây theo thứ tự trước.
- Duyệt cây theo thứ tự giữa.
- Duyệt cây theo thứ tự sau.

Thao tác thêm node vào cây nhị phân tìm kiếm:



Hình 4.16. Thêm node vào cây BST

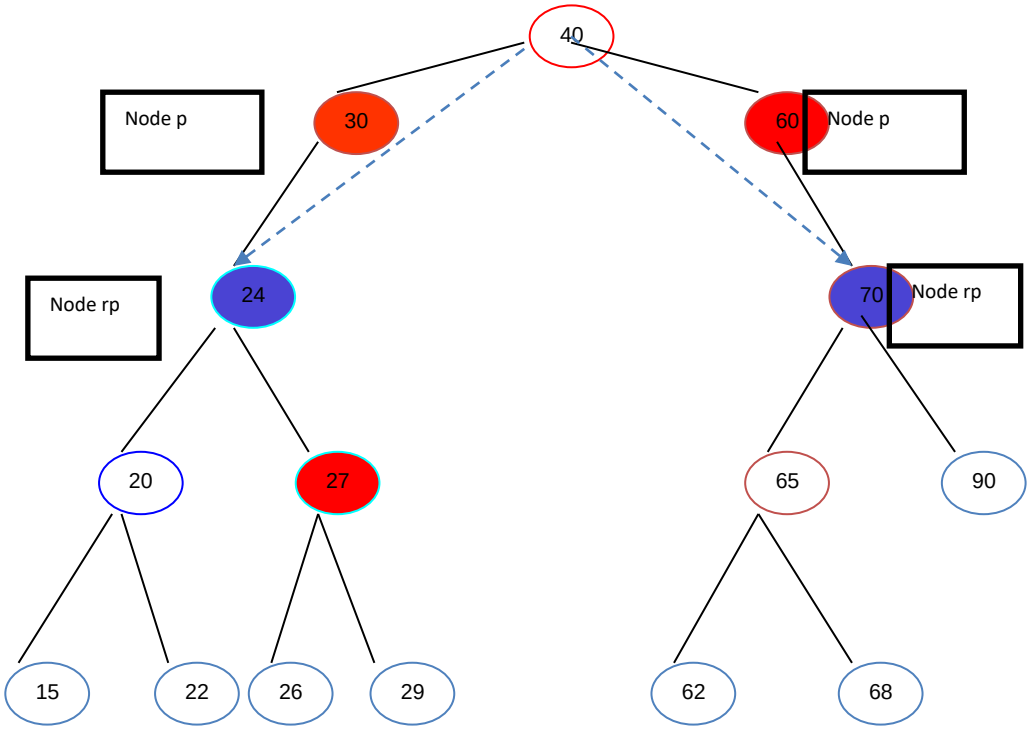
Để thêm node vào cây nhị phân tìm kiếm ta cần xem xét những trường hợp sau:

- Nếu cây rỗng thì node cần thêm chính là node gốc.
- Nếu cây không rỗng và node cần thêm có giá trị nhỏ hơn giá trị node gốc thì ta thêm node vào cây con bên trái. Ví dụ ta cần thêm node có giá trị 36 trên cây Hình 4.16.
- Nếu cây không rỗng và node cần thêm có giá trị lớn hơn giá trị node gốc thì ta thêm node vào cây con bên phải. Ví dụ ta cần thêm node có giá trị 55 trên cây Hình 4.16.

node* BST::insert(node* root, int value){//*thêm node có giá trị value*

```
        if (root == NULL) //nếu cây rỗng
            return newNode(value); //đây chính là node gốc
        if (value < root ->data) //nếu điều này xảy ra
            root ->left  = insert(root ->left, value);//ta thêm vào cây con trái
        else //trường hợp value>node->data
            root ->right = insert(root ->right, value); //ta thêm vào cây con phải
        return root;//trả lại node
    }
```

Loại bỏ node trên cây nhị phân tìm kiếm:



Hình 4.17. Loại bỏ node trên cây tìm kiếm.

Để bảo toàn phép loại bỏ node trên cây nhị phân tìm kiếm ta cần xem xét đầy đủ các trường hợp sau:

- **Trường hợp 1**: nếu cây rỗng thì phép loại bỏ hiển nhiên được bảo toàn.
- **Trường hợp 2**: nếu node loại bỏ là node lá thì sau khi loại bỏ node ta vẫn nhận được một cây nhị phân tìm kiếm. Ví dụ ta loại bỏ các node có giá trị 15, 22 trên cây Hình 4.17.
- **Trường hợp 3**: nếu node loại bỏ chỉ có cây con phải. Ví dụ ta cần loại bỏ node p = 60 trên cây Hình 4.17. Trong trường hợp này ta chỉ việc liên kết node cha của p với nhánh cây con phải của p.

- **Trường hợp 4**: nếu node loại bỏ chỉ có cây con trái. Ví dụ ta cần loại bỏ node p = 30 trên cây Hình 4.17. Trong trường hợp này ta chỉ việc liên kết node cha của p với nhánh cây con trái của p.
- **Trường hợp 5**: nếu node loại bỏ có cả hai cây con. Ví dụ ta cần loại bỏ node 24 trên cây Hình 4.17. Trong trường hợp này ta được phép lấy node có giá trị lớn nhất thuộc nhánh cây con bên trái của node 24 (node 22) hoặc lấy node có giá trị nhỏ nhất thuộc nhánh cây con phải node 24 (node 26) thay thế vào node cần loại bỏ.

```
node*BST:: deleteNode(node* root, int value){//loại bỏ node có giá trị value
        if (root == NULL) //nếu cây  rỗng
                return root;//không có gì để loại bỏ
        if (value < root->data) //nếu node cần loại có giá trị nhỏ hơn node gốc
                root->left = deleteNode(root->left, value); //tìm sang cây con trái để loại
        else if (value > root->data) //nếu node cần loại có giá trị lớn hơn node gốc
                root->right = deleteNode(root->right,value); //tìm sang cây con phải để loại
        else { //Chú ý chỗ này: nếu tìm thấy node có giá trị value cần loại bỏ
                if (root->left == NULL){ //nếu node cần loại chỉ có cây con phải
                        node *temp = root->right;
                        free(root);
                        return temp;
                }
                else if (root->right == NULL)  {//nếu node cần loại chỉ có cây con trái
                        node *temp = root->left;
                        free(root);
                        return temp;
                }
                // trường hợp node có cả hai cây con
                //ta lấy node trái nhất của cây con phải
                node* temp = minValueNode(root->right);
                root->data = temp->data; //thay thế nội dung node cần loại bỏ
                root->right = deleteNode(root->right, temp->data);//loại bỏ node thay thế
        }
        return root;
}
```

Tìm node trên cây nhị phân tìm kiếm: phép tìm node có giá trị value trên cây tìm kiếm được thực hiện như sau:

- Nếu node cần tìm có giá trị lớn hơn nội dung node gốc thì ta tìm sang cây con bên phải.
- Nếu node cần tìm có giá trị bé hơn nội dung node gốc thì ta tìm sang cây con bên trái.
- Đưa ra kết luận tìm thấy node hay không tìm thấy node.

```
node*BST:: search(node* root, int value){ //tìm node có giá trị value trên cây
        if (root == NULL || root->data == value) //nếu gốc rỗng hoặc tìm thấy node
                return root; //trả lại node
        if (root->data < value)//nếu node có giá trị bé hơn value
                return search(root->right, value);//tìm ở cây con phải
        return search(root->left, value); //tìm ở cây con trái
}
```

5.5.4. Chương trình cài đặt cây nhị phân tìm kiếm

```
//Cài đặt cây nhị phân tìm kiếm
#include <iostream>
#include <iomanip>
using namespace std;
struct node {//biểu diễn node trên cây
        int data;//thành phần dữ liệu của node
        struct node *left;//thành phần liên kết với cây con trái
        struct node *right;// thành phần liên kết với cây con phải
};
class BST {
  public:
        node *root; //đây là cây nhị phân gốc root
        BST(void){ //constructor của lớp
                root = NULL; //được thiết lập ban đầu là NULL
        }
        node *newNode(int value); //tạo node rời rạc có giá trị value
        node* search(node* root, int value); //tìm node có giá trị value trên cây
        void preorder(node *root);//duyệt theo thứ tự trước
        void inorder(node *root); //duyệt theo thứ tự giữa
        void postorder(node *root); //duyệt theo thứ tự sau
        node* insert(node* T, int value); //thêm node có giá trị value vào cây T
        node * minValueNode(node* p); //tìm node có giá trị nhỏ nhất trên cây gốc p
        node* deleteNode(node* root, int data);   //loại node có giá trị vaule
```

```
        void Function(void);//hàm thực thi các thao tác
};
node *BST::newNode(int value){ //tạo node có giá trị value
        node *temp =  new node;//cấp phát miền nhớ cho node
        temp->data = value; //thiết lập thành phần dữ liệu
        temp->left = NULL; //thiết lập liên kết trái cho node
        temp->right = NULL; //thiết lập liên kết phải cho node
        return temp; // node rời rạc được tạo ra
}
node*BST:: search(node* root, int value){ //tìm node có giá trị value trên cây
        if (root == NULL || root->data == value) //nếu gốc rỗng hoặc tìm thấy node
                return root; //trả lại node
        if (root->data < value)//nếu node có giá trị bé hơn value
                return search(root->right, value);//tìm ở cây con phải
        return search(root->left, value); //tìm ở cây con trái
}
void BST::preorder(node *root){ //duyệt theo thứ tự trước
        if (root != NULL){ //nếu cây không rỗng
                cout<<root->data<<setw(3);//thăm node
                preorder(root->left);//duyệt NLR cây con trái
                preorder(root->right);//duyet NLR cay con phải
        }
}
void BST::inorder(node *root){ //duyệt theo thứ tự giữa
        if (root != NULL){ //nếu cây không rỗng
                inorder(root->left); //duyệt LNR cây con trái
                cout<< root->data<<setw(3);//thăm node
                inorder(root->right); //duyệt LNR cây con phải
        }
}
void BST::postorder(node *root){ //duyệt theo thứ tự sau
        if (root != NULL){ //nếu cây không rỗng
                postorder(root->left); //duyệt LRN cây con trái
                postorder(root->right); //duyệt LRN cây con phải
                cout<< root->data<<setw(3);//thăm node
        }
}
```

```
node* BST::insert(node* root, int value){//thêm node có giá trị value
        if (root == NULL) //nếu cây rỗng
                return newNode(value); //đây chính là node gốc
        if (value < root ->data) //nếu điều này xảy ra
                root ->left  = insert(root ->left, value);//ta thêm vào cây con trái
        else //trường hợp value>node->data
                root ->right = insert(root ->right, value); //ta thêm vào cây con phải
        return root;//trả lại node
}
node *BST:: minValueNode(node* p){//trả lại node có giá trị nhỏ nhất trên cây gốc p
        node *current = p;//current trỏ đến p
        while (current->left != NULL)//quay trái liên tục
                current = current->left;
        return current; //đây chính là node trái nhất của cây gốc p
}
node*BST:: deleteNode(node* root, int value){//loại bỏ node có giá trị value
        if (root == NULL) //nếu cây  rỗng
                return root;//không có gì để loại bỏ
        if (value < root->data) //nếu node cần loại có giá trị nhỏ hơn node gốc
                root->left = deleteNode(root->left, value); //sang cây con trái ta loại
        else if (value > root->data) //nếu node cần loại có giá trị lớn hơn node gốc
                root->right = deleteNode(root->right, value); //sang cây con phải ta loại

        else { //nếu tìm thấy node có giá trị value
                if (root->left == NULL){ //nếu node cần loại chỉ có cây con phải
                        node *temp = root->right;
                        free(root);
                        return temp;
                }
                else if (root->right == NULL)  {//nếu node cần loại chỉ có cây con trái
                        node *temp = root->left;
                        free(root);
                        return temp;
                }
                // trường hợp node có cả hai cây con
                //ta lấy node trái nhất của cây con phải
                node* temp = minValueNode(root->right);
```

```
            root->data = temp->data; //thay thế nội dung node cần loại bỏ
            root->right = deleteNode(root->right, temp->data);//loại bỏ node thay thế
        }
        return root;
}
void BST::Function(void) {
        node *tmp; int value, choice;
        do {
            cout<<"\n CÂY NHỊ PHÂN TÌM KIẾM";
            cout<<"\n 1. Thêm node vào cây BST";
            cout<<"\n 2. Loại node trên cây BST ";
            cout<<"\n 3. Tìm node trên cây BST ";
            cout<<"\n 4. Duyệt theo thứ tự trước";
            cout<<"\n 5. Duyệt theo thứ tự giữa ";
            cout<<"\n 6. Duyệt theo thứ tự sau ";
            cout<<"\n 0. Thoát ";
            cout<<"\n Đưa vào lựa chọn:";cin>>choice;
            switch(choice) {
                case 1:
                    cout<<"\n Giá trị node cần thêm:"; cin>> value;
                    root = insert(root, value);break;
                case 2:
                    cout<<"\n Giá trị node cần loại:"; cin>> value;
                    root = deleteNode(root, value);break;
                case 3:
                    cout<<"\n Giá trị node cần tìm:"; cin>> value;
                    tmp = search(root, value);
                    if (tmp==NULL) cout<<"\n Node không có trên cây";
                    else cout<<"\n Node "<< tmp->value<<"có trên cây";
                    break;
                case 4:  preorder(root); break;
                case 5: inorder(root); break;
                case 6: postorder(root); break;
                default:
                    if(choice!=0) cout<<"\n Lua chon sai";    break;
            }
            cout<<"\n Duyệt trước:"; preorder(root);
```

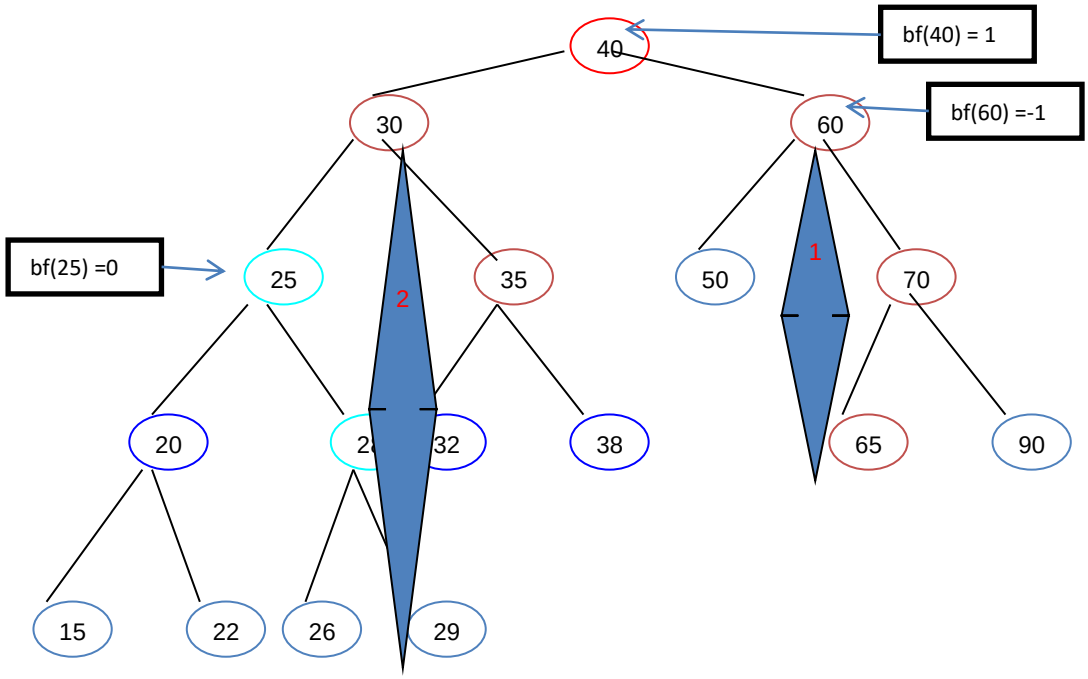
```
            cout<<"\n Duyệt sau:"; inorder(root);
            cout<<"\n Duyệt giữa:"; postorder(root);
        }while(choice!=0);
}
int main(){
        BST X; X.Function();
}
```

5.6. Cây nhị phân tìm kiếm cân bằng

Tìm kiếm trên cây BST không cải thiện được nhiều so với cây nhị phân thông thường. Độ phức tạp thuật toán tìm kiếm node trên cây BTS trong trường hợp tốt nhất là log(N), trong trường hợp xấu nhất là O(N), với N là số node trên cây. Trường hợp xấu nhất xảy ra khi cây nhị phân tìm kiếm là cây lệch trái hoặc cây lệnh phải. Trường hợp tốt nhất xảy ra khi cây nhị phân tìm kiếm có độ cao là $Log_2(N)$. Giảm chiều sâu (độ cao) của các cây nhị phân tìm kiếm để cải thiện thời gian tìm kiếm node trên cây là mục tiêu chính của cây nhị phân tìm kiếm cân bằng.

5.6.1. Định nghĩa cây nhị phân tìm kiếm cân bằng

Cây tìm nhị phân tìm kiếm cân bằng có tính chất độ cao của cây con bên trái và độ cao cây con bên phải luôn lệch nhau không quá 1. Hình 4.18 là một cây nhị phân tìm kiếm tự cân bằng (AVL).



Hình 4.18. Ví dụ về cây AVL

Gọi *lh(p) , rh(p)* là chiều sâu của cây con phải và chiều sâu của cây con trái. Khi đó, *bf(p) = lh(p) – rh(p)* được gọi là chỉ số cân bằng của node *p*. Dựa vào chỉ số cân bằng của node, ta có thể xác định một cây con gốc p mất cân bằng các trường hợp sau:

Trường hợp 1. *lh(p) - rh(p)> 1* ta nói node p lệch bên trái.

Trường hợp 2. *rh(p) - lh(p)>1* ta nói node p lệch bên phải.

5.6.2. Biểu diễn cây nhị phân tìm kiếm cân bằng

Biểu diễn cây nhị phân tìm kiếm cân bằng cũng giống như biểu diễn cây nhị phân tìm kiếm thông thường. Tuy nhiên, để thuận tiện trong việc tính toán chỉ số cân bằng của các node trên cây ta đưa thêm thông tin độ cao mỗi node. Cây AVL được biểu diễn như sau:

```
struct node {//định nghĩa cấu trúc node
        int data;//thành phần dữ liệu của node
        int height; //độ cao của node
        struct node *left;//thành phần con trỏ cây con trái
        struct node *right; //thành phần con trỏ cây con phải
} *AVL_Tree; //định nghĩa cây AVL
```

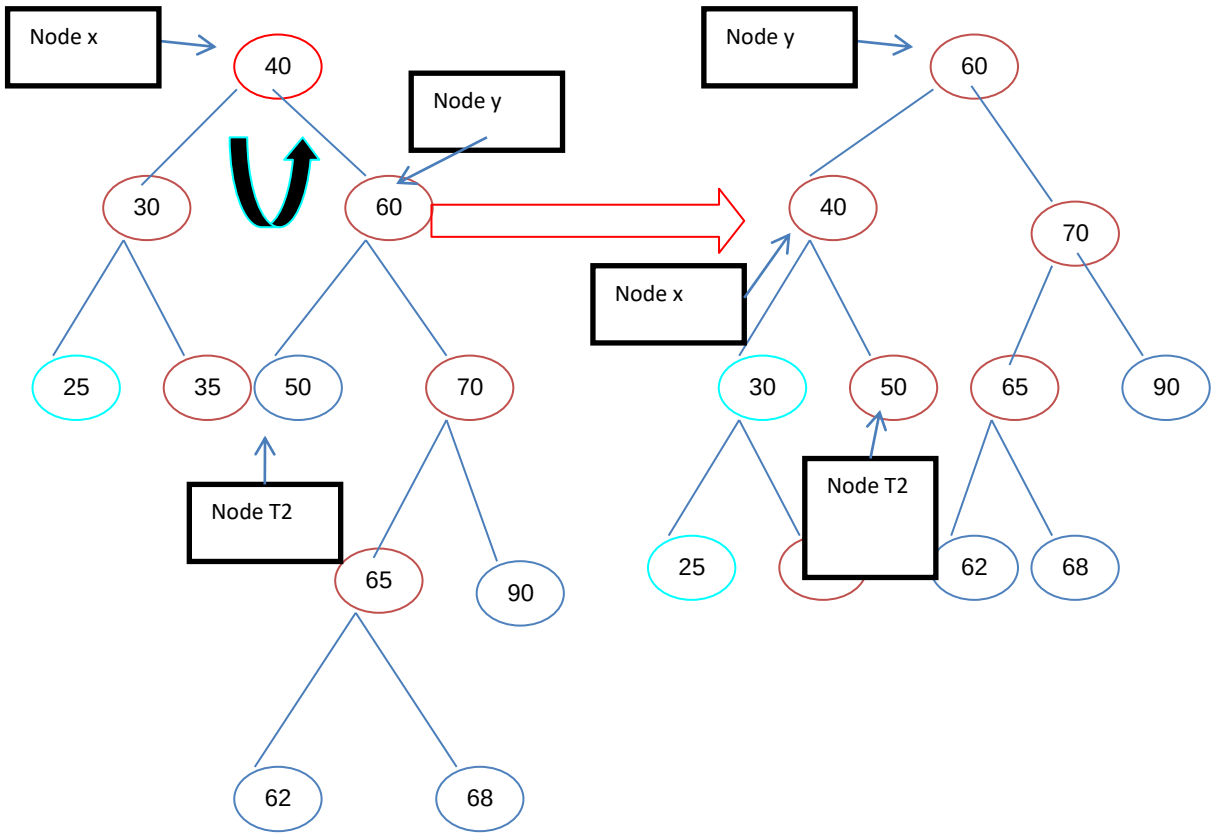
5.6.3. Các thao tác trên cây nhị phân tìm kiếm cân bằng

Khác với cây nhị phân tìm kiếm thông thường, khi thêm node vào cây nhị phân tìm kiếm cân bằng ta vẫn nhận được một cây nhị phân tìm kiếm cân bằng. Khi loại bỏ node khỏi cây nhị phân tìm kiếm cân bằng. Hai thao tác thêm node và loại bỏ node luôn duy trì độ cao của cây luôn là $\log_2(N)$, với N là số node trên cây. Để bảo toàn được hai phép thêm và loại bỏ node, cây AVL trang bị hai phép xoay cây bên trái và xoay cây bên phải để giảm độ cao cây con phải và giảm độ cao cây con trái. Lớp các thao tác cụ thể trên cây AVL khác với lớp các thao tác trên cây nhị phân tìm kiếm các thao tác: xoay trái cây AVL, xoay phải cây nhị phân AVL, thêm node vào cây AVL, loại bỏ node trên cây AVL. Lớp các thao tác trên cây AVL được thể hiện như dưới đây:

```
class AVL_Tree { //biểu diễn lớp cây AVL_Tree
  public:
        node *root; //đây là gốc của cây
        AVL_Tree(void){ //constructor của lớp
                root = NULL;//lúc đầu đặt là NULL
        }
        int  max(int a, int b); // tìm số lớn nhất trong hai số
        node* newNode(int value); //tạo node rời rạc có giá trị value
        int height(node *N); //lấy độ cao node N
        node* search(node* root, int value);//tìm node có giá trị value
```

```
        node * rightRotate(node *y); //phép quay phải tại node y
        node * leftRotate(node *x);//quay trái tại node x
        int  getBalance(node *N);//lấy chỉ số cân bằng của node
        node* insert(node* node, int data);//thêm node vào cây AVL
        node* deleteNode(node* root, int key);//loại node trên cây AVL
        void preOrder(node *root){//duyệt theo thứ tự trước
        void AVL_Tree ::inOrder(node *root); //duyệt theo thứ tự giữa
        void AVL_Tree ::postOrder(node *root); //duyệt theo thứ tự sau
        node * AVL_Tree ::minValueNode(node* root);//tìm node có giá trị nhỏ nhất
        void AVL_Tree ::Function(void); //hàm điều khiển các thao tác
};
```

Thao tác xoay trái cây AVL: khi ta thêm hoặc loại bỏ node trên cây AVL có thể làm cây mất cân bằng. Phép xoay trái trên cây AVL nhằm hạ độ cao cây con phải. Ví dụ ta cần xoay trái tại node 40 trên cây Hình 4.19.

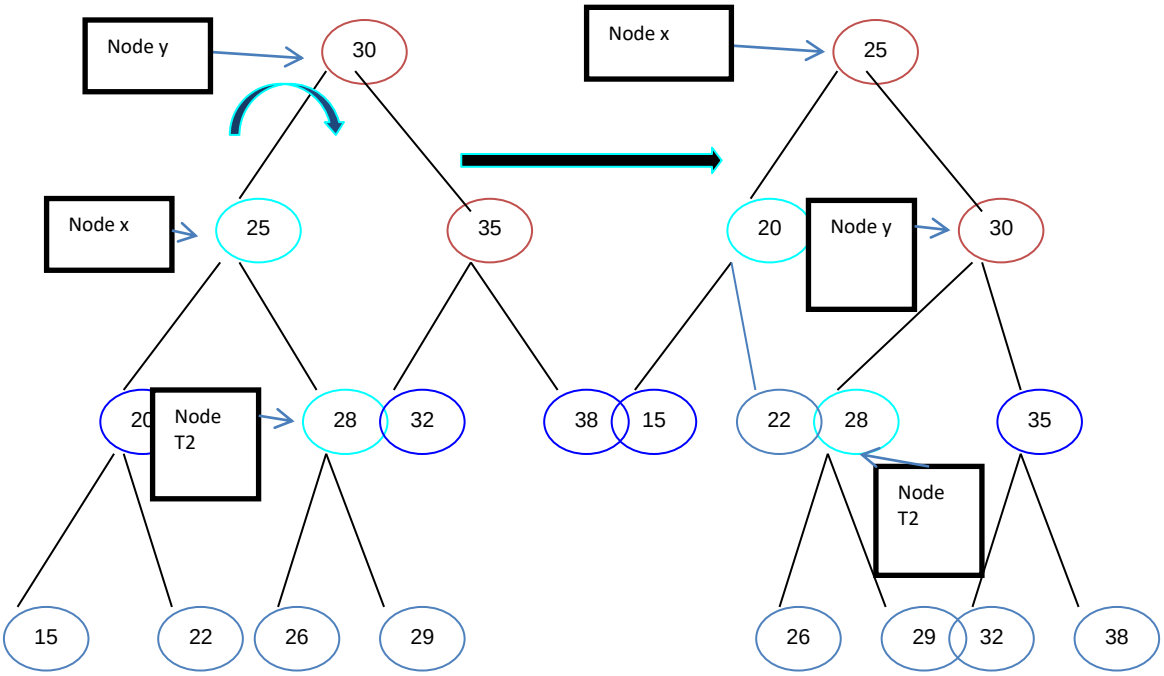


Hình 4.19. Thao tác xoay trái trên cây AVL

```
node * AVL_Tree ::leftRotate(node *x){//quay trái tại node x
        node *y = x->right; //y là node con phải của x
```

```
        node *T2 = y->left; //T2 là node con trái của y
        // phép quay được thực hiện như dưới đây
        y->left = x; //node trái của y bây giờ là x
        x->right = T2; //node phải của x bây giờ là T2
        // cập nhật lại độ cao
        x->height = max(height(x->left), height(x->right))+1;
        y->height = max(height(y->left), height(y->right))+1;
        return y;//node gốc mới là y
}
```

Thao tác xoay phải cây AVL: khi ta thêm hoặc loại bỏ node trên cây AVL có thể làm cây mất cân bằng. Phép xoay phải trên cây AVL nhằm hạ độ cao cây con trái. Ví dụ ta cần xoay phải tại node 40 trên cây Hình 4.20.



Hình 4.20. Phép xoay phải cây nhị phân.

```
node * AVL_Tree ::rightRotate(node *y){ //phép quay phải tại node y
        node *x = y->left; //x là node con trái của y
        node *T2 = x->right; //T2 là node con phải của x
        //phép quay được thực hiện như dưới đây
        x->right = y; //node con phải của x bây giờ là y
        y->left = T2; //node con trái của y bây giờ là T2
        // cập nhật lại độ cao
        y->height = max(height(y->left), height(y->right))+1;
        x->height = max(height(x->left), height(x->right))+1;
        return x; //trả lại gốc mới là x
```

}

Thao tác thêm node vào cây AVL: thuật toán thêm node w vào cây AVL được thực hiện như sau:

Bước 1. Thực hiện thêm node w vào cây tìm kiếm giống như cây thông thường.

Bước 2. Xuất phát từ node w, duyệt lên trên để tìm node mất cân bằng đầu tiên. Gọi z là node mất cân bằng đầu tiên, y là con của z và x là cháu của z .

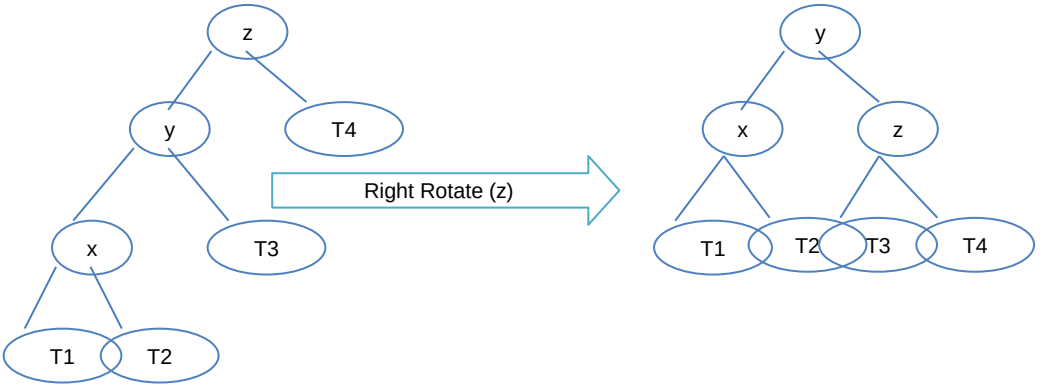
Bước 3. Cân bằng lại cây bằng các phép quay thích hợp tại cây con gốc z. Có 4 khả năng có thể xảy ra như sau:

a) Node y là node con trái của z và x là node con trái của y (left-left-case). Trường hợp này ta thực hiện phép soay phải tại node z (right rotation). Ví dụ của phép xoay left-left trên cây trong Hình 4.21.

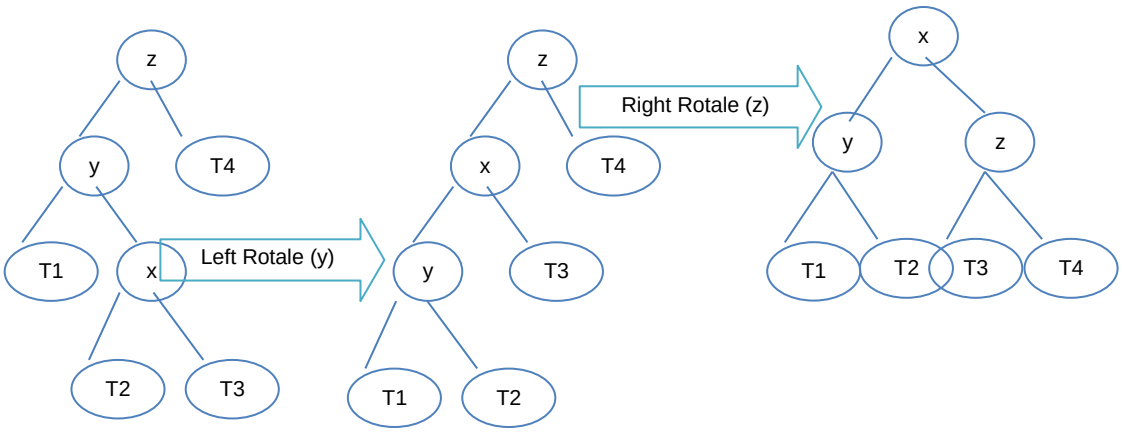
b) Node y là node con trái của z và x là node con phải của y (left-right-case). Trường hợp này ta thực hiện phép soay trái tại node y sau đó soay phải tại node z. Ví dụ của phép xoay left-right trên cây trong Hình 4.22.

c) Node y là node con phải của z và x là node con phải của y (right-right-case). Trường hợp này ta thực hiện phép soay trái tại node z (left rotation). Ví dụ của phép xoay right-right trên cây trong Hình 4.23.

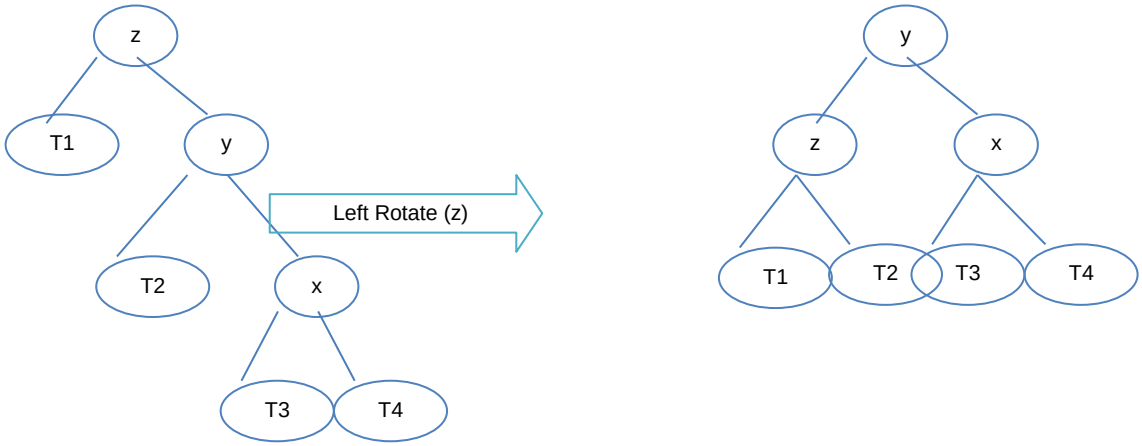
d) Node y là node con phải của z và x là node con trái của y (right-left-case). Trường hợp này ta thực hiện phép soay trái trái tại node y sau đó soay phải tại node z. Ví dụ của phép xoay right-left trên cây trong Hình 4.23.



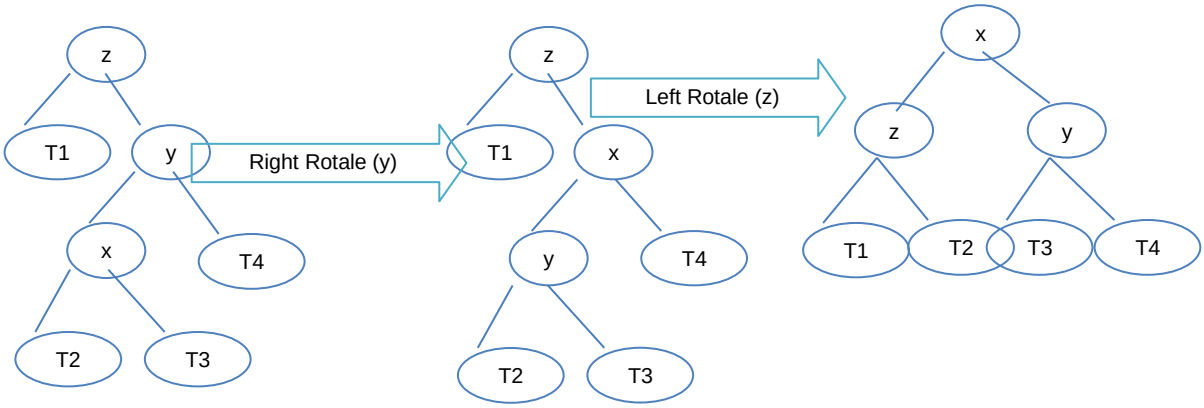
Hình 4.21. Trường hợp left-left



Hình 4.22. Trường hợp left-right



Hình 4.23. Trường hợp right-right



Hình 4.24. Trường hợp right-left

```
node* AVL_Tree ::insert(node* node, int data){//thêm node vào cây AVL
        //Bước 1. Thực hiện thêm node giống như cây tìm kiếm
        if (node == NULL)
                return(newNode(data));
        if (data < node->data)
                node->left  = insert(node->left, data);
```

```
        else
                node->right = insert(node->right, data);
        // Bước 2. Cập nhật độ cao của cây
        node->height = max(height(node->left), height(node->right)) + 1;
        //Bước 3. Lấy chỉ số cân bằng của cây
        int balance = getBalance(node);
        // 4 trường hợp làm cây mất cân bằng được xem xét
        //Trường hợp a: Left-Left Case
        if (balance > 1 && data < node->left->data) //ta quay phải
                return rightRotate(node);
        // Trường hợp b:  Right Right Case
        if (balance < -1 && data > node->right->data)//ta quay trái
                return leftRotate(node);
        // Trường hợp c: Left Right Case
        if (balance > 1 && data > node->left->data){
                node->left =  leftRotate(node->left);//quay trái trước
                return rightRotate(node); // quay phải sau
        }
        // Trường hợp d: Right Left Case
        if (balance < -1 && data < node->right->data){
                node->right = rightRotate(node->right);//quay phải trước
                return leftRotate(node);//quay trái sau
        }
        return node;//trả lại node
}
```

Thao tác loại bỏ node trên cây AVL: thuật toán loại node w trên cây AVL được thực hiện như sau:

Bước 1. Thực hiện loại bỏ node w vào cây tìm kiếm giống như cây thông thường.

Bước 2. Xuất phát từ node w, duyệt lên trên để tìm node mất cân bằng đầu tiên. Gọi z là node mất cân bằng đầu tiên, y là con của z và x là cháu của z .

Bước 3. Cân bằng lại cây bằng các phép quay thích hợp tại cây con gốc z. Có 4 khả năng có thể xảy ra như sau:

a) Node y là node con trái của z và x là node con trái của y (left-left-case). Trường hợp này ta thực hiện phép soay phải tại node z (right rotation). Ví dụ của phép xoay left-left trên cây trong Hình 4.21.

b) Node y là node con trái của z và x là node con phải của y (left-right-case). Trường hợp này ta thực hiện phép soay trái tại node y sau đó soay phải tại node z. Ví dụ của phép xoay left-right trên cây trong Hình 4.22.
c) Node y là node con phải của z và x là node con phải của y (right-right-case). Trường hợp này ta thực hiện phép soay trái tại node z (left rotation). Ví dụ của phép xoay right-right trên cây trong Hình 4.23.
d) Node y là node con phải của z và x là node con trái của y (right-left-case). Trường hợp này ta thực hiện phép soay trái trái tại node y sau đó soay phải tại node z. Ví dụ của phép xoay right-left trên cây trong Hình 4.23.

```
node* AVL_Tree ::deleteNode(node* root, int key){//loại node trên cây AVL
    //Bước 1: Thực hiện loại node giống như cây tìm kiếm
    if (root == NULL)//nếu cây rỗng
        return root;
    if ( key < root->data )//nếu điều này xảy ra
        root->left = deleteNode(root->left, key);//tìm node sang bên trái
    else if( key > root->data )//nếu điều này xảy ra
        root->right = deleteNode(root->right, key); //tìm node sang bên phải
    else { //nếu tìm thấy đúng node giá trị key
        // nếu node là lá hoặc có một cây con
        if( (root->left == NULL) || (root->right == NULL) ){
            node *temp = root->left ? root->left : root->right;
            if(temp == NULL){ //lấy temp = NULL
                temp = root;
                root = NULL;
            }
            else // nếu node chỉ có một cây con
                *root = *temp; //thay thế root bằng temp
            free(temp);//giải phóng temp
        }
        else { //trường hợp node có hai cây con
            //lấy node trái nhất của cây con phải
            node* temp = minValueNode(root->right);
            root->data = temp->data;//thay nội dung node hiện tại
            // sau đó loại bỏ node temp
            root->right = deleteNode(root->right, temp->data);
        }
```

```
        }
        //Bước2. Tìm node làm cây mất bằng cây:
        if (root == NULL)//nếu cây rỗng
                return root; //không cần cân bằng lại
        // Cập nhật độ cao node
        root->height = max(height(root->left), height(root->right)) + 1;
        // Bước 3.  Cân bằng cây
        int balance = getBalance(root);
        // 4 trường hợp làm cây mất cân bằng được xem xét:
        //Trường hợp a. Cây mất cân bằng trái (Left Left Case)
        if (balance > 1 && getBalance(root->left) >= 0)
                return rightRotate(root); //ta quay phải
        // Trường hợp b. Left Right Case
        if (balance > 1 && getBalance(root->left) < 0) {
                root->left =  leftRotate(root->left);//ta quay trái trước
                return rightRotate(root); //rồi quay phải sau
        }
        // Trường hợp c. Right Right Case
        if (balance < -1 && getBalance(root->right) <= 0)
                return leftRotate(root); //ta quay trái
        // Trường hợp d. Right Left Case
        if (balance < -1 && getBalance(root->right) > 0) {
                root->right = rightRotate(root->right);//quay phải trước
                return leftRotate(root); //quay trái sau
        }
        return root;
}
```

5.6.4. Chương trình cài đặt cây nhị phân tìm kiếm cân bằng

```
//cài đặt cây AVL
#include <iostream>
using namespace std;
struct node {//định nghĩa cấu trúc node
        int data;//thành phần dữ liệu của node
        struct node *left;//thành phần con trỏ cây con trái
        struct node *right; //thành phần con trỏ cây con phải
        int height; //độ cao của node
};
```

```
class AVL_Tree { //biểu diễn lớp cây AVL_Tree
  public:
        node *root; //đây là gốc của cây
        AVL_Tree(void){ //constructor của lớp
                root = NULL;//lúc đầu đặt là NULL
        }
        int  max(int a, int b); // tìm số lớn nhất trong hai số
        node* newNode(int value); //tạo node rời rạc có giá trị value
        int height(node *N); //lấy độ cao node N
        node* search(node* root, int value);//tìm node có giá trị value
        node * rightRotate(node *y); //phép quay phải tại node y
        node * leftRotate(node *x);//quay trái tại node x
        int  getBalance(node *N);//lấy chỉ số cân bằng của node
        node* insert(node* node, int data);//thêm node vào cây AVL
        node* deleteNode(node* root, int key);//loại node trên cây AVL
        void preOrder(node *root){//duyệt theo thứ tự trước
        void AVL_Tree ::inOrder(node *root); //duyệt theo thứ tự giữa
        void AVL_Tree ::postOrder(node *root); //duyệt theo thứ tự sau
        node * AVL_Tree ::minValueNode(node* root);//tìm node có giá trị nhỏ nhất
        void AVL_Tree ::Function(void){ //hàm điều khiển các thao tác
};
int AVL_Tree ::max(int a, int b){// tìm số lớn nhất trong hai số
        return (a > b)? a : b;
}
int AVL_Tree ::height(node *N){ //lấy độ cao node N
        if (N == NULL)
                return 0;
        return N->height;
}
node* AVL_Tree ::newNode(int value){ //tạo node rời rạc có giá trị value
        node* p = new node;
        p->data   = value;
        p->left   = NULL;
        p->right  = NULL;
        p->height = 1;
        return(p);
}
```

```
node* AVL_Tree ::search(node* root, int value){//tìm node có giá trị value
        if (root == NULL || root->data == value)
                return root;
        if (root->data < value)
                return search(root->right, value);
        return search(root->left, value);
}
node * AVL_Tree ::rightRotate(node *y){ //phép quay phải tại node y
        node *x = y->left; //x là node con trái của y
        node *T2 = x->right; //T2 là node con phải của x
        //phép quay được thực hiện như dưới đây
        x->right = y; //node con phải của x bây giờ là y
        y->left = T2; //node con trái của y bây giờ là T2
        // cập nhật lại độ cao
        y->height = max(height(y->left), height(y->right))+1;
        x->height = max(height(x->left), height(x->right))+1;
        return x; //trả lại gốc mới là x
}
node * AVL_Tree ::leftRotate(node *x){//quay trái tại node x
        node *y = x->right; //y là node con phải của x
        node *T2 = y->left; //T2 là node con trái của y
        // phép quay được thực hiện như dưới đây
        y->left = x; //node trái của y bây giờ là x
        x->right = T2; //node phải của x bây giờ là T2
        // cập nhật lại độ cao
        x->height = max(height(x->left), height(x->right))+1;
        y->height = max(height(y->left), height(y->right))+1;
        return y;//node gốc mới là y
}
int AVL_Tree ::getBalance(node *N){//lấy chỉ số cân bằng của node
        if (N == NULL) //nếu N là rỗng
                return 0; //chỉ số cân bằng là 0
        return height(N->left) - height(N->right); //hiệu độ cao hai cây con
}
node* AVL_Tree ::insert(node* node, int data){//thêm node vào cây AVL
        //Bước 1. Thực hiện thêm node giống như cây tìm kiếm
        if (node == NULL)
```

```
            return(newNode(data));
        if (data < node->data)
            node->left  = insert(node->left, data);
        else
            node->right = insert(node->right, data);
        // Bước 2. Cập nhật độ cao của cây
        node->height = max(height(node->left), height(node->right)) + 1;
        //Bước 3. Lấy chỉ số cân bằng của cây
        int balance = getBalance(node);
        // 4 trường hợp làm cây mất cân bằng được xem xét
        //Trường hợp a: Left-Left Case
        if (balance > 1 && data < node->left->data) //ta quay phải
            return rightRotate(node);
        // Trường hợp b:  Right Right Case
        if (balance < -1 && data > node->right->data)//ta quay trái
            return leftRotate(node);
        // Trường hợp c: Left Right Case
        if (balance > 1 && data > node->left->data){
            node->left =  leftRotate(node->left);//quay trái trước
            return rightRotate(node); // quay phải sau
        }
        // Trường hợp d: Right Left Case
        if (balance < -1 && data < node->right->data){
            node->right = rightRotate(node->right);//quay phải trước
            return leftRotate(node);//quay trái sau
        }
        return node;//trả lại node
}
void AVL_Tree ::preOrder(node *root){//duyệt theo thứ tự trước
        if(root != NULL) {
            printf("%3d ", root->data);
            preOrder(root->left);
            preOrder(root->right);
        }
}
void AVL_Tree ::inOrder(node *root){ //duyệt theo thứ tự giữa
        if(root != NULL) {
```

```
                inOrder(root->left);
                printf("%3d ", root->data);
                inOrder(root->right);
        }
}
void AVL_Tree ::postOrder(node *root){ //duyệt theo thứ tự sau
        if(root != NULL) {
                postOrder(root->left);
                postOrder(root->right);
                printf("%3d ", root->data);
        }
}
node * AVL_Tree ::minValueNode(node* root){//tìm node có giá trị nhỏ nhất
        node* current = root;
        while (current->left != NULL)//quay trái liên tục
                current = current->left;
        return current;//node có giá trị nhỏ nhất
 }
node* AVL_Tree ::deleteNode(node* root, int key){//loại node trên cây AVL
        //Bước 1: Thực hiện loại node giống như cây tìm kiếm
        if (root == NULL)//nếu cây rỗng
                return root;
        if ( key < root->data )//nếu điều này xảy ra
                root->left = deleteNode(root->left, key);//tìm node sang bên trái
        else if( key > root->data )//nếu điều này xảy ra
                root->right = deleteNode(root->right, key); //tìm node sang bên phải
        else { //nếu tìm thấy đúng node giá trị key
                // nếu node là lá hoặc có một cây con
                if( (root->left == NULL) || (root->right == NULL) ){
                        node *temp = root->left ? root->left : root->right;
                        if(temp == NULL){ //lấy temp = NULL
                                temp = root;
                                root = NULL;
                        }
                        else // nếu node chỉ có một cây con
                                *root = *temp; //thay thế root bằng temp
                        free(temp);//giải phóng temp
```

```
            }
            else { //trường hợp node có hai cây con
                //lấy node trái nhất của cây con phải
                node* temp = minValueNode(root->right);
                root->data = temp->data;//thay nội dung node hiện tại
                // sau đó loại bỏ node temp
                root->right = deleteNode(root->right, temp->data);
            }
        }
        //Bước2. Tìm node làm cây mất bằng cây:
        if (root == NULL)//nếu cây rỗng
            return root; //không cần cân bằng lại
        // Cập nhật độ cao node
        root->height = max(height(root->left), height(root->right)) + 1;
        // Bước 3.  Cân bằng cây
        int balance = getBalance(root);
        // 4 trường hợp làm cây mất cân bằng được xem xét:
        //Trường hợp a. Cây mất cân bằng trái (Left Left Case)
        if (balance > 1 && getBalance(root->left) >= 0)
            return rightRotate(root); //ta quay phải
        // Trường hợp b. Left Right Case
        if (balance > 1 && getBalance(root->left) < 0) {
            root->left =  leftRotate(root->left);//ta quay trái trước
            return rightRotate(root); //rồi quay phải sau
        }
        // Trường hợp c. Right Right Case
        if (balance < -1 && getBalance(root->right) <= 0)
            return leftRotate(root); //ta quay trái
        // Trường hợp d. Right Left Case
        if (balance < -1 && getBalance(root->right) > 0) {
            root->right = rightRotate(root->right);//quay phải trước
            return leftRotate(root); //quay trái sau
        }
        return root;
}
void AVL_Tree ::Function(void){ //hàm điều khiển các thao tác
        //struct node *root = NULL;
```

```
node *tmp;
int data, choice;
do {
        printf("\n Cac thao tac tren cay AVL");
        printf("\n 1. Them node vao cay AVL");
        printf("\n 2. Loai node tren cay AVL");
        printf("\n 3. Tim node tren cay AVL");
        printf("\n 4. Duyet theo thu tu truoc");
        printf("\n 5. Duyet theo thu tu giua");
        printf("\n 6. Duyet theo thu tu sau");
        printf("\n 0. Thoat khoi chuong trinh");
        printf("\n Dua vao lua chon:");scanf("%d",&choice);
        switch(choice) {
                case 1:
                        printf("\n Gia tri node can them:"); scanf("%d",&data);
                        root = insert(root, data);break;
                case 2:
                        printf("\n Gia tri node can loai:"); scanf("%d",&data);
                        root = deleteNode(root, data);break;
                case 3:
                        printf("\n Gia tri node can tim:"); scanf("%d",&data);
                        tmp = search(root, data);
                        if (tmp==NULL) printf("\n Node khong co tren cay");
                        else printf("\n Node %d co tren cay", tmp->data);
                        break;
                case 4:
                        preOrder(root); break;
                case 5:
                        inOrder(root); break;
                case 6:
                        postOrder(root); break;
                default:
                        if(choice!=0) printf("\n Lua chon sai");
                        break;
        }
        printf("\n Duyet truoc:"); preOrder(root);
        printf("\n Duyet giua:"); inOrder(root);
```

```
            printf("\n Duyet sau:"); postOrder(root);
        } while(choice!=0);
}
int main(void){
        AVL_Tree X;
        X.Function();
}
```

BÀI TẬP

BÀI 1. DUYỆT CÂY 1

Cho phép duyệt cây nhị phân Inorder và Preorder, hãy đưa ra kết quả phép duyệt Postorder của cây nhị phân. Ví dụ với cây nhị phân có các phép duyệt cây nhị phân của cây dưới đây:

Inorder : 4 2 5 1 3 6
Preorder: : 1 2 4 5 3 6
Postorder : 4 5 2 6 3 1

Input:

- Dòng đầu tiên đưa vào số lượng test T.
- Những dòng tiếp theo đưa vào các bộ test. Mỗi bộ test gồm 3 dòng: dòng đầu tiên đưa vào số N là số lượng node; dòng tiếp theo đưa vào N số theo phép duyệt Inorder; dòng cuối cùng đưa vào N số là kết quả của phép duyệt Preorder; các số được viết cách nhau một vài khoảng trống.
- T, N, node thỏa mãn ràng buộc: $1 \le T \le 100$; $1 \le N \le 1000$; $1 \le$ giá trị node $\le 10^4$;

Output:

- Đưa ra kết quả phép duyệt Postorder theo từng dòng.

Ví dụ:

Input	Output
1 6 4 2 5 1 3 6 1 2 4 5 3 6	4 5 2 6 3 1

BÀI 2. DUYỆT CÂY 2

Cho mảng A[] gồm N node là biểu diễn phép duyệt theo thứ tự giữa (Preorder) của cây nhị phân tìm kiếm. Nhiệm vụ của bạn là đưa ra phép duyệt theo thứ tự sau của cây nhị phân tìm kiếm.

Input:

- Dòng đầu tiên đưa vào số lượng test T.
- Những dòng tiếp theo đưa vào các bộ test. Mỗi bộ test gồm 2 dòng: dòng đầu tiên đưa vào số N là số lượng node; dòng tiếp theo đưa vào N số A[i]; các số được viết cách nhau một vài khoảng trống.
- T, N, node thỏa mãn ràng buộc: $1 \le T \le 100$; $1 \le N \le 10^3$; $1 \le A[i] \le 10^4$;

Output:

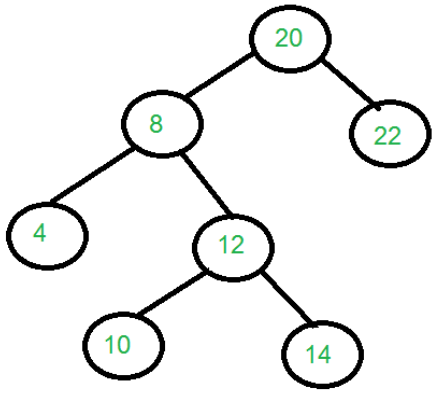
- Đưa ra kết quả phép duyệt Postorder theo từng dòng.

Ví dụ:

Input	Output
2 5 40 30 35 80 100 8 40 30 32 35 80 90 100 120	35 30 100 80 40 35 32 30 120 100 90 80 40

BÀI 3. DUYỆT CÂY 3

Cho cây nhị phân, nhiệm vụ của bạn là duyệt cây theo Level-order. Phép duyệt level-order trên cây là phép thăm node theo từng mức của cây. Ví dụ với cây dưới đây sẽ cho ta kết quả của phép duyệt level-order: 20 8 22 4 12 10 14.



Input:

- Dòng đầu tiên đưa vào số lượng test T.
- Những dòng tiếp theo đưa vào các bộ test. Mỗi bộ test gồm 2 dòng: dòng đầu tiên đưa vào số N là số lượng cạnh của cây; dòng tiếp theo đưa vào N bộ ba (u, v, x), trong đó u là node cha, v là node con, x= R nếu v là con phải, x=L nếu v là con trái; u, v, x được viết cách nhau một vài khoảng trống.
- T, N, u, v, thỏa mãn ràng buộc: $1 \le T \le 100$; $1 \le N \le 10^3$; $1 \le u, v \le 10^4$;

Output:

- Đưa ra kết quả phép duyệt level-order theo từng dòng.

Ví dụ:

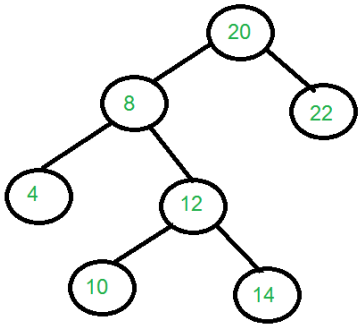
Input	Output
2 2 1 2 R 1 3 L 4 10 20 L 10 30 R 20 40 L 20 60 R	1 3 2 10 0 30 40 60

BÀI 4. DUYỆT CÂY 4

Cho hai mảng là phép duyệt Inorder và Level-order, nhiệm vụ của bạn là xây dựng cây nhị phân và đưa ra kết quả phép duyệt Postorder. Level-order là phép duyệt theo từng mức của cây. Ví dụ như cây dưới đây ta có phép Inorder và Level-order như dưới đây:

Inorder : 4 8 10 12 14 20 22

Level order: 20 8 22 4 12 10 14



Input:

- Dòng đầu tiên đưa vào số lượng test T.
- Những dòng tiếp theo đưa vào các bộ test. Mỗi bộ test gồm 3 dòng: dòng đầu tiên đưa vào số N là số lượng node; dòng tiếp theo đưa vào N số là phép duyệt Inorder; dòng cuối cùng đưa vào N số là phép duyệt Level-order; các số được viết cách nhau một vài khoảng trống.
- T, N, node thỏa mãn ràng buộc: $1 \le T \le 100$; $1 \le N \le 10^3$; $1 \le A[i] \le 10^4$;

Output:

- Đưa ra kết quả phép duyệt Postorder theo từng dòng.

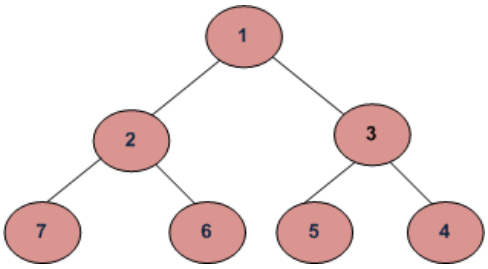
Ví dụ:

Input	Output
2 3	1 2 0 3 4 1 5 6 2 0

1 0 2 0 1 2 7 3 1 4 0 5 2 6 0 1 2 3 4 5 6	

BÀI 5. DUYỆT CÂY 5

Cho cây nhị phân, nhiệm vụ của bạn là duyệt cây theo xoắn ốc (spiral-order). Phép. Ví dụ với cây dưới đây sẽ cho ta kết quả của phép duyệt spiral-order: 1 2 3 4 5 6 7.



Input:

- Dòng đầu tiên đưa vào số lượng test T.
- Những dòng tiếp theo đưa vào các bộ test. Mỗi bộ test gồm 2 dòng: dòng đầu tiên đưa vào số N là số lượng cạnh của cây; dòng tiếp theo đưa vào N bộ ba (u, v, x), trong đó u là node cha, v là node con, x= R nếu v là con phải, x=L nếu v là con trái; u, v, x được viết cách nhau một vài khoảng trống.
- T, N, u, v, thỏa mãn ràng buộc: $1 \le T \le 100$; $1 \le N \le 10^3$; $1 \le u, v \le 10^4$;

Output:

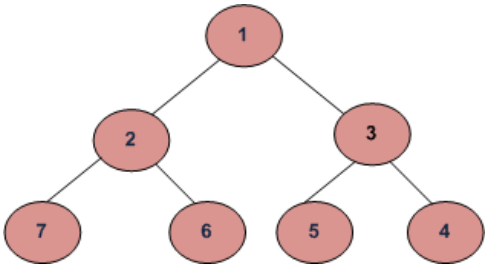
- Đưa ra kết quả phép duyệt level-order theo từng dòng.

Ví dụ:

Input	Output
2 2 1 2 R 1 3 L 4 10 20 L 10 30 R 20 40 L 20 60 R	1 3 2 10 0 30 60 40

BÀI 6. DUYỆT CÂY 6

Cho cây nhị phân, nhiệm vụ của bạn là duyệt cây theo mức đảo ngược (revese-level-order). Với cây dưới đây sẽ cho ta kết quả của phép duyệt theo mức đảo ngược là : 7 6 5 4 3 2 1.



Input:

- Dòng đầu tiên đưa vào số lượng test T.
- Những dòng tiếp theo đưa vào các bộ test. Mỗi bộ test gồm 2 dòng: dòng đầu tiên đưa vào số N là số lượng cạnh của cây; dòng tiếp theo đưa vào N bộ ba (u, v, x), trong đó u là node cha, v là node con, x= R nếu v là con phải, x=L nếu v là con trái; u, v, x được viết cách nhau một vài khoảng trống.
- T, N, u, v, thỏa mãn ràng buộc: $1 \le T \le 100$; $1 \le N \le 10^3$; $1 \le u, v \le 10^4$;

Output:

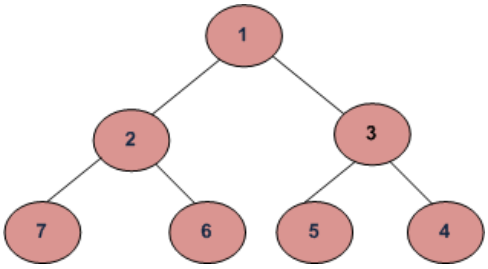
- Đưa ra kết quả phép duyệt revese-level-order theo từng dòng.

Ví dụ:

Input	Output
2 2 1 2 R 1 3 L 4 10 20 L 10 30 R 20 40 L 20 60 R	3 2 1 40 20 30 10

BÀI 7. KIỂM TRA NODE LÁ

Cho cây nhị phân, nhiệm vụ của bạn là kiểm tra xem tất cả các node lá của cây có cùng một mức hay không? Ví dụ với cây dưới đây sẽ cho ta kết quả là Yes.



Input:

- Dòng đầu tiên đưa vào số lượng test T.
- Những dòng tiếp theo đưa vào các bộ test. Mỗi bộ test gồm 2 dòng: dòng đầu tiên đưa vào số N là số lượng cạnh của cây; dòng tiếp theo đưa vào N bộ ba (u, v, x), trong đó u là node cha, v là node con, x= R nếu v là con phải, x=L nếu v là con trái; u, v, x được viết cách nhau một vài khoảng trống.
- T, N, u, v, thỏa mãn ràng buộc: $1 \le T \le 100$; $1 \le N \le 10^3$; $1 \le u, v \le 10^4$;

Output:

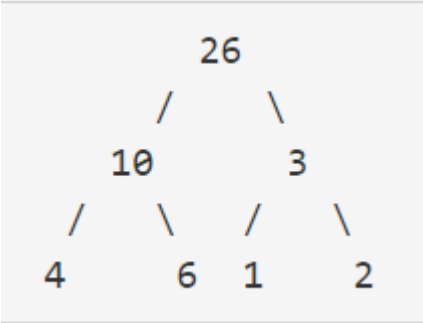
- Đưa ra kết quả mỗi test theo từng dòng.

Ví dụ:

Input	Output
2 2 1 2 R 1 3 L 4 10 20 L 10 30 R 20 40 L 20 60 R	1 0

BÀI 8. CÂY NHỊ PHÂN TỔNG

Cho cây nhị phân, nhiệm vụ của bạn là kiểm tra xem cây nhị phân có phải là một cây tổng hay không? Một cây nhị phân được gọi là cây tổng nếu tổng các node con của node trung gian bằng giá trị node cha. Node không có node con trái hoặc phải được hiểu là có giá trị 0. Ví dụ dưới đây là một cây tổng



Input:

- Dòng đầu tiên đưa vào số lượng test T.
- Những dòng tiếp theo đưa vào các bộ test. Mỗi bộ test gồm 2 dòng: dòng đầu tiên đưa vào số N là số lượng cạnh của cây; dòng tiếp theo đưa vào N bộ ba (u, v, x), trong đó u là node cha, v là node con, x= R nếu v là con phải, x=L nếu v là con trái; u, v, x được viết cách nhau một vài khoảng trống.
- T, N, u, v, thỏa mãn ràng buộc: $1 \le T \le 100$; $1 \le N \le 10^3$; $1 \le u, v \le 10^4$;

Output:

- Đưa ra kết quả mỗi test theo từng dòng.

Ví dụ:

Input	Output
2 2 3 1 L 3 2 R 4 10 20 L 10 30 R 20 40 L 20 60 R	1 0

BÀI 9. CÂY NHỊ PHÂN HOÀN HẢO

Cho cây nhị phân, nhiệm vụ của bạn là kiểm tra xem cây nhị phân có phải là một cây hoàn hảo hay không (perfect tree)? Một cây nhị phân được gọi là cây hoàn hảo nếu tất cả các node trung gian của nó đều có hai node con và tất cả các node lá đều có cùng một mức.

Input:

- Dòng đầu tiên đưa vào số lượng test T.
- Những dòng tiếp theo đưa vào các bộ test. Mỗi bộ test gồm 2 dòng: dòng đầu tiên đưa vào số N là số lượng cạnh của cây; dòng tiếp theo đưa vào N bộ ba (u, v, x), trong đó u là node cha, v là node con, x= R nếu v là con phải, x=L nếu v là con trái; u, v, x được viết cách nhau một vài khoảng trống.
- T, N, u, v, thỏa mãn ràng buộc: $1 \le T \le 100$; $1 \le N \le 10^3$; $1 \le u, v \le 10^4$;

Output:

- Đưa ra kết quả mỗi test theo từng dòng.

Ví dụ:

Input	Output
3 6 10 20 L 10 30 R 20 40 L 20 50 R 30 60 L 30 70 R 2 18 15 L 18 30 R 5 1 2 L 2 4 R 1 3 R 3 5 L 3 6 R	Yes Yes No

BÀI 10. CÂY NHỊ PHÂN ĐỦ

Cho cây nhị phân, nhiệm vụ của bạn là kiểm tra xem cây nhị phân có phải là một cây đủ hay không (full binary tree)? Một cây nhị phân được gọi là cây đủ nếu tất cả các node trung gian của nó đều có hai node con.

Input:

- Dòng đầu tiên đưa vào số lượng test T.
- Những dòng tiếp theo đưa vào các bộ test. Mỗi bộ test gồm 2 dòng: dòng đầu tiên đưa vào số N là số lượng cạnh của cây; dòng tiếp theo đưa vào N bộ ba (u, v, x), trong đó u là node cha, v là node con, x= R nếu v là con phải, x=L nếu v là con trái; u, v, x được viết cách nhau một vài khoảng trống.
- T, N, u, v, thỏa mãn ràng buộc: $1 \le T \le 100$; $1 \le N \le 10^3$; $1 \le u, v \le 10^4$;

Output:

- Đưa ra kết quả mỗi test theo từng dòng.

Ví dụ:

Input	Output
2 4 1 2 L 1 3 R 2 4 L 2 5 R 3 1 2 L 1 3 R 2 4 L	1 0

CHƯƠNG 6. ĐỒ THỊ (GRAPH)

Đồ thị là một cấu trúc dữ liệu rời rạc bao gồm các đỉnh và các cạnh nối các cặp đỉnh này. Với quan niệm như trên, một mạng máy tính được xem như một đồ thị có mỗi đỉnh là một máy tính, có các cạnh là một liên kết giữa các máy tính khác nhau trong mạng. Các mạch điện, các hệ thống giao thông, các mạng xã hội đều được xem xét như một đồ thị. Có thể nói đồ thị được ứng dụng rộng rãi trong nhiều lĩnh vực khác nhau của khoa học máy tính. Nội dung chính của chương này đề cập đến phương pháp biểu diễn và các thuật toán trên đồ thị.

6.1. Định nghĩa và khái niệm

Ta có thể phân chia đồ thị thành hai loại: đồ thị vô hướng (directed graph) và đồ thị có hướng (directed graph). Mỗi loại đồ thị lại được chia thành 3 loại: đơn đồ thị, đa đồ thị và giả đồ thị. Mỗi loại đồ thị có các thuật ngữ chung và những khái niệm riêng. Dưới đây là một số thuật ngữ cơ bản trên các loại đồ thị.

5.1.1. Một số thuật ngữ cơ bản trên đồ thị

Định nghĩa. Bộ đô $G =<V, E>$, trong đó $V = \{1, 2 , .., n\}$ là tập hợp hữu hạn được gọi là tập đỉnh, E là tập có thứ tự hoặc không có thứ tự các cặp đỉnh trong V được gọi là tập cạnh.

Đồ thị vô hướng. Đồ thị G=<V,E> được gọi là đồ thị vô hướng nếu các cạnh thuộc E là các cặp không tính đến thứ tự các đỉnh trong V.

Đơn đồ thị vô hướng. Đồ thị G=<V,E> được gọi là đơn đồ thị vô hướng nếu G là đồ thị vô hướng và giữa hai đỉnh bất kỳ thuộc V có nhiều nhất một cạnh nối.

Đa đồ thị vô hướng. Đồ thị G=<V,E> được gọi là đơn đồ thị vô hướng nếu là đồ thị vô hướng và tồn tại một cặp đỉnh trong V có nhiều hơn một cạnh nối. Cạnh $e_1 \in E$, $e_2 \in E$ được gọi là cạnh bội nếu chúng cùng chung cặp đỉnh.

Giả đồ thị vô hướng. Đồ thị G = <V, E> bao gồm V là tập đỉnh, E là họ các cặp không có thứ tự gồm hai phần tử (hai phần tử không nhất thiết phải khác nhau) trong V được gọi là các cạnh. Cạnh e được gọi là khuyên nếu có dạng e =(u, u), trong đó u là đỉnh nào đó thuộc V.

Đơn đồ thị có hướng. Đồ thị G = <V, E> bao gồm V là tập các đỉnh, E là tập các cặp có thứ tự gồm hai phần tử của V gọi là các cung. Giữa hai đỉnh bất kỳ của G tồn tại nhiều nhất một cung.

Đa đồ thị có hướng. Đồ thị G = <V, E> bao gồm V là tập đỉnh, E là cặp có thứ tự gồm hai phần tử của V được gọi là các cung. Hai cung e_1, e_2 tương ứng với cùng một cặp đỉnh được gọi là cung lặp.

Giả đồ thị có hướng. Đa đồ thị G = <V, E>, trong đó V là tập đỉnh, E là tập các cặp không có thứ tự gồm hai phần tử (hai phần tử không nhất thiết phải khác nhau) trong V được gọi là các cung. Cung e được gọi là khuyên nếu có dạng e =(u, u), trong đó u là đỉnh nào đó thuộc V.

6.1.2. Một số thuật ngữ trên đồ thị vô hướng

Đỉnh kề. Hai đỉnh *u* và *v* của đồ thị vô hướng $G = <V, E>$ được gọi là kề nhau nếu (u,v) là cạnh thuộc đồ thị *G*. Nếu $e = (u, v)$ là cạnh của đồ thị *G* thì ta nói cạnh này liên thuộc với hai đỉnh *u* và *v*, hoặc ta nói cạnh *e* nối đỉnh *u* với đỉnh *v*, đồng thời các đỉnh *u* và *v* sẽ được gọi là đỉnh đầu của cạnh (u,v).

Bậc của đỉnh. Ta gọi bậc của đỉnh *v* trong đồ thị vô hướng là số cạnh liên thuộc với nó và ký hiệu là $deg(v)$. Đỉnh có bậc là 0 được gọi là đỉnh cô lập. Đỉnh có đỉnh bậc 1 được gọi là đỉnh treo.

Đường đi, chu trình. Đường đi độ dài *n* từ đỉnh *u* đến đỉnh *v* trên đồ thị vô hướng $G=<V,E>$ là dãy $x_0, x_1, \ldots, x_{n-1}, x_n$, trong đó *n* là số nguyên dương, $x_0=u$, $x_n=v$, $(x_i, x_{i+1}) \in E$, $i = 0, 1, 2, \ldots, n-1$. Đường đi có đỉnh đầu trùng với đỉnh cuối gọi là chu trình.

Tính liên thông. Đồ thị vô hướng được gọi là liên thông nếu luôn tìm được đường đi giữa hai đỉnh bất kỳ của nó.

Thành phần liên thông. Đồ thị vô hướng liên thông thì số thành phần liên thông là 1. Đồ thị vô hướng không liên thông thì số liên thông của đồ thị là số các đồ thị con của nó liên thông.

Đỉnh trụ. Đỉnh $u \in V$ được gọi là đỉnh trụ nếu loại bỏ u cùng với các cạnh nối với u làm tăng thành phần liên thông của đồ thị.

Cạnh cầu. Cạnh $(u,v) \in E$ được gọi là cầu nếu loại bỏ (u,v) làm tăng thành phần liên thông của đồ thị.

Đỉnh rẽ nhánh. Đỉnh s được gọi là đỉnh rẽ nhánh (đỉnh thắt) của cặp đỉnh u, v nếu mọi đường đi từ u đến v đều qua s.

6.1.3. Một số thuật ngữ trên đồ thị có hướng

Đỉnh kề. Nếu $e=(u,v)$ là cung của đồ thị có hướng *G* thì ta nói hai đỉnh *u* và *v* là kề nhau, và nói cung (u, v) nối đỉnh *u* với đỉnh *v*, hoặc nói cung này đi ra khỏi đỉnh *u* và đi vào đỉnh *v*. Đỉnh *u* được gọi là đỉnh đầu, đỉnh *v* được gọi là đỉnh cuối của cung (u,v).

Bán bậc của đỉnh. Ta gọi bán bậc ra của đỉnh *v* trên đồ thị có hướng là số cung của đồ thị đi ra khỏi *v* và ký hiệu là $deg^+(v)$. Ta gọi bán bậc vào của đỉnh *v* trên đồ thị có hướng là số cung của đồ thị đi vào *v* và ký hiệu là $deg^-(v)$.

Đường đi. Đường đi độ dài *n* từ đỉnh *u* đến đỉnh *v* trong đồ thị có hướng $G=<V,A>$ là dãy $x_0, x_1, \ldots, x_n$, trong đó, *n* là số nguyên dương, $u = x_0$, $v = x_n$, $(x_i, x_{i+1}) \in E$. Đường đi như trên có thể biểu diễn thành dãy các cung : $(x_0, x_1), (x_1, x_2), \ldots, (x_{n-1}, x_n)$. Đỉnh *u* được gọi là đỉnh đầu, đỉnh *v* được gọi là đỉnh cuối của đường đi. Đường đi có đỉnh đầu trùng với đỉnh cuối ($u=v$) được gọi là một chu trình. Đường đi hay chu trình được gọi là đơn nếu như không có hai cạnh nào lặp lại.

Liên thông mạnh. Đồ thị có hướng G=<V,E> được gọi là liên thông mạnh nếu giữa hai đỉnh bất kỳ $u \in V$, $v \in V$ đều có đường đi từ u đến v.

Liên thông yếu. Ta gọi đồ thị vô hướng tương ứng với đồ thị có hướng G=<V,E> là đồ thị tạo bởi G và bỏ hướng của các cạnh trong G. Khi đó, đồ thị có hướng G=<V,E> được gọi là liên thông yếu nếu đồ thị vô hướng tương ứng với nó là liên thông.

Thành phần liên thông mạnh. Đồ thị con có hướng H =<V1, E1> được gọi là một thành phần liên thông mạnh của đồ thị có hướng G=<V,E> nếu $V1 \subseteq V$, $E1 \subseteq E$ và H liên thông mạnh.

6.1.4. Một số loại đồ thị đặc biệt

Dưới đây là một số dang đơn đồ thị vô hướng đặc biệt có nhiều ứng dụng khác nhau của thực tế.

Đồ thị đầy đủ. Đồ thị đầy đủ n đỉnh, ký hiệu là Kn, là đơn đồ thị vô hướng mà giữa hai đỉnh bất kỳ của nó đều có cạnh nối. Ví dụ đồ thị K3, K4, K5 trong Hình 5.1a.

Đồ thị vòng. Đồ thị vòng Cn ($n \geq 3$) có các cạnh (1,2), (2,3),..,(n-1,n), (n,1). Ví dụ đồ thị C3, C4, C5 trong Hình 5.1.b.

Đồ thị bánh xe. Đồ thị bánh xe Wn thu được bằng cách bổ sung một đỉnh nối với tất cả các đỉnh của Cn. Ví dụ đồ thị W3, W4, W5 trong Hình 5.1.c.

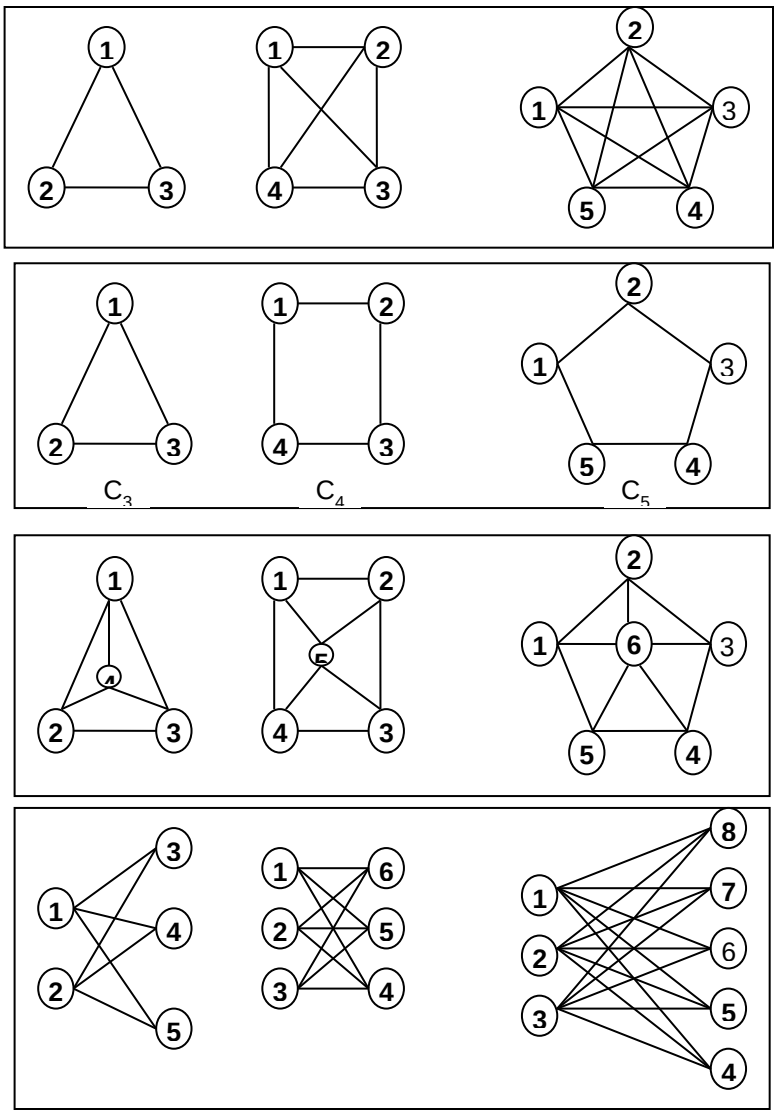
Đồ thị hai phía. Đồ thị G =<V,E> được gọi là đồ thị hai phía nếu tập đỉnh V của nó có thể phân hoạch thành hai tập X và Y sao cho mỗi cạnh của đồ thị chỉ có dạng (*x*, *y*), trong đó $x \in X$ và $y \in Y$. Ví dụ đồ thị $K_{2,3}$, K_{33}, $K_{3,5}$ trong Hình 5.1.d.

Hình 4.1a

Hình 4.1b

Hình 4.1c

Hình 4.1d



Hình 5.1. Một số dạng đồ thị đặc biệt.

6.2. Biểu diễn đồ thị

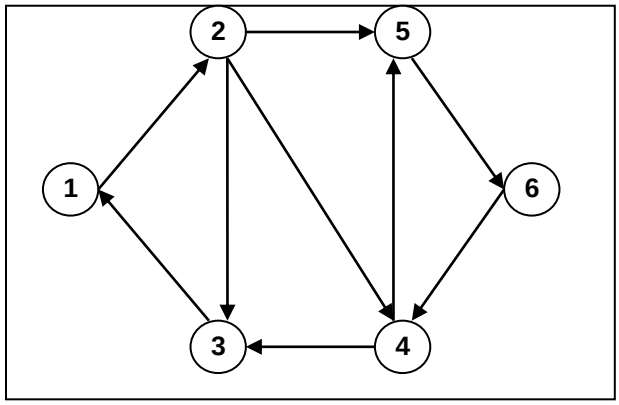
Để lưu trữ, xử lý hiệu quả ta cần có phương pháp biểu diễn đồ thị trên máy tính. Ba phương pháp biểu diễn thông dụng thường được ứng dụng trên đồ thị đó là: ma trận kề, danh sách cạnh và danh sách kề. Phương pháp biểu diễn cụ thể được thể hiện như dưới đây.

6.2.1. Biểu diễn bằng ma trận kề

Phương pháp biểu diễn đồ thị bằng ma trận kề là phép làm tương ứng đồ thị G = <V,E> với một ma trận vuông cấp n. Các phần tử của ma trận kề được xác định như dưới đây.

$$a_{uv} = \begin{cases} 1 & nếu\ (u,v) \in E \\ 0 & otherwise \end{cases}$$

Ví dụ với đồ thị cho bởi Hình 5.2 sẽ cho ta biểu diễn ma trận kề như sau:



$$\begin{vmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{vmatrix}$$

Hình 5.2. Biểu diễn ma trận kề của đồ thị.

Tính chất của ma trận kề:

- Ma trận kề biểu diễn đồ thị vô hướng G = <V,E> là ma trận đối xứng.
- Ma trận kề biểu diễn đồ thị có hướng G =<V,E> thường là không đối xứng.
- Tổng các phần tử của ma trận kề biểu diễn đồ thị vô hướng G = <V,E> bằng *2.m*, trong đó *m* là số cạnh của đồ thị.
- Tổng các phần tử của ma trận kề biểu diễn đồ thị có hướng G = <V,E> bằng đúng *m*, trong đó *m* là số cạnh của đồ thị.
- Tổng các phần tử của hàng *u* hoặc cột *u* của ma trận kề biểu diễn đồ thị vô hướng G= <V,E> là bậc của đỉnh u (deg(u)).
- Tổng các phần tử của hàng *u* của ma trận kề biểu diễn đồ thị có hướng G= <V,E> là bán bậc ra của đỉnh u ($deg^+(u)$).
- Tổng các phần tử của cột *u* của ma trận kề biểu diễn đồ thị có hướng G= <V,E> là bán bậc vào của đỉnh u ($deg^-(u)$).

Ưu điểm của ma trận kề:

- Đơn giản dễ cài đặt trên máy tính bằng cách sử dụng một mảng hai chiều.
- Dễ dàng kiểm tra được hai đỉnh *u, v* có kề với nhau hay không bằng đúng một phép so sánh (a[u][v]≠0).

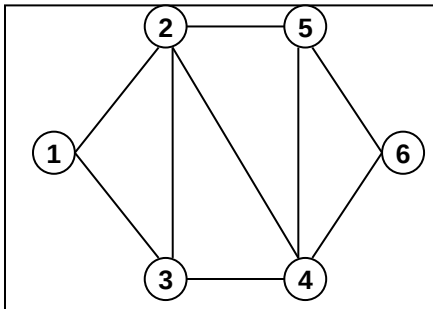
Nhược điểm của ma trận kề:

- Lãng phí bộ nhớ: bất kể số cạnh nhiều hay ít ta cần n^2 đơn vị bộ nhớ để biểu diễn.
- Không thể biểu diễn được với các đồ thị có số đỉnh lớn .
- Để xem xét đỉnh đỉnh *u* có những đỉnh kề nào cần mất n phép so sánh kể cả đỉnh u là đỉnh cô lập hoặc đỉnh treo.

6.2.2. Biểu diễn đồ thị bằng danh sách cạnh

Phương pháp biểu diễn đồ thị G =<V,E> bằng cách liệt kê tất cả các cạnh của nó được gọi là phương pháp biểu diễn bằng danh sách cạnh. Đối với đồ thị có hướng ta liệt

kê các cung tương ứng. Đối với đồ thị vô hướng ta chỉ cần liệt kê các cạnh (u,v)∈E mà không cần liệt kê các cạnh (v,u)∈E. Ví dụ về biểu diễn đồ thị bằng danh sách cạnh được cho trong Hình 5.3.



Đỉnh đầu	Đỉnh cuối
1	2
1	3
2	3
2	4
2	5
3	4
4	5
4	6

Hình 5.3. Biểu diễn đồ thị bằng danh sách cạnh

Tính chất của danh sách cạnh:

- Đỉnh đầu luôn nhỏ hơn đỉnh cuối của mỗi cạnh đối với đồ thị vô hướng.
- Đỉnh đầu không phải lúc nào cũng nhỏ hơn đỉnh cuối của mỗi cạnh đối với đồ thị có hướng.
- Số các số có giá trị *u* thuộc cả tập đỉnh đầu và tập đỉnh cuối các cạnh là bậc của đỉnh u đối với đồ thị vô hướng.
- Số các số có giá trị *u* thuộc cả tập đỉnh đầu các cạnh là bán bậc ra của đỉnh u đối với đồ thị có hướng.
- Số các số có giá trị *u* thuộc cả tập đỉnh cuối là bán bậc vào của đỉnh *u* đối với đồ thị có hướng.

Ưu điểm của danh sách cạnh:

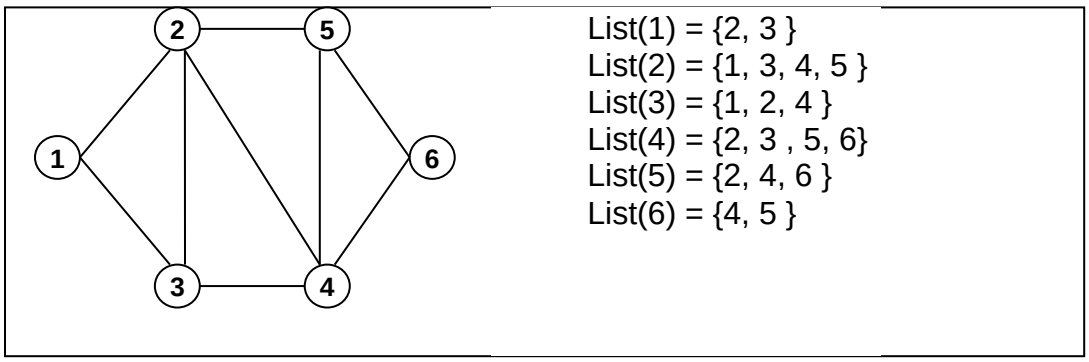
- Trong trường hợp đồ thị thưa (m<6n), biểu diễn bằng danh sách cạnh tiết kiệm được không gian nhớ.
- Thuận lợi cho một số thuật toán chỉ quan tâm đến các cạnh của đồ thị.

Nhược điểm của danh sách cạnh:

- Khi cần duyệt các đỉnh kề với đỉnh u bắt buộc phải duyệt tất cả các cạnh của đồ thị. Điều này làm cho thuật toán có chi phí tính toán cao.

6.2.3. Biểu diễn đồ thị bằng danh sách kề

Ta định nghĩa $List(u) = \{v \in V : (u,v) \in E\}$ là danh sách các đỉnh kề với đỉnh *u*. Biểu diễn đồ thị bằng danh sách kề là phương pháp liệt kê tập đỉnh kề của mỗi đỉnh. Ví dụ về biểu diễn đồ thị bằng danh sách kề được cho trong Hình 5.4.



Hình 5.4. Biểu diễn đồ thị bằng danh sách kề.

Tính chất của danh sách kề:

- Lực lượng tập đỉnh kề của đỉnh *u* là bậc của đỉnh *u* đối với đồ thị vô hướng (deg(*u*)=|List(*u*)|.
- Lực lượng tập đỉnh kề của đỉnh *u* là bán bậc ra của đỉnh *u* đối với đồ thị có hướng ($deg^+(u)$=|List(*u*)|.
- Số các số có giá trị *u* thuộc tất cả các danh sách kề là bán bậc vào của đỉnh *u* đối với đồ thị có hướng.

Ưu điểm của danh sách kề:

- Dễ dàng duyệt tất cả các đỉnh của một danh sách kề.
- Dễ dàng duyệt các cạnh của đồ thị trong mỗi danh sách kề.
- Tối ưu việc cài đặt một số giải thuật trên đồ thị.

Nhược điểm của danh sách kề:

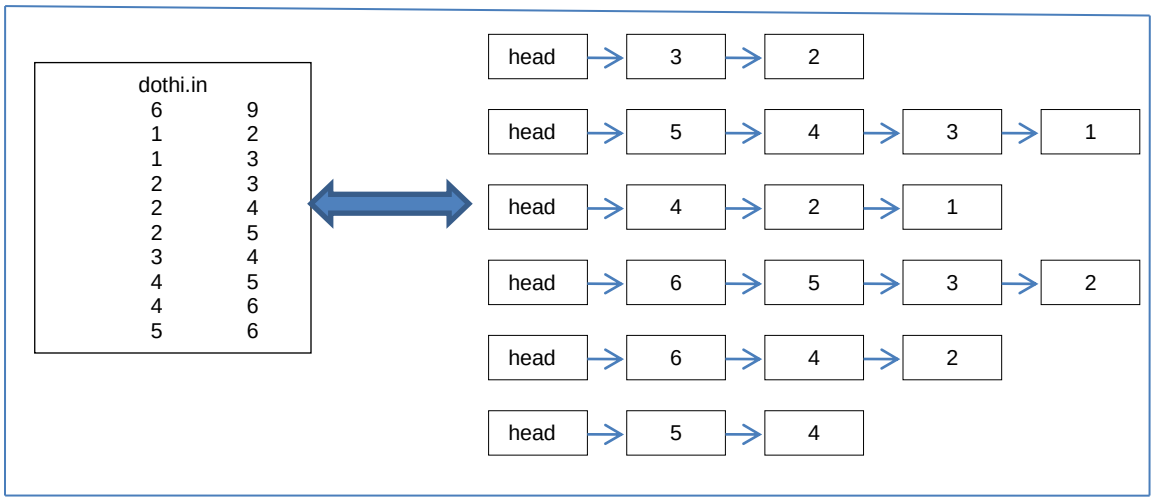
- Khó khăn cho người học có kỹ năng lập trình yếu vì khi biểu diễn đồ thị ta phải dùng một mảng, mỗi phần tử của nó là một danh sách liên kết.

6.2.4. Biểu diễn đồ thị bằng danh sách kề dựa vào danh sách liên kết

Như đã đề cập ở trên, để biểu diễn đồ thị G =<V,E> bằng danh sách kề ta cần phải tạo nên một mảng, mỗi phần từ của mảng là một danh sách liên kết. Mỗi danh danh sách liên kết lưu trữ danh sách kề của đỉnh tương ứng. Chương trình dưới đây cài đặt phương pháp biểu diễn đồ thị bằng danh sách kề với khuôn dạng dữ liệu trong file dothi.in theo khuôn dạng:

- Dòng đầu tiên ghi lại hai số N, M tương ứng với số đỉnh và số cạnh của đồ thị. Hai số được viết cách nhau một vài khoảng trống.
- M dòng kế tiếp mỗi dòng ghi lại một cạnh của đồ thị. Đỉnh đầu và đỉnh cuối mỗi được viết cách nhau một vài khoảng rỗng.

Ví dụ với đồ thị trong Hình 5.4 sẽ cho ta file dothi.in như dưới đây:



Hình 5.5. Ví dụ về file dothi.in

```
#include <iostream>
#include <fstream>
using namespace std;
struct AdjListNode{//định nghĩa danh sách kề của đỉnh
        int dest;//đỉnh kề của đỉnh
        struct AdjListNode* next; //con trỏ đến đỉnh kề tiếp theo
};
 struct AdjList{ //danh sách kề của đỉnh
        AdjListNode *head;//là một danh sách liên kết
};

class Graph{
   private:
        int V;//tập đỉnh của đồ thị
        AdjList* array;//mảng mỗi phần tử là một danh sách kề
   public:
        Graph(int V){//constructor của lớp
                this->V = V; //thiết lập tập đỉnh V là V
                array = new AdjList [V];//cấp phát miền nhớ cho V danh sách liên kết
                for (int i = 0; i < V; ++i) //thiết lập tất cả các danh sách liên kết là NULL
                        array[i].head = NULL;
        }
        AdjListNode* newAdjListNode(int dest) {//thêm một đỉnh mới vào danh sách kề
                AdjListNode* newNode = new AdjListNode; //cấp phát miền nhớ cho node
                newNode->dest = dest;//thiết lập node mới là một đỉnh kề dest
```

```
            newNode->next = NULL;//thiết lập liên kết cho đỉnh kề dest là NULL
            return newNode;//trả lại node kề mới
        }
        void addEdge(int src, int dest){//src là đỉnh đầu, dest là đỉnh cuối của cạnh
            AdjListNode* newNode;
             newNode = newAdjListNode(dest);//cấp phát miền nhớ cho đỉnh kề dest
            newNode->next = array[src].head;//danh sách kề của node src
            array[src].head = newNode;//liên kết dest với danh sách kề của src
            //đồ thị vô hướng ta thêm đoạn này. Đồ thị có hướng bỏ đoạn này
            newNode = newAdjListNode(src);//thêm src vào danh sách kề của dest
            newNode->next = array[dest].head; //liên kết src với danh sách kề của dest
            array[dest].head = newNode;//node src được thêm vào dest
        }
        void printGraph(){//hien thi bieu dien
            int v;
            for (v = 0; v < V; ++v){
                AdjListNode* pCrawl = array[v].head;
                cout<<"\n Danh sách kề đỉnh "<<v<<"\n head ";
                while (pCrawl){
                    cout<<"-> "<<pCrawl->dest;
                    pCrawl = pCrawl->next;
                }
                cout<<endl;
            }
    }
};//end class
 int main(){
        ifstream fp("dothi.in"); //mở file để đọc
        int n, m, dau, cuoi;
        fp>>n>>m;//đọc số đỉnh và số cạnh của đồ thị
        Graph gh(n+1); //gh là đồ thị n đỉnh tính từ 1, 2, .., n
        for(int i=1; i<=m; i++){ //duyệt m cạnh tiếp theo
            fp>>dau>>cuoi;//đọc một cạnh
            gh.addEdge(dau, cuoi);//thêm cuoi vào list(dau) và thêm dau vào list(cuoi)
        }
        fp.close();
        gh.printGraph();
```

```
}
```

6.2.5. Biểu diễn đồ thị bằng danh sách kề dựa vào list STL

```
#include<iostream>
#include <list>
#include <fstream>
using namespace std;
class Graph{//định nghĩa lớp Graph
        int V;    // tập đỉnh V của đồ thị
        list<int> *adj;    // con trỏ đến mảng các danh sách kề
  public:
        Graph(int V);  // Constructor của lớp
        void addEdge(int v, int w); // thêm một cạnh vào đồ thị
        void printGraph();//in ra danh sách kề
};
 Graph::Graph(int V){
        this->V = V;
        adj = new list<int>[V];
}
void Graph::addEdge(int v, int w){
        adj[v].push_back(w); // thêm đỉnh w vào list(v).
        adj[w].push_back(v); // Đồ thị vô hướng: thêm đỉnh v vào list(w).
}
 void Graph::printGraph(){
        int v; list<int>::iterator i;
        for (v = 1; v < V; ++v){
                cout<<"\n Danh sách kề của đỉnh "<<v<<"\n head ";
                for(i=adj[v].begin(); i != adj[v].end(); ++i)
                        cout<<"-> "<<*i;
                cout<<endl;
        }
}
int main(){
        ifstream fp("dothi.in");  int n, m, dau, cuoi;
        fp>>n>>m;    Graph gh(n+1);
        for(int i=1; i<=m; i++){
                fp>>dau>>cuoi;
```

```
            gh.addEdge(dau, cuoi);
        }
        fp.close();
        gh.printGraph();
}
```

6.3. Các thuật toán tìm kiếm trên đồ thị

Có nhiều thuật toán trên đồ thị được xây dựng để duyệt tất cả các đỉnh của đồ thị sao cho mỗi đỉnh được viếng thăm đúng một lần. Những thuật toán như vậy được gọi là thuật toán tìm kiếm trên đồ thị. Chúng ta cũng sẽ làm quen với hai thuật toán tìm kiếm cơ bản, đó là duyệt theo chiều sâu DFS (Depth First Search) và duyệt theo chiều rộng BFS (Breath First Search). Trên cơ sở của hai phép duyệt cơ bản, ta có thể áp dụng chúng để giải quyết một số bài toán quan trọng của lý thuyết đồ thị.

6.3.1. Thuật toán tìm kiếm theo chiều sâu (Depth First Search)

Tư tưởng cơ bản của thuật toán tìm kiếm theo chiều sâu là bắt đầu tại một đỉnh v_0 nào đó, chọn một đỉnh u bất kỳ kề với v_0 và lấy nó làm đỉnh duyệt tiếp theo. Cách duyệt tiếp theo được thực hiện tương tự như đối với đỉnh v_0 với đỉnh bắt đầu là u.

Để kiểm tra việc duyệt mỗi đỉnh đúng một lần, chúng ta sử dụng một mảng chuaxet[] gồm n phần tử (tương ứng với n đỉnh), nếu đỉnh thứ u đã được duyệt, phần tử tương ứng trong mảng chuaxet[u] có giá trị FALSE. Ngược lại, nếu đỉnh chưa được duyệt, phần tử tương ứng trong mảng có giá trị TRUE.

Biểu diễn thuật toán DFS(u):

```
Thuật toán DFS (u): //u là đỉnh bắt đầu duyệt
Begin
        <Thăm đỉnh u>;//duyệt đỉnh u
        chuaxet[u] := FALSE;//xác nhận đỉnh u đã duyệt
        for   each v ∈ke(u) do //lấy mỗi đỉnh v∈Ke(u).
                if  (chuaxet[v] )  then //vếu đỉnh v chưa duyệt
                        DFS(v); //duyệt theo chiều sâu bắt đầu từ đỉnh v
                EndIf;
        EndFor;
End.
```

Thuật toán DFS(u) có thể khử đệ qui bằng cách sử dụng ngăn xếp như Hình 5.6.

```
Thuật toán DFS(u):
Begin
    Bước 1 (Khởi tạo):
        stack = ∅; //Khởi tạo stack là ∅
        Push(stack, u); //Đưa đỉnh u vào ngăn xếp
        <Thăm đỉnh u>; //Duyệt đỉnh u
        chuaxet[u] = False; //Xác nhận đỉnh u đã duyệt
    Bước 2 (Lặp) :
        while ( stack ≠ ∅ ) do
            s = Pop(stack); //Loại đỉnh ở đầu ngăn xếp
            for each t∈ Ke(s) do //Lấy mỗi đỉnh t∈Ke(s)
                if ( chuaxet[t] ) then //Nếu t đúng là chưa duyệt
                    <Thăm đỉnh t>; // Duyệt đỉnh t
                    chuaxet[t] = False; // Xác nhận đỉnh t đã duyệt
                    Push(stack, s);//Đưa s vào stack
                    Push(stack, t); //Đưa t vào stack
                    break;          //Chỉ lấy một đỉnh t
                EndIf;
            EndFor;
        EndWhile;
    Bước 3 (Trả lại kết quả):
        Return(<Tập đỉnh đã duyệt>);
End.
```

Hình 5.6. Thuật toán DFS(u) dựa vào ngăn xếp.

Độ phức tạp thuật toán DFS:

Độ phức tạp thuật toán DFS(u) phụ thuộc vào phương pháp biểu diễn đồ thị. Độ phức tạp thuật toán DFS(u) theo các dạng biểu diễn đồ thị như sau:

- Độ phức tạp thuật toán là $O(n^2)$ trong trường hợp đồ thị biểu diễn dưới dạng ma trận kề, với n là số đỉnh của đồ thị.
- Độ phức tạp thuật toán là O(n.m) trong trường hợp đồ thị biểu diễn dưới dạng danh sách cạnh, với n là số đỉnh của đồ thị, m là số cạnh của đồ thị.
- Độ phức tạp thuật toán là O(max(n, m)) trong trường hợp đồ thị biểu diễn dưới dạng danh sách kề, với n là số đỉnh của đồ thị, m là số cạnh của đồ thị.

Bạn đọc tự chứng minh hoặc có thể tham khảo trong các tài liệu [1, 2, 3].

Cài đặt thuật toán DFS đệ qui:

Thuật toán được cài đặt theo khuôn dạng dữ liệu tổ chức trong file dothi.in được qui ước dưới dạng danh sách cạnh.

```
#include	<iostream>
#include	<list>
#include	<fstream>
#include	<iomanip>
#include	<stack>
using namespace std;
class Graph{//định nghĩa lớp Graph
   private:
	int	V;	// số đỉnh của đồ thị
	list	<int>	*adj;	// con trỏ đến mảng các danh sách kề adj
	bool *chuaxet; //mảng ghi nhận các đỉnh cưa xét
  public:
	Graph(int V);  // constructor của lớp
	void addEdge(int v, int w); // thêm cạnh (u, w) vào đồ thị
	void printGraph();//in biểu diễn danh sách kề
	void DFS(int u);//thuật toán DFS đệ qui
};
Graph::Graph(int V){ //constructor của lớp
	this->V = V;//thiết lập V=0, 1, 2, .., V-1
	adj = new list<int>[V];//thiết lập V danh sách kề
	chuaxet = new bool[V];//thiết lập trạng thái các đỉnh đều chưa xét
	for(int u=0; u<V; u++) //duyệt trên tập đỉnh V
		chuaxet[u]=true;  //thiết lập tất cả các đỉnh đều chưa xét
}
void Graph::addEdge(int v, int w){//thêm cạnh (v,w) vào đồ thị
	adj[v].push_back(w); //thêm w vào list(v) nếu là đồ thị có hướng
	adj[w].push_back(v); //thêm v vào list(w) nếu là đồ thị vô hướng
}
void Graph::DFS(int u){ //thuật toán duyệt theo chiều sâu
	cout << u <<setw(3); //thăm đỉnh u
	chuaxet[u] = false; //ghi nhận đỉnh u đã xét
	list	<int>  ::iterator  v;//lấy v là iterator
	for (v = adj[u].begin(); v != adj[u].end(); ++v){ //duyệt các đỉnh adj[v]
		if (chuaxet[*v]) //nếu v đúng là chưa được xét
			DFS(*v); //duyệt theo chiều sâu bắt đầu từ v
	}
}
void Graph::printGraph(){
	list<int>::iterator v;
	cout<<"\n Số đỉnh đồ thị:"<<V-1;
	for (int u = 1; u < V; ++u){
		printf("\n List(%d):",u);
		for(v=adj[u].begin(); v != adj[u].end(); ++v)
```

```
                    cout<<"-> "<<*v;
                cout<<endl;
        }
}
int main(void){
        ifstream fp("dothi.in");
        int n, m, dau, cuoi;
        fp>>n>>m;//đọc số đỉnh và số cạnh của đồ thị
        Graph gh(n+1);//khởi tạo đồ thị với V=1, 2, ..,n
        for(int i=1; i<=m; i++){//chuyển đồ thị thành biểu diễn danh sách kề
                fp>>dau>>cuoi; //đọc một cạnh
                gh.addEdge(dau, cuoi);//thêm cạnh vào đồ thị
        }
        fp.close();
        gh.printGraph(); //in biểu diễn danh sách kề của đồ thị
        cout<<"\n Kết quả duyệt:";
        gh.DFS (5);
}
```

Cài đặt thuật toán DFS dựa vào ngăn xếp:

Thuật toán được cài đặt theo khuôn dạng dữ liệu tổ chức trong file dothi.in được qui ước dưới dạng danh sách cạnh.

```
#include      <iostream>
#include      <list>
#include      <fstream>
#include      <iomanip>
#include      <stack>
using namespace std;
class Graph{//định nghĩa lớp Graph
   private:
        int      V;    // số đỉnh của đồ thị
        list     <int>   *adj;    // con trỏ đến mảng các danh sách kề adj
        bool *chuaxet; //mảng ghi nhận các đỉnh cưa xét
  public:
        Graph(int V);  // constructor của lớp
        void addEdge(int v, int w); // thêm cạnh (u, w) vào đồ thị
        void DFS_Stack(int u);//thuật toán DFS dựa vào ngăn xếp
};
Graph::Graph(int V){ //constructor của lớp
        this->V = V;//thiết lập V=0, 1, 2, .., V-1
        adj = new list<int>[V];//thiết lập V danh sách kề
        chuaxet = new bool[V];//thiết lập trạng thái các đỉnh đều chưa xét
```

```
	for(int u=0; u<V; u++) //duyệt trên tập đỉnh V
		chuaxet[u]=true;  //thiết lập tất cả các đỉnh đều chưa xét
}
void Graph::addEdge(int v, int w){//thêm cạnh (v,w) vào đồ thị
	adj[v].push_back(w); //thêm w vào list(v) nếu là đồ thị có hướng
	adj[w].push_back(v); //thêm v vào list(w) nếu là đồ thị vô hướng
}
void Graph::DFS_Stack(int u){ //thuật toán DFS dựa vào stack
	stack <int> Stack; //khởi tạo stack rỗng
	Stack.push(u); //đưa đỉnh u vào ngăn xếp
	cout << u <<setw(3); //thăm đỉnh u
	chuaxet[u] = false; //ghi nhận đỉnh u đã thăm
	list<int>::iterator t; //t là iterator trên danh sách
	while(!Stack.empty()){ //lặp khi nào stack rỗng
		int s = Stack.top(); // lấy s là đỉnh đầu ngăn xếp
		Stack.pop(); //đưa s ra khỏi ngăn xếp
		for (t = adj[s].begin(); t != adj[s].end(); ++t){//duyệt các đỉnh t thuộc adj[s]
			if(chuaxet[*t]){ //tìm đỉnh t đầu tiên chưa được xét đến
				cout << *t <<setw(3);//thăm đỉnh t
				Stack.push(s); //đưa s vào ngăn xếp
				Stack.push(*t); //đưa t vào ngăn xếp
				chuaxet[*t]=false; //ghi nhận đỉnh t đã xét
				break;
			}
		}
	}
}
int main(void){
	ifstream fp("dothi.in");  int n, m, dau, cuoi;
	fp>>n>>m;//đọc số đỉnh và số cạnh của đồ thị
	Graph gh(n+1);//khởi tạo đồ thị với V=1, 2, ..,n
	for(int i=1; i<=m; i++){//chuyển đồ thị thành biểu diễn danh sách kề
		fp>>dau>>cuoi; //đọc một cạnh
		gh.addEdge(dau, cuoi);//thêm cạnh vào đồ thị
	}
	fp.close();
	gh.printGraph(); //in biểu diễn danh sách kề của đồ thị
	cout<<"\n Kết quả duyệt:"; gh.DFS_Stack(5);
}
```

6.3.2. Thuật toán tìm kiếm theo chiều rộng (Breadth First Search)

Để ý rằng, với thuật toán tìm kiếm theo chiều sâu, đỉnh thăm càng muộn sẽ trở thành đỉnh sớm được duyệt xong. Đó là kết quả tất yếu vì các đỉnh thăm được nạp vào

stack trong thủ tục đệ qui. Khác với thuật toán tìm kiếm theo chiều sâu, thuật toán tìm kiếm theo chiều rộng thay thế việc sử dụng stack bằng hàng đợi (queue). Trong thủ tục này, đỉnh được nạp vào hàng đợi đầu tiên là u, các đỉnh kề với u là (v_1, v_2, . . ., v_k) được nạp vào hàng đợi nếu như nó chưa được xét đến. Quá trình duyệt tiếp theo được bắt đầu từ các đỉnh còn có mặt trong hàng đợi.

Để ghi nhận trạng thái duyệt các đỉnh của đồ thị, ta cũng vẫn sử dụng mảng chuaxet[] gồm n phần tử thiết lập giá trị ban đầu là TRUE. Nếu đỉnh u của đồ thị đã được duyệt, giá trị chuaxet[u] sẽ nhận giá trị FALSE. Thuật toán dừng khi hàng đợi rỗng. Hình 5.7. dưới đây mô tả chi tiết thuật toán BFS(u).

Biểu diễn thuật toán:

```
Thuật toán BFS(u):
    Bước 1(Khởi tạo):
        Queue = ∅; //tạo lập hàng đợi rỗng
        Push(Queue,u); //đưa u vào hàng đợi
        chuaxet[u] = False; //ghi nhận đỉnh u đã xét
    Bước 2 (Lặp):
        while (Queue ≠ ∅ ) do //lặp đến khi hàng đợi rỗng
            s = Pop(Queue); //đưa s ra khỏi hàng đợi
            <Thăm đỉnh s>;
            for each t∈Ke(s) do //duyệt trên danh sách ke(s)
                if ( chuaxet[t] ) then //nếu t chưa xét
                    Push(Queue, t); //đưa t vào hàng đợi
                    chuaxet[t] = False; //ghi nhận t đã xét
                EndIf ;
            EndFor ;
        EndWhile ;
    Bước 3 (Trả lại kết quả) :
        Return(<Tập đỉnh được duyệt>) ;
End.
```

Hình 5.7. Thuật toán BFS(u).

Độ phức tạp tính toán:

Độ phức tạp thuật toán BFS(u) phụ thuộc vào phương pháp biểu diễn đồ thị. Độ phức tạp thuật toán BFS(u) theo các dạng biểu diễn đồ thị như sau:

- Độ phức tạp thuật toán là $O(n^2)$ trong trường hợp đồ thị biểu diễn dưới dạng ma trận kề, với n là số đỉnh của đồ thị.
- Độ phức tạp thuật toán là O(n.m) trong trường hợp đồ thị biểu diễn dưới dạng danh sách cạnh, với n là số đỉnh của đồ thị, m là số cạnh của đồ thị.

- Độ phức tạp thuật toán là O(max(n, m)) trong trường hợp đồ thị biểu diễn dưới dạng danh sách kề, với n là số đỉnh của đồ thị, m là số cạnh của đồ thị.

Bạn đọc tự chứng minh hoặc có thể tham khảo trong các tài liệu [1, 2, 3].

Cài đặt thuật toán: thuật toán được cài đặt theo khuôn dạng dữ liệu tổ chức trong file dothi.in dưới dạng danh sách cạnh.

```
#include      <iostream>
#include      <list>
#include      <fstream>
#include      <iomanip>
#include      <stack>
using namespace std;
class Graph{//định nghĩa lớp Graph
   private:
        int      V;    // số đỉnh của đồ thị
        list     <int>   *adj;    // con trỏ đến mảng các danh sách kề adj
        bool *chuaxet; //mảng ghi nhận các đỉnh cưa xét
  public:
        Graph(int V);  // constructor của lớp
        void addEdge(int v, int w); // thêm cạnh (u, w) vào đồ thị
        void BFS (int u);//thuật toán BFS dựa vào hàng đợi
};
Graph::Graph(int V){ //constructor của lớp
        this->V = V;//thiết lập V=0, 1, 2, .., V-1
        adj = new list<int>[V];//thiết lập V danh sách kề
        chuaxet = new bool[V];//thiết lập trạng thái các đỉnh đều chưa xét
        for(int u=0; u<V; u++) //duyệt trên tập đỉnh V
                chuaxet[u]=true;  //thiết lập tất cả các đỉnh đều chưa xét
}
void Graph::addEdge(int v, int w){//thêm cạnh (v,w) vào đồ thị
        adj[v].push_back(w); //thêm w vào list(v) nếu là đồ thị có hướng
        adj[w].push_back(v); //thêm v vào list(w) nếu là đồ thị vô hướng
}
void Graph::BFS(int u){//thuật toán BFS
        list<int> queue; //tạo hàng đợi rỗng
        queue.push_back(u);//đưa u vào hàng đợi
        chuaxet[u] = false; //ghi nhận đỉnh u đã xét
        list<int>::iterator t; //sử dụng t là một iterator
        while(!queue.empty()){//lặp đến khi hàng đợi rỗng
                int s = queue.front();//lấy s ở đầu hàng đợi
                cout << s << " ";//thăm đỉnh s
                queue.pop_front();//loại s ra khỏi hàng đợi
                for(t = adj[s].begin(); t != adj[s].end(); ++t){ //duyệt các đỉnh t thuộc list(s)
```

```
                    if(chuaxet[*t]){ //nếu đỉnh t chưa xét
                        queue.push_back(*t); //đưa t vào hàng đợi
                        chuaxet[*t] = false; //xác nhận t đã xét
                    }
                }
        }
}
int main(void){
        ifstream fp("dothi.in");  int n, m, dau, cuoi;
        fp>>n>>m;//đọc số đỉnh và số cạnh của đồ thị
        Graph gh(n+1);//khởi tạo đồ thị với V=1, 2, ..,n
        for(int i=1; i<=m; i++){//chuyển đồ thị thành biểu diễn danh sách kề
                fp>>dau>>cuoi; //đọc một cạnh
                gh.addEdge(dau, cuoi);//thêm cạnh vào đồ thị
        }
        fp.close();
        cout<<"\n Kết quả duyệt:"; gh.BFS (5);
}
```

6.3.3. Ứng dụng của thuật toán DFS và BFS

Có rất nhiều ứng dụng khác nhau của thuật toán DFS và BFS trên đồ thị. Trong khuôn khổ của tài liệu này, ta chỉ xem xét đến một vài ứng dụng cơ bản. Những ứng dụng cụ thể hơn bạn đọc có thể tìm thấy rất nhiều trong các tài liệu khác nhau. Những ứng dụng cơ bản của thuật toán DFS và BFS được đề cập bao gồm:

- Duyệt tất cả các đỉnh của đồ thị.
- Duyệt tất cả các thành phần liên thông của đồ thị.
- Tìm đường đi từ đỉnh s đến đỉnh t trên đồ thị.
- Duyệt các đỉnh trụ trên đồ thị vô hướng.
- Duyệt các cạnh cầu trên đồ thị vô hướng.
- Định chiều đồ thị vô hướng.
- Duyệt các đỉnh rẽ nhánh của cặp đỉnh s, t.
- Xác định tính liên thông mạnh trên đồ thị có hướng.
- Xây dựng cây khung trên đồ thị vô hướng.
- Tìm chu trình trên đồ thị vô hướng, có hướng.
- Định chiều đồ thị…

6.4. Đồ thị Euler

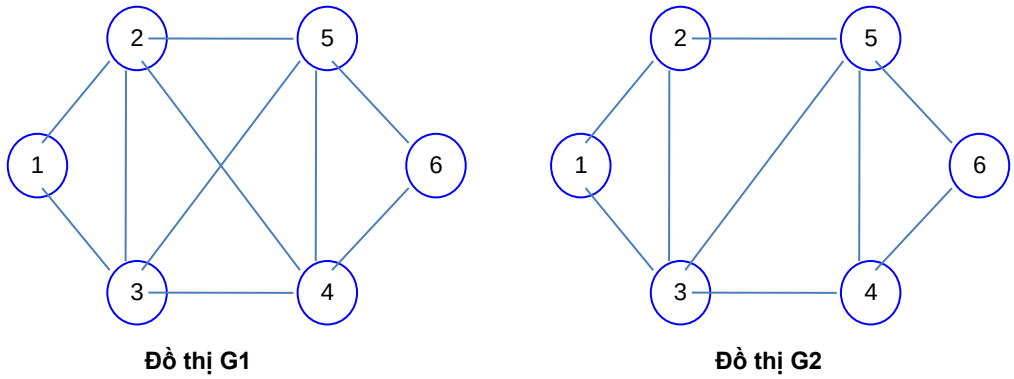
Đồ thị Euler là một dạng đồ thị có nhiều ứng thực tế được định nghĩa như sau:

Định nghĩa.

- Chu trình đơn trong đồ thị G =<V,E> đi qua tất cả các cạnh của đồ thị được gọi là chu trình Euler.
- Đường đi đơn trong G =<V,E> đi qua tất cả các cạnh của đồ thị được gọi là đường đi Euler.
- Đồ thị được G =<V,E> có chu trình Euler được gọi là đồ thị Euler.
- Đồ thị được G =<V,E> có đường đi Euler được gọi là đồ thị nửa Euler.

Ví dụ đồ thị trong Hình 5.8 :

- Đồ thị G1 là Euler vì G1 có chu trình Euler: 1-2-3-4-5-2-4-6-5-3-1.
- Đồ thị G2 là nửa Euler nhưng không là Euler vì G2 có đường đi Euler: 2-1-3-2-5-3-4-6-5-4.



Hình 5.8. Ví dụ về đồ thị Euler, nửa Euler.

Định lý:

- Đồ thị vô hướng liên thông G =<V, E> là Euler khi và chỉ khi tất cả các đỉnh của V đều có bậc chẵn.
- Đồ thị vô hướng liên thông G =<V, E> là nửa Euler nhưng không là Euler khi và chỉ khi G có đúng hai đỉnh bậc lẻ. Đường đi Euler xuất phát tại một đỉnh bậc lẻ và kết thúc tại đỉnh bậc lẻ còn lại.
- Đồ thị có hướng liên thông yếu G =<V, E> là Euler khi và chỉ khi mọi đỉnh của G đều có bán bậc ra bằng bán đỉnh bậc vào của đỉnh.
- Đồ thị có hướng liên thông yếu G =<V, E> là nửa Euler nhưng không là Euler khi và chỉ khi tồn tại đúng hai đỉnh u, v sao cho bán bậc ra của u trừ bán đỉnh bậc vào của u bằng bán đỉnh bậc vào của v trừ bán đỉnh bậc ra của v và bằng 1. Các đỉnh khác u, v còn lại có bán bậc ra bằng bán bậc vào. Đường đi Euler trên đồ thị sẽ xuất phát tại u và kết thúc tại v.

Dựa vào điều kiện cần và đủ đề đồ thị G =<V,E> là Euler hay nửa Euler, ta xây dựng thuật toán kiểm tra và tìm một đường đi, hoặc chu trình Euler trên đồ thị vô hướng và có hướng. Điểm khác biệt giữa thuật toán kiểm tra G=<V,E> là Euler hay nửa Euler chỉ là việc xem xét bậc của đỉnh hay bán bậc của đỉnh. Điểm khác biệt giữa thuật toán tìm

đường đi và chu trình Euler chỉ là việc xem xét đỉnh nào là đỉnh bắt đầu xây dựng đường đi hay chu trình. Để thấy rõ được sự khác biệt này, ta xem xét từng trường hợp cho đồ thị vô hướng và đồ thị có hướng.

6.4.1. Thuật toán tìm một chu trình Euler trên đồ thị vô hướng

Để xác định đồ thị vô hướng G =<V,E> có phải là đồ thị Euler hay không ta chỉ cần dựa vào tính chất của ma trận kề, danh sách cạnh, danh sách kề biểu diễn đồ thị. Thuật toán được thể hiện trong Hình 5.9. Thuật toán tìm một chu trình Euler trên đồ thị vô hướng liên thông được thể hiện trong Hình 5.10.

a) Biểu diễn thuật toán

```
Thuật toán Check_Euler(G=<V, E>):
begin
        for each u∈V do //duyệt trên tập các đỉnh trong V
                if ( deg(u) mode 2 ) //nếu bậc của đỉnh là lẻ
                        return false; //trả lại giá trị false
                endif;
        endfor;
        return true;//nếu tất cả các đỉnh đều có bậc chẵn
end.
```

Hình 5.9. Thuật toán kiểm tra đồ thị vô hướng có là Euler?

Thuật toán Euler-Cycle(u):

Bước 1 (Khởi tạo) :

stack = ∅ ; //*khởi tạo một stack bắt đầu là* ∅

CE = ∅ ; //*khởi tạo mảng CE bắt đầu là* ∅

Push (stack, u) ; //*đưa đỉnh u vào ngăn xếp*

Bước 2 (Lặp):

```
while (stack≠∅ ) do { //lặp cho đến khi stack rỗng
        s = get(stack); //lấy s đỉnh ở đầu ngăn xếp
        if ( Ke(s) ≠ ∅ ) then { //nếu danh sách Ke(s) chưa rỗng
                t =< Đỉnh đầu tiên trong Ke(s)>;
                Push(stack, t)' //đưa t vào stack
                E = E \ (s,t); // loại bỏ cạnh (s,t)
        }
        else { //trường hợp Ke(s)=∅
                s = Pop(stack);// đưa s ra khỏi ngăn xếp
                s ⇒CE; //đưa s sang CE
        }
}
```

Bước 3 (Trả lại kết quả) :

<*Lật ngược lại các đỉnh trong CE ta được chu trình Euler*> ;

Hình 5.10. Thuật toán tìm một chu trình Euler trên đồ thị.

b) **Độ phức tạp thuật toán**:

Bạn đọc tự tìm hiểu và chứng minh độ phức tạp thuật toán trong các tài liệu tham khảo liên quan.

c) **Kiểm nghiệm thuật toán**:

Bạn đọc tự tìm hiểu phương pháp kiểm nghiệm thuật toán trong các tài liệu tham khảo liên quan.

d) **Cài đặt thuật toán**:

Chương trình dưới đây cài đặt thuật toán cho đồ thị đầu vào biểu diễn dưới dạng danh sách cạnh. Chương trình sẽ chuyển đổi biểu diễn đồ thị thành danh sách kề và cài đặt các thuật toán trong Hình 5.9 và 5.10.

```
#include<iostream>
#include <list>
#include <fstream>
#include <iomanip>
#include <stack>
```

```
using namespace std;
class Graph{ //định nghĩa lớp Graph
  private:
        int       V;    // tập đỉnh của đồ thị
        list      <int>  *adj;    // con trỏ đến mảng các danh sách kề
  public:
        Graph(int V);  // constructor của lớp
        void addEdge(int v, int w); // thêm một cạnh của đồ thị vào danh sách kề
        bool CheckEuler(void);//thuật toán Hình 5.9
        void EulerCycle(int u); //thuật toán Hình 5.10
};
Graph::Graph(int V){//Constructor: khởi tạo đồ thị ban đầu
        this->V = V;//thiết lập tập đỉnh
        adj = new list<int>[V];//thiết lập V danh sách kề
}
void Graph::addEdge(int v, int w){
        adj[v].push_back(w); //thêm đỉnh w vào danh sách kề đỉnh v.
        adj[w].push_back(v); //thêm đỉnh v vào danh sách kề đỉnh w.
}
bool Graph::CheckEuler(){ //thuật toán kiểm tra đồ thị Euler
        for(int u=1; u<V; u++){ //duyệt u trên tập đỉnh V
                int deg_u = adj[u].size(); //lấy bậc của đỉnh
                if (deg_u%2!=0) //nếu bậc của đỉnh là lẻ
                  return false;//đồ thị không phải Euler
        }
        return true;//đồ thị là Euler
}
void Graph::EulerCycle(int u){//tìm chu trình Euler bắt đầu tại đỉnh u
        //Bước 1 (Khởi tạo):
        stack <int> stack; //tạo stack rỗng
        int *CE = new int[V], k=0;//tạo mảng CE rỗng
        stack.push(u); //đưa u vào ngăn xếp
        //Bước 2 (Lặp):
        while(!stack.empty()){//lặp đến khi stack rỗng
                int s = stack.top(); //lấy s là đỉnh đầu ngăn xếp
                if(!adj[s].empty()){ //nếu ke(s) khác rỗng
                        int t= adj[s].front();//t là đỉnh đầu tiên trong ke(s)
```

```
                stack.push(t);//đưa t vào ngăn xếp
                adj[s].remove(t);//loại t khỏi ke(s)
                adj[t].remove(s); //loại s khỏi ke(t)
            }
            else {//nếu ke(s) là rỗng
                stack.pop();//loại s khỏi stack
                CE[k]= s; k++;//đưa s sang CE
            }
        }
        //Bước 3 (Trả lại kết quả):
        cout<<"\n Kết quả:";
        for(int t=k-1;t>=0; t--){//lật ngược lại CE
            cout<<EC[t]<<"-";
        }
        cout<<endl;
}
int main(void){
        ifstream fp("dothi.in");
        int n, m, dau, cuoi;
        fp>>n>>m;//đọc số đỉnh và số cạnh của đồ thị
        Graph gh(n+1);//thiết lập đồ thị n đỉnh V=1, 2, .., n
        for(int i=1; i<=m; i++){//chuyển biểu diễn đồ thị thành danh sách kề
            fp>>dau>>cuoi;
            gh.addEdge(dau, cuoi);
        }
        fp.close();
        if(gh.CheckEuler()){ //nếu đồ thị là Euler
            gh.EulerCycle(3);//đưa ra một chu trình Euler
        }
}
```

6.4.2. Thuật toán tìm một chu trình Euler trên đồ thị có hướng

Tìm chu trình Euler trên đồ thị vô hướng và đồ thị có hướng khác biệt nhau duy nhất ở việc kiểm tra đồ thị có hướng là Euler hay không? Để kiểm tra đồ thị có hướng G=<V,E> là Euler ta chỉ cần kiểm duyệt bán bậc ra và bán bậc vào của đỉnh. Nếu tất cả các đỉnh đều có bán đỉnh bậc ra và bán đỉnh bậc vào thì ta kết luận đồ thị là Euler. Nếu tồn tại một đỉnh có bán bậc ra khác bán bậc vào ta kết luận đồ thị không là Euler. Thuật toán kiểm tra đồ thị có hướng G=<V,E> là đồ thị Euler được mô tả trong Hình 5.11.

Thuật toán tìm một chu trình Euler trên đồ thị có hướng bắt đầu tại đỉnh u được giữ nguyên trong Hình 5.10.

a) Biểu diễn thuật toán

```
Thuật toán Check_Euler(G=<V, E>):
begin
        for each u∈V do //duyệt trên tập các đỉnh trong V
                if ( deg+(u) !=deg-(u)) //nếu bán bậc ra khác bán bậc vào của u
                        return false; //trả lại giá trị false
                endif;
        endfor;
        return true;//nếu tất cả các đỉnh đều có bậc chẵn
end.
```

Hình 5.11. Thuật toán kiểm tra đồ thị vô hướng có là Euler?

b) **Độ phức tạp thuật toán**:

Bạn đọc tự tìm hiểu và chứng minh độ phức tạp thuật toán trong các tài liệu tham khảo liên quan.

c) **Kiểm nghiệm thuật toán**:

Bạn đọc tự tìm hiểu phương pháp kiểm nghiệm thuật toán trong các tài liệu tham khảo liên quan.

d) **Cài đặt thuật toán**:

Chương trình dưới đây cài đặt thuật toán cho đồ thị có hướng được biểu diễn dưới dạng danh sách cạnh. Chương trình sẽ chuyển đổi biểu diễn đồ thị thành danh sách kề và cài đặt các thuật toán trong Hình 5.11 và 5.10.

```
#include<iostream>
#include <list>
#include <fstream>
#include <iomanip>
#include <stack>
using namespace std;
class Graph{ //định nghĩa lớp Graph
  private:
        int     V;   // tập đỉnh của đồ thị
        list    <int>  *adj;   // con trỏ đến mảng các danh sách kề
  public:
        Graph(int V);  // constructor của lớp
        void addEdge(int v, int w); // thêm một cạnh của đồ thị vào danh sách kề
```

```
        int NegativeDeg(int u); //tìm bán đỉnh bậc vào của đỉnh u
        bool CheckEuler(void);//thuật toán Hình 5.11
        void EulerCycle(int u); //thuật toán Hình 5.10
};
Graph::Graph(int V){//Constructor: khởi tạo đồ thị ban đầu
        this->V = V;//thiết lập tập đỉnh
        adj = new list<int>[V];//thiết lập V danh sách kề
}
void Graph::addEdge(int v, int w){
        adj[v].push_back(w); //thêm đỉnh w vào danh sách kề đỉnh v.
}
int Graph::NegativeDeg(int u){
        int dem =0;    list <int>::iterator v;
        for(int s=1; s<V; s++){
                for(v=adj[s].begin(); v != adj[s].end(); ++v){
                        if(u==(*v)) dem++;
                }
        }
        return dem;
}

bool Graph::CheckEuler(){ //thuật toán kiểm tra đồ thị Euler
        for(int u=1; u<V; u++){ //duyệt u trên tập đỉnh V
                int deg_u = adj[u].size(); //lấy bán bậc ra của đỉnh u
                int deg_u1 = Negative_Deg(); //lấy bán bậc vào của đỉnh u
                if (deg_u%2!=deg_u1) //nếu u có bán bậc ra khác bán bậc vào
                  return false;//đồ thị không phải Euler
        }
        return true;//đồ thị là Euler
}
void Graph::EulerCycle(int u){//tìm chu trình Euler bắt đầu tại đỉnh u
        //Bước 1 (Khởi tạo):
        stack <int> stack; //tạo stack rỗng
        int *CE = new int[V], k=0;//tạo mảng CE rỗng
        stack.push(u); //đưa u vào ngăn xếp
        //Bước 2 (Lặp):
        while(!stack.empty()){//lặp đến khi stack rỗng
```

```
            int s = stack.top(); //lấy s là đỉnh đầu ngăn xếp
            if(!adj[s].empty()){ //nếu ke(s) khác rỗng
                int t= adj[s].front();//t là đỉnh đầu tiên trong ke(s)
                stack.push(t);//đưa t vào ngăn xếp
                adj[s].remove(t);//loại t khỏi ke(s)
                adj[t].remove(s); //loại s khỏi ke(t)
            }
            else {//nếu ke(s) là rỗng
                stack.pop();//loại s khỏi stack
                CE[k]= s; k++;//đưa s sang CE
            }
        }
        //Bước 3 (Trả lại kết quả):
        cout<<"\n Kết quả:";
        for(int t=k-1;t>=0; t--){//lật ngược lại CE
            cout<<EC[t]<<"-";
        }
        cout<<endl;
}


int main(void){
        ifstream fp("dothi.in");
        int n, m, dau, cuoi;
        fp>>n>>m;//đọc số đỉnh và số cạnh của đồ thị
        Graph gh(n+1);//thiết lập đồ thị n đỉnh V=1, 2, .., n
        for(int i=1; i<=m; i++){//chuyển biểu diễn đồ thị thành danh sách kề
            fp>>dau>>cuoi;
            gh.addEdge(dau, cuoi);
        }
        fp.close();
        if(gh.CheckEuler()){ //nếu đồ thị là Euler
            gh.EulerCycle(3);//đưa ra một chu trình Euler
        }
}
```

6.4.3. Thuật toán tìm một đường đi Euler trên đồ thị vô hướng

Tìm một đường đi Euler cũng giống như tìm một chu trình Euler trên đồ thị vô hướng. Điểm khác biệt duy nhất giữa hai thuật toán này là đỉnh xuất phát để tìm đường đi hay chu trình Euler. Đối với đồ thị Euler ta có thể xây dựng chu trình Euler tại bất kỳ đỉnh nào thuộc tập đỉnh. Đối với đồ thị là nửa Euler nhưng không là Euler, đỉnh để xây dựng đường đi Euler là một trong hai đỉnh có bậc lẻ. Thuật toán kiểm tra một đồ thị vô hướng liên thông là nửa Euler nhưng không là Euler được thể hiện như Hình 5.12.

a) Biểu diễn thuật toán

```
Thuật toán Check_Semi_Euler(G=<V, E>):
Begin
        int   so_dinh_le=0;
        for each u∈V do //duyệt trên tập các đỉnh trong V
                if ( deg (u) %2!=0) //bậc của đỉnh u lẻ
                        so_dinh_le = so_dinh_le +1; //tăng số đỉnh bậc lẻ
                endif;
        endfor;
        if (so_dinh_le==2) //nếu số đỉnh bậc lẻ là 2
                return true;//kết luận đồ thị là nửa euler
        endif;
        return false;//kết luận đồ thị không là nửa Euler
end.
```

Hình 5.12. Thuật toán kiểm tra đồ thị vô hướng liên thông là nửa Euler

b) Độ phức tạp thuật toán

Bạn đọc tự tìm hiểu và chứng minh độ phức tạp thuật toán trong các tài liệu tham khảo liên quan.

c) Kiểm nghiệm thuật toán

Bạn đọc tự tìm hiểu và kiểm nghiệm thuật toán trong các tài liệu tham khảo liên quan.

d) Cài đặt thuật toán

Thuật toán được cài đặt cho đồ thị vô hướng liên thông được biểu diễn dưới dạng danh sách cạnh sau đó chuyển đổi biểu diễn thành dánh sách kề như dưới đây.

```
#include<iostream>
#include <list>
#include <fstream>
#include <iomanip>
#include <stack>
```

```
using namespace std;
class Graph{ //định nghĩa lớp Graph
  private:
        int       V;    // tập đỉnh của đồ thị
        list      <int>  *adj;    // con trỏ đến mảng các danh sách kề
  public:
        Graph(int V);  // constructor của lớp
        void addEdge(int v, int w); // thêm một cạnh của đồ thị vào danh sách kề
        void SemiEuler(void);//thuật toán Hình 5.12
        void EulerCycle(int u); //thuật toán Hình 5.10
};
Graph::Graph(int V){//Constructor: khởi tạo đồ thị ban đầu
        this->V = V;//thiết lập tập đỉnh
        adj = new list<int>[V];//thiết lập V danh sách kề
}
void Graph::addEdge(int v, int w){
        adj[v].push_back(w); //thêm đỉnh w vào danh sách kề đỉnh v.
        adj[w].push_back(v); //thêm đỉnh v vào danh sách kề đỉnh w.
}
void Graph::SemiEuler(void){
        int deg_u, dem=0; //đếm số đỉnh bậc chẵn
        int s = 0, t=0;
        for(int u=1; u<V; u++){ //duyệt trên tập đỉnh V
                deg_u = adj[u].size(); //tìm bậc của đỉnh
                if (deg_u%2!=0) { //nếu đỉnh có bậc lẻ
                        dem++; //ghi nhận số đỉnh bậc lẻ
                        if (s==0 && t==0) s =u; //đỉnh bậc lẻ đầu tiên là s
                        else if (s!=0 && t==0) //đỉnh bậc lẻ thứ nhì là t
                                t = u;
                }
        }
        if (dem==2) EulerCycle(s); //nếu có đúng hai đỉnh bậc lẻ
        else cout<<"\n Đồ thị không là nửa Euler";
}
void Graph::EulerCycle(int u){//tìm chu trình Euler bắt đầu tại đỉnh u
        //Bước 1 (Khởi tạo):
        stack <int> stack; //tạo stack rỗng
```

```
        int *CE = new int[V], k=0;//tạo mảng CE rỗng
        stack.push(u); //đưa u vào ngăn xếp
        //Bước 2 (Lặp):
        while(!stack.empty()){//lặp đến khi stack rỗng
                int s = stack.top(); //lấy s là đỉnh đầu ngăn xếp
                if(!adj[s].empty()){ //nếu ke(s) khác rỗng
                        int t= adj[s].front();//t là đỉnh đầu tiên trong ke(s)
                        stack.push(t);//đưa t vào ngăn xếp
                        adj[s].remove(t);//loại t khỏi ke(s)
                        adj[t].remove(s); //loại s khỏi ke(t)
                }
                else {//nếu ke(s) là rỗng
                        stack.pop();//loại s khỏi stack
                        CE[k]= s; k++;//đưa s sang CE
                }
        }
        //Bước 3 (Trả lại kết quả):
        cout<<"\n Kết quả:";
        for(int t=k-1;t>=0; t--){//lật ngược lại CE
                cout<<EC[t]<<"-";
        }
        cout<<endl;
}
int main(void){
        ifstream fp("dothi.in");
        int n, m, dau, cuoi;
        fp>>n>>m;//đọc số đỉnh và số cạnh của đồ thị
        Graph gh(n+1);//thiết lập đồ thị n đỉnh V=1, 2, .., n
        for(int i=1; i<=m; i++){//chuyển biểu diễn đồ thị thành danh sách kề
                fp>>dau>>cuoi;
                gh.addEdge(dau, cuoi);
        }
        fp.close();
        gh.SemiEuler();
}
```

6.4.4. Thuật toán tìm một đường đi Euler trên đồ thị có hướng

Tìm một đường đi Euler cũng giống như tìm một chu trình Euler trên đồ thị có hướng. Điểm khác biệt duy nhất giữa hai thuật toán này là đỉnh xuất phát để tìm đường đi hay chu trình Euler. Đối với đồ thị Euler ta có thể xây dựng chu trình Euler tại bất kỳ đỉnh nào thuộc tập đỉnh. Đối với đồ thị là nửa Euler nhưng không là Euler, đỉnh để xây dựng đường đi Euler là m đỉnh u có deg+(u)-deg-(u) =1. Thuật toán kiểm tra một đồ thị có hướng liên thông yếu là nửa Euler nhưng không là Euler được thể hiện như Hình 5.13.

a) Biểu diễn thuật toán

```
Thuật toán Check_Semi_Euler(G=<V, E>):
Begin
        int  s, t, dem1 = 0, dem2 = 0;
        for each u∈V do //duyệt trên tập các đỉnh trong V
                if ( deg+(u)- deg-(u)==1){ //tìm hiệu bán bậc của đỉnh u
                        dem1++; s =u; //ghi nhận đỉnh u có deg+(u)- deg-(u)==1
                endif;
                else if (deg-(u)- deg+(u)==1){
                        dem2++; t =u; //ghi nhận đỉnh u có deg-(u)- deg+(u)==1
                endesle;
        endfor;
        if (dem1==1 && dem2==1) //nếu điều này  xảy ra
                return true;//kết luận đồ thị là nửa euler
        endif;
        return false;//kết luận đồ thị không là nửa Euler
end.
```

Hình 5.13. Thuật toán kiểm tra đồ thị có hướng là nửa Euler.

b) Độ phức tạp thuật toán

Bạn đọc tự tìm hiểu và chứng minh độ phức tạp thuật toán trong các tài liệu tham khảo liên quan.

c) Kiểm nghiệm thuật toán

Bạn đọc tự tìm hiểu và kiểm nghiệm thuật toán trong các tài liệu tham khảo liên quan.

d) Cài đặt thuật toán

Thuật toán được cài đặt cho đồ thị vô hướng liên thông yếu được biểu diễn dưới dạng danh sách cạnh sau đó chuyển đổi biểu diễn thành dánh sách kề như dưới đây.

```
#include<iostream>
#include <list>
```

```
#include <fstream>
#include <iomanip>
#include <stack>
using namespace std;
class Graph{ //định nghĩa lớp Graph
  private:
        int       V;    // tập đỉnh của đồ thị
        list      <int>   *adj;    // con trỏ đến mảng các danh sách kề
  public:
        Graph(int V);  // constructor của lớp
        void addEdge(int v, int w); // thêm một cạnh của đồ thị vào danh sách kề
        int NegativeDeg(int u); //tìm bán đỉnh bậc vào của đỉnh u
        void SemiEuler(void);//thuật toán Hình 5.13
        void EulerCycle(int u); //thuật toán Hình 5.10
};
Graph::Graph(int V){//Constructor: khởi tạo đồ thị ban đầu
        this->V = V;//thiết lập tập đỉnh
        adj = new list<int>[V];//thiết lập V danh sách kề
}
void Graph::addEdge(int v, int w){
        adj[v].push_back(w); //thêm đỉnh w vào danh sách kề đỉnh v.
}
int Graph::NegativeDeg(int u){
        int dem =0;    list <int>::iterator v;
        for(int s=1; s<V; s++){
                for(v=adj[s].begin(); v != adj[s].end(); ++v){
                        if(u==(*v)) dem++;
                }
        }
        return dem;
}
void Graph::SemiEuler(){
        int deg_u, deg_u1; //bán bậc ra và bán bậc vào của đỉnh
        int dem1=0, dem2=0;//số lượng định thỏa mãn đồ thị semi-Euler
        int s,  t; //ghi nhận đỉnh có deg+(u)-deg-(u) =1 hoặc deg-(u)-deg+(u) =1
        for(int u=1; u<V; u++){ //duyệt trên tập đỉnh V
                deg_u = adj[u].size(); //tìm bán bậc ra của đỉnh
```

```
            deg_u1 = NegativeDeg(u);//tìm bán bậc vào của đỉnh
            if (deg_u - deg_u1 ==1) { //nếu deg+(u)-deg-(u) =1
                dem1 ++; s = u; //ghi nhận số đỉnh và s = u
            }
            else if (deg_u1 - deg_u ==1){ //nếu deg-(u)-deg+(u) =1
                dem2++; t = u; //ghi nhận số đỉnh và t =u
            }
        }
        if (dem1==1 && dem2==1) //nếu điều này xảy ra
            EulerCycle(s); //ta làm giống như chu trình Euler
        else cout<<"\n Đồ thị không là nửa Euler";
}
void Graph::EulerCycle(int u){//tìm chu trình Euler bắt đầu tại đỉnh u
        //Bước 1 (Khởi tạo):
        stack <int> stack; //tạo stack rỗng
        int *CE = new int[V], k=0;//tạo mảng CE rỗng
        stack.push(u); //đưa u vào ngăn xếp
        //Bước 2 (Lặp):
        while(!stack.empty()){//lặp đến khi stack rỗng
            int s = stack.top(); //lấy s là đỉnh đầu ngăn xếp
            if(!adj[s].empty()){ //nếu ke(s) khác rỗng
                int t= adj[s].front();//t là đỉnh đầu tiên trong ke(s)
                stack.push(t);//đưa t vào ngăn xếp
                adj[s].remove(t);//loại t khỏi ke(s)
                adj[t].remove(s); //loại s khỏi ke(t)
            }
            else {//nếu ke(s) là rỗng
                stack.pop();//loại s khỏi stack
                CE[k]= s; k++;//đưa s sang CE
            }
        }
        //Bước 3 (Trả lại kết quả):
        cout<<"\n Kết quả:";
        for(int t=k-1;t>=0; t--){//lật ngược lại CE
            cout<<EC[t]<<"-";
        }
        cout<<endl;
```

}

6.5. Bài toán xây dựng cây khung của đồ thị

Trong mục này ta đề cập đến một loại đồ thị đơn giản nhất đó là cây. Cây được ứng dụng rộng rãi trong nhiều lĩnh vực khác nhau của tin học như tổ chức các thư mục, lưu trữ dữ liệu, biểu diễn tính toán, biểu diễn quyết định và tổ chức truyền tin. Ta có thể tiếp cận cây bằng lý thuyết đồ thị như dưới đây.

Định nghĩa 1. Ta gọi cây là đồ thị vô hướng liên thông không có chu trình. Đồ thị không liên thông được gọi là rừng. Như vậy, rừng là đồ thị mà mỗi thành phần liên thông của nó là một cây.

Định lý 1. Giả sử T= <V, E> là đồ thị vô hướng n đỉnh. Khi đó những khẳng định sau là tương đương

a) T là một cây.
b) T không có chu trình và có n-1 cạnh.
c) T liên thông và có đúng n-1 cạnh.
d) T liên thông và mỗi cạnh của nó đều là cầu.
e) Giữa hai đỉnh bất kỳ của T được nối với nhau bởi đúng một đường đi đơn.
f) T không chứa chu trình nhưng hễ cứ thêm vào nó một cạnh ta thu được đúng một chu trình.

Định nghĩa 2. Cho G =<V, E> là đồ thị vô hướng liên thông. Ta gọi đồ thị con H=<V,T> là một cây khung của G nếu H là một cây và $T \subseteq E$.

Tiếp cận cây bằng lý thuyết đồ thị, người ta qua tâm đến hai bài toán cơ bản về cây:

Bài toán 1. Cho đồ thị vô hướng G =<V,E>. Hãy xây dựng một cây khung của đồ thị bắt đầu tại đỉnh $u \in V$.

Bài toán 2. Cho đồ thị vô hướng G =<V,E> có trọng số. Hãy xây dựng cây khung có độ dài nhỏ nhất.

Bài toán 1 được giải quyết bằng các thuật toán tìm kiếm cơ bàn: tìm kiếm theo chiều rộng (BFS) hoặc thuật toán tìm kiếm theo chiều sâu (DFS). Bài toán 2 được giải quyết bằng thuật toán Kruskal hoặc PRIM. Dưới đây là nội dung các thuật toán.

6.5.1. Xây dựng cây khung của đồ thị bằng thuật toán DFS

Khi ta thực hiện thủ tục tìm kiếm theo chiều sâu bắt đầu tại đỉnh $u \in V$, ta nhận được tập đỉnh $v \in V$ có cùng thành phần liên thông với u. Nếu DFS(u) = V thì ta kết luận đồ thị liên thông và phép duyệt DFS(u) đi qua đúng n-1 cạnh. Nếu Nếu DFS(u) $\neq$ V thì ta kết luận đồ thị không liên thông. Chính vì vậy, ta có thể sử dụng phép duyệt DFS(u) để xây dựng cây khung của đồ thị. Trong mỗi bước của thuật toán DFS, xuất phát tại đỉnh u ta sẽ thăm được đỉnh v và ta kết nạp cạnh (u,v) vào tập cạnh của cây khung. Các bước tiếp theo được tiến hành tại đỉnh v cho đến khi kết thúc thuật toán DFS. Thuật toán được xây dựng dựa vào ngăn xếp được mô tả chi tiết trong Hình 5.14.

a) Biểu diễn thuật toán

```
Thuật toán Tree-DFS(u):
Begin
    Bước 1 (Khởi tạo):
        T = ∅; //tập cạnh cây khung ban đầu.
        stack = ∅; //thiết lập stack rỗng;
        Push(stack, u); //đưa u vào stack;
        chuaxet[u] = False;//bật trạng thái đã xét của đỉnh u
    Bước 2 (Lặp):
        while (stack≠∅ ) do { //lặp cho đến khi stack rỗng
            s = Pop(stack); //lấy s ra khỏi stack
            for each t∈Ke(s) do { //lặp trên danh sách Ke(s)
                if (chuaxet[t] ) then { //nếu đỉnh t chuaxet
                    Push(stack, s);// đưa s vào stack trước
                    Push(stack, s);// đưa t vào stack sau
                    T = T∪(s,t); //kết nạp (s,t) vào cây khung
                    chuaxet[t] = False; //ghi nhận t đã xét
                    break ;//chỉ lấy đỉnh đầu tiên
                endif ;
            endfor ;
        endwwhile ;
    Bước 3 (Trả lại kết quả) :
        if (| T | < n-1 ) <Đồ thị không liên thông> ;
        else <Ghi nhận tập cạnh T của cây khung> ;
end.
```

Hình 5.14. Xây dựng cây khung của đồ thị bằng thuật toán DFS.

b) Độ phức tạp thuật toán

Độ phức tạp thuật toán Tree-DFS(u) đúng bằng độ phức tạp thuật toán DFS(u).

c) Kiểm nghiệm thuật toán

Bạn đọc tự tìm hiểu và kiểm nghiệm thuật toán trong các tài liệu tham khảo liên quan.

d) Cài đặt thuật toán

```
#include<iostream>
#include <list>
#include <fstream>
#include <iomanip>
#include <stack>
using namespace std;
struct canh{ //biểu diễn một cạnh của đồ thị
    int dau; //đỉnh đầu của cạnh
```

```
        int cuoi; //đỉnh cuối của cạnh
};
class Graph{ //xây dựng lớp đồ thị
  private:
        int V;    // số đỉnh của đồ thị
        list<int> *adj;    // con trỏ đến mảng các danh sách kề
        bool *chuaxet; //mảng chưa xét
        canh *T;//tập cạnh của cây khung
        int sc; //số cạnh của cây khung
 public:
        Graph(int V);  // constructor của lớp
        void addEdge(int v, int w); // thêm một cạnh vào đồ thị
        void Tree_BFS(int u);//thuật toán Tree-BFS
        void Tree_DFS(int u);//thuật toán Tree-DFS
};
Graph::Graph(int V){ //constructor của lớp
        this->V = V;//thiết lập tập đỉnh
        adj = new list<int>[V];//thiết lập V danh sách kề
        T       = new canh[V]; sc =1; //thiết lập số cạnh cây khung
        chuaxet = new bool[V];//thiết lập giá trị mảng chưa xét
        for(int u=0; u<V; u++)       chuaxet[u]=true;
}
void Graph::addEdge(int v, int w){ //thêm một cạnh vào danh sách kề
        adj[v].push_back(w); // thêm w vào list(v)
        adj[w].push_back(v); // thêm v vào list(w)
}
void Graph::Tree_DFS(int u){
        //Bước 1 (Khởi tạo):
        stack <int> Stack; //tạo lập stack rỗng
        Stack.push(u);//đưa u vào ngăn xếp
        chuaxet[u] = false; //xác nhận u đã xét
        list<int>::iterator t; //t là iterator của list
        while(!Stack.empty()){//lặp đến khi stack rỗng
                int s = Stack.top(); //lấy s là đỉnh đầu ngăn xếp
                Stack.pop();//loại s ra khỏi ngăn xếp
                for (t = adj[s].begin(); t != adj[s].end(); ++t){//duyệt trên list(s)
                        if(chuaxet[*t]){//nếu đúng t chưa xét
```

```
                        Stack.push(s);//đưa s vào stack trước
                        Stack.push(*t); //đưa t vào stack sau
                        chuaxet[*t]=false; //ghi nhận t đã xét
                        T[sc].dau = s;T[sc].cuoi = *t;//thêm (s,t) vào cây
                        sc++; //tăng số cạnh lên 1
                        break;//chỉ lấy đỉnh đầu tiên
                    }
                }
            }
            if(sc<V-1){//nếu |T|<n-1
                cout<<"\n Đồ thị không liên thông";
            }
            else {
                cout<<"\n Tập cạnh cây khung:"<<endl;
                for(int i=1; i<sc; i++){
                    cout<<T[i].dau<<setw(3)<<T[i].cuoi<<endl;
                }
            }
        }
        int main(void){
            ifstream fp("Graph.in"); //mở file để đọc
            int n, m, dau, cuoi;
            fp>>n>>m;//đọc số đỉnh và số cạnh của đồ thị
            Graph gh(n+1);//thiết lập đồ thị gồm n đỉnh
            for(int i=1; i<=m; i++){//chuyển đồ thị sang danh sách kề
                fp>>dau>>cuoi;
                gh.addEdge(dau, cuoi);
            }
            fp.close();
            gh.printGraph();
            gh.Tree_BFS(1);
        }
```

6.5.2. Xây dựng cây khung của đồ thị bằng thuật toán BFS

Để tìm một cây khung trên đồ thị vô hướng liên thông ta có thể sử dụng kỹ thuật tìm kiếm theo chiều rộng. Giả sử ta cần xây dựng một cây bao trùm xuất phát tại đỉnh *u* nào đó. Trong cả hai trường hợp, mỗi khi ta đến được đỉnh *v* tức (*chuaxet[v] = False*) từ đỉnh u thì cạnh (*u,v*) được kết nạp vào cây khung.

a) Biểu diễn thuật toán

```
Thuật toán Tree-BFS(u):
Begin
        Bước 1 (Khởi tạo):
                T = ∅; //tập cạnh cây khung ban đầu.
                Queue = ∅; //thiết lập hàng đợi ban đầu;
                Push(Queue, u); //đưa u vào hàng đợi;
                chuaxet[u] = False;//bật trạng thái đã xét của đỉnh u
        Bước 2 (Lặp):
                while (Queue≠∅ ) do { //lặp cho đến khi hàng đợi rỗng
                        s = Pop(Queue); //lấy s ra khỏi hàng đợi
                        for each t∈Ke(s) do { //lặp trên danh sách Ke(s)
                                if (chuaxet[t] ) then { //nếu đỉnh t chuaxet
                                        Push(Queue, t);// đưa t vào hàng đợi
                                        T = T∪(s,t); //kết nạp (s,t) vào cây khung
                                        chuaxet[t] = False; //ghi nhận t đã xét
                                endif ;
                        endfor ;
                endwwhile ;
        Bước 3 (Trả lại kết quả) :
                if (| T | < n-1 ) <Đồ thị không liên thông> ;
                else <Ghi nhận tập cạnh T của cây khung" ;
end.
```

Hình 5.15. Thuật toán Tree-BFS

b) Độ phức tạp thuật toán

Độ phức tạp thuật toán Tree-BFS(u) đúng bằng độ phức tạp thuật toán BFS(u).

c) Kiểm nghiệm thuật toán

Bạn đọc tự tìm hiểu và kiểm nghiệm thuật toán trong các tài liệu tham khảo liên quan.

d) Cài đặt thuật toán

```
#include<iostream>
#include <list>
#include <fstream>
#include <iomanip>
#include <stack>
using namespace std;
struct canh{ //biểu diễn một cạnh của đồ thị
        int dau; //đỉnh đầu của cạnh
```

```
        int cuoi; //đỉnh cuối của cạnh
};
class Graph{ //xây dựng lớp đồ thị
  private:
        int V;    // số đỉnh của đồ thị
        list<int> *adj;    // con trỏ đến mảng các danh sách kề
        bool *chuaxet; //mảng chưa xét
        canh *T;//tập cạnh của cây khung
        int sc; //số cạnh của cây khung
 public:
        Graph(int V);  // constructor của lớp
        void addEdge(int v, int w); // thêm một cạnh vào đồ thị
        void Tree_BFS(int u);//thuật toán Tree-BFS
        void Tree_DFS(int u);//thuật toán Tree-DFS
};
Graph::Graph(int V){ //constructor của lớp
        this->V = V;//thiết lập tập đỉnh
        adj = new list<int>[V];//thiết lập V danh sách kề
        T       = new canh[V]; sc =1; //thiết lập số cạnh cây khung
        chuaxet = new bool[V];//thiết lập giá trị mảng chưa xét
        for(int u=0; u<V; u++)       chuaxet[u]=true;
}
void Graph::addEdge(int v, int w){ //thêm một cạnh vào danh sách kề
        adj[v].push_back(w); // thêm w vào list(v)
        adj[w].push_back(v); // thêm v vào list(w)
}
void Graph::Tree_BFS(int u){//thuật toán Tree-BFS
        list<int> queue;//tạo hàng đợi rỗng
        //Bước 1: Khởi tạo
        queue.push_back(u);//đưa u vào hàng đợi
        chuaxet[u] = false;//thiết lập trạng thái đỉnh u
        list<int>::iterator t; //t là iterator của list
        //Bước 2 (lặp):
        while(!queue.empty()){//lặp đến khi hàng đợi rỗng
                int s = queue.front();//lấy s ở đầu hàng đợi
                queue.pop_front();//loại s ra khỏi hàng đợi
                for(t = adj[s].begin(); t != adj[s].end(); ++t){//duyệt t∈list(s)
```

```
                        if(chuaxet[*t]){//nếu đỉnh t chưa xét
                                queue.push_back(*t);//đưa t vào hàng đợi
                                chuaxet[*t] = false;//ghi nhận t đã xét
                                T[sc].dau = s; T[sc].cuoi = *t; //kết nạp cạnh (s,t)
                                sc++; //tăng số cạnh lên 1
                        }
                }
        }
        if (sc<V-1)    cout<<"Đồ thị không liên thông"<<endl;
        else {
                cout<<"\n Tập cạnh cây khung:"<<endl;
                for(int i=1; i<sc; i++) cout<<T[i].dau<<setw(3)<<T[i].cuoi<<endl;
        }
}
int main(void){
        ifstream fp("Graph.in"); //mở file để đọc
        int n, m, dau, cuoi;   fp>>n>>m;//đọc số đỉnh và số cạnh của đồ thị
        Graph gh(n+1);//thiết lập đồ thị gồm n đỉnh
        for(int i=1; i<=m; i++){//chuyển đồ thị sang danh sách kề
                fp>>dau>>cuoi;  gh.addEdge(dau, cuoi);
        }
        fp.close(); gh.printGraph(); gh.Tree_BFS(1);
}
```

6.5.3. Xây dựng cây khung nhỏ nhất của đồ thị bằng thuật toán Kruskal

Bài toán tìm cây khung nhỏ nhất là một trong những bài toán tối ưu trên đồ thị có ứng dụng trong nhiều lĩnh vực khác nhau của thực tế. Bài toán được phát biểu như dưới sau. Cho $G=<V, E>$ là đồ thị vô hướng liên thông với tập đỉnh $V = \{1, 2, \ldots, n\}$ và tập cạnh E gồm m cạnh. Mỗi cạnh e của đồ thị được gán với một số không âm $c(e)$ được gọi là độ dài cạnh. Giả sử $H=<V, T>$ là một cây khung của đồ thị G. Ta gọi độ dài $c(H)$ của cây khung H là tổng độ dài các cạnh: $c(H) = \sum_{e \in T} c(e)$. Bài toán được đặt ra là, trong số các cây khung của đồ thị hãy tìm cây khung có độ dài nhỏ nhất của đồ thị.

a) Biểu diễn thuật toán

Thuật toán Kruskal xây dựng cây khung nhỏ nhất cho đồ thị vô hướng liên thông có trọng số được thực hiện thep mô hình tham lam trong Hình 5.16.

Thuật toán Kruskal:
Begin
Bước 1 (Khởi tạo):
T = ∅; //*Khởi tạo tập cạnh cây khung là ∅*
d(H) = 0; //*Khởi tạo độ dài nhỏ nhất cây khung là 0*
Bước 2 (Sắp xếp):
<Sắp xếp các cạnh của đồ thị theo thứ tự giảm dần của trọng số>;
Bước 3 (Lặp):
while (|T<n-1| && E≠∅) do { // *Lặp nếu E≠∅ và |T|<n-1*
e = <Cạnh có độ dài nhỏ nhất>;
E = E \ {e}; //*Loại cạnh e ra khỏi đồ thị*
if (T∪{e} không tạo nên chu trình) then {
T = ∪{e}; // *Kết nạp e vào tập cạnh cây khung*
d(H) = d(H) + d(e); //*Độ dài của tập cạnh cây khung*
endif;
endwhile;
Bước 4 (Trả lại kết quả):
if (|T| < n-1) then <Đồ thị không liên thông>;
else
Return(T, d(H));
end.

Hình 5.16. Thuật toán Kruskal tìm cây khung nhỏ nhất

b) Độ phức tạp thuật toán

Độ phức tạp thuật toán là O(E.log(V)), vơi E là số cạnh và V là số đỉnh của đồ thị. Bạn đọc tự tìm hiểu và chứng minh trong các tài liệu tham khảo liên quan.

c) Kiểm nghiệm thuật toán

Bạn đọc tự tìm hiểu phương pháp kiểm nghiệm thuật toán trong các tài liệu liên quan.

d) Cài đặt thuật toán

```
#include <iostream>
#include <fstream>
using namespace std;
struct Edge {//biểu diễn cạnh của đồ thị
    int src; //đỉnh đầu của cạnh
    int dest; //đỉnh cuối của cạnh
    int weight; //trọng số của cạnh
};
struct Graph {//biểu diễn đồ thị vô hướng
```

```
        int V, E; //số đỉnh, số cạnh của đồ thị
        Edge* edge; //mảng lưu trữ các cạnh của đồ thị
};
Graph* createGraph(int V, int E){ //tạo lập đồ thị G=<V,E>
        Graph* graph = new Graph;//cấp phát miền nhớ cho đồ thị
        graph->V = V; //thiết lập số đỉnh của đồ thị
        graph->E = E; //thiết lập số cạnh của đồ thị
        graph->edge =  new Edge[E]; //cấp phát miền nhớ lưu trữ E cạnh
        return graph; //trả lại đồ thị được tạo ra
}
struct subset { //định nghĩa cấu trúc để hợp
        int parent; //node cha
        int rank; //node thứ cấp
};
int find( subset subsets[], int i){//tìm đường đi đến đỉnh i trên subset
        if (subsets[i].parent != i) //nếu parent không phải cha của i
                subsets[i].parent= find(subsets, subsets[i].parent); //lần tiếp từ parent
   return subsets[i].parent;//trả lại node cha của i
}
void Union(subset subsets[], int x, int y){// hợp cạnh (x,y)
        int xroot = find(subsets, x);          int yroot = find(subsets, y);
        if (subsets[xroot].rank < subsets[yroot].rank)
                subsets[xroot].parent = yroot;
        else if (subsets[xroot].rank > subsets[yroot].rank)
                subsets[yroot].parent = xroot;
        else {
                subsets[yroot].parent = xroot;
                subsets[xroot].rank++;
        }
}
void sort(Graph* graph){//sắp xếp theo thứ tự tăng dần trọng số các cạnh
        int E = graph->E-1;
        for(int i=1; i<E; i++){
                for(int j=i+1; j<=E; j++){
                        if(graph->edge[i].weight>graph->edge[j].weight){
                                Edge temp = graph->edge[i];
                                graph->edge[i]=graph->edge[j];
```

```
                        graph->edge[j]=temp;
                    }
                }
            }
        }
void KruskalMST( Graph* graph){//thuật toán Kruskal
        int V = graph->V; //tập đỉnh của đồ thị
        Edge result[V];  // tập cạnh của cây khung nhỏ nhất
        int e = 1, int i = 1;  //số cạnh của cây khung
        int FOPT =0;//độ dài ban đầu của cây khung
        //Bước 1. Sắp xếp theo thứ tự tăng dần trọng số các cạnh
        sort(graph);
        subset *subsets =new subset[V];//cấp phát miền nhớ cho tập cạnh
        for (int v = 1; v <= V; ++v) {//tạo các tập con của v
                subsets[v].parent = v;
                subsets[v].rank = 0;
        }
          //Bước 2. Lặp khi e< V-1
        while (e < V - 1){
                Edge next_edge = graph->edge[i++]; //chọn cạnh có độ dài nhỏ nhất
                int x = find(subsets, next_edge.src);//tìm node cha của src
                int y = find(subsets, next_edge.dest); //tìm node cha của dest
                if (x != y) {//nếu không tạo nên chu trình
                        result[e++] = next_edge;//thêm cạnh vào cây khung
                        FOPT = FOPT + next_edge.weight;//độ dài tăng thêm weight
                        Union(subsets, x, y);//hợp x, y vào subset
                }
       }
        //Bước 3. Trả lại kết quả
        if(e<V-1){
                cout<<"\n Đồ thị không liên thông";
        }
        else {
                cout<<"\n Độ dài nhỏ nhất:"<<FOPT;
                cout<<"\n Tập cạnh cây khung:";
                for (i = 1; i < e; ++i){
                   printf("\n%d%3d%3d",result[i].src,result[i].dest,result[i].weight);
```

```
                }
        }
}
int main(){
        int V ;  // số đỉnh của đồ thị
        int E ;  // số cạnh của đồ thị
        ifstream fp("dothi.in"); //đọc đồ thị dưới dạng danh sách cạnh
        fp>>V>>E;
        printf("\n Số đỉnh đồ thị:%d", V);
        printf("\n Số cạnh đồ thị:%d", E);
        Graph* graph = createGraph(V+1, E+1); //tạo lập G=<V,E>
        for(int i=1; i<=E; i++){
                fp>>(graph->edge[i].src);
                fp>>(graph->edge[i].dest);
                fp>>(graph->edge[i].weight);
        }
        fp.close();
        KruskalMST(graph);
}
```

6.5.4. Xây dựng cây khung nhỏ nhất của đồ thị bằng thuật toán PRIM

Thuật toán Kruskal làm việc kém hiệu quả đối với những đồ thị có số cạnh khoảng m=n(n-1)/2. Trong những tình huống như vậy, thuật toán Prim tỏ ra hiệu quả hơn. Thuật toán Prim còn được mang tên là người láng giềng gần nhất. Trong thuật toán này, bắt đầu tại một đỉnh tuỳ ý *s* của đồ thị, nối *s* với đỉnh *y* sao cho trọng số cạnh *graph[s, y]* là nhỏ nhất. Tiếp theo, từ đỉnh *s* hoặc *y* tìm cạnh có độ dài nhỏ nhất, điều này dẫn đến đỉnh thứ ba *z* và ta thu được cây bộ phận gồm *3* đỉnh *2* cạnh. Quá trình được tiếp tục cho tới khi ta nhận được cây gồm *n-1* cạnh, đó chính là cây bao trùm nhỏ nhất cần tìm. Thuật toán Prim được mô tả trong Hình 5.17.

a) Biểu diễn thuật toán

Thuật toán PRIM (s):
Begin:
Bước 1 (Khởi tạo):
V_H = {s}; //Tập đỉnh cây khung thiết lập ban đầu là s
V = V\{s}; //Tập đỉnh V được bớt đi s
T = ∅; //Tập cạnh cây khung thiết lập ban đầu là ∅
d(H) = 0; //Độ dài cây khung được thiết lập là 0
Bước 2 (Lặp):
while (V≠ ∅) do {
e = <u, v>: cạnh có độ dài nhỏ nhất thỏa mãn u∈V, v∈V_H;
d(H) = d(H) + d(e); // Thiết lập đồ dài cây khung nhỏ nhất
T = T ∪ {e}; //Kết nạp e vào cây khung
V = V \{u}; // Tập đỉnh V bớt đi đỉnh u
V_H = V_H∪{u}; // Tập đỉnh V_H thêm vào đỉnh u
endwhile;
Bước 3 (Trả lại kết quả):
if (|T|<n-1) then <Đồ thị không liên thông>;
else Return(T, d(H));
End.

Hình 5.16. Thuật toán PRIM

b) Độ phức tạp thuật toán

Độ phức tạp thuật toán là $O(V^2)$, với V là số đỉnh của đồ thị. Bạn đọc tự tìm hiểu phương pháp chứng minh độ phức tạp thuật toán trong các tài liệu liên quan.

c) Kiểm nghiệm thuật toán

Bạn đọc tự tìm hiểu phương pháp kiểm nghiệm thuật toán trong các tài liệu liên quan.

d) Cài đặt thuật toán

```
#include <iostream>
#include <fstream>
#define MAX 100
using namespace std;
//Hàm tìm đỉnh có khoảng cách nhỏ nhất từ V đến V_T
int minKey(int key[], bool mstSet[], int n){
    int min = INT_MAX, min_index;
```

```
        for (int v = 1; v <= n; v++)
        if (mstSet[v] == false && key[v] < min){
                min = key[v];  min_index = v;
        }
        return min_index;
}

//xây dựng tập cây khung của đồ thị trong parent[]
int printMST(int parent[],  int graph[MAX][MAX], int n){
        printf("Edge   Weight\n");
        for (int i = 2; i <= n; i++)
                printf("%d - %d    %d \n", parent[i], i, graph[i][parent[i]]);
}
//thuật toán xây dựng cây khung nhỏ nhất sử dụng ma trận kề
void primMST(int graph[MAX][MAX], int n) {
        int parent[MAX]; // mảng lưu trữ các cạnh cây khung nhỏ nhất
        int key[MAX];   // giá trị để lấy cạnh có độ dài nhỏ nhất
         bool mstSet[MAX];  // tập đỉnh của cây khung & tập đỉnh chưa xét
         int FOPT =0; //độ dài nhỏ nhất cây khung
        for (int i = 1; i <= n; i++){ //thiết lập giá trị nhỏ nhất các đỉnh là INF
                key[i] = INT_MAX;
                mstSet[i] = false;
        }
        key[1] = 0;     // đỉnh đầu tiên là 1
        parent[1] = -1; //giá trị đầu tiên lấy là -1
        // Cây khung sẽ có V đỉnh count:
        for (int count = 0; count < n-1; count++)   {
                //lấy đỉnh u có key nhỏ nhất chưa có trong cây khung
                int u = minKey(key, mstSet,n);
                 mstSet[u] = true; //thêm u vào tập đỉnh cây khung
                //cập nhật giá trị các đỉnh kề chưa có trong cây khung
                for (int v = 1; v <= n; v++) {
                        if (graph[u][v] && !mstSet[v] && graph[u][v] <  key[v]){
                                parent[v]  = u;
                                 key[v] = graph[u][v], FOPT=FOPT+key[v];
                        }
                }
```

```
        }
        //Đưa ra cây khung nhỏ nhất
        printf("\n Độ dài cây khung nhỏ nhất:%d\n", FOPT);
        printMST(parent, graph, n);
}
int main(){
        int n, m, dau, cuoi, w;//n, m: số đỉnh, số cạnh của đồ thị;
        int graph[MAX][MAX]; //ma trận kề biểu diễn đồ thị
        ifstream fp("dothi1.in"); //file được biểu diễn dưới dạng danh sách cạnh
        fp>>n>>m; //đọc số đỉnh và số cạnh của đồ thị
        for (int i=1; i<=m; i++){//đọc m cạnh tiếp theo
                fp>>dau>>cuoi>>w; //đọc cạnh (dau, cuoi, trongso)
                graph[dau][cuoi]=graph[cuoi][dau]=w;
        }
        fp.close();
        primMST(graph,n);
}
```

6.6. Bài toán tìm đường đi ngắn nhất

Xét đồ thị $G=<V, E>$; trong đó $|V| = n$, $|E| = m$. Với mỗi cạnh $(u, v)\in E$, ta đặt tương ứng với nó một số thực A[u][v] được gọi là trọng số của cạnh. Ta sẽ đặt $A[u,v]=\infty$ nếu $(u, v)\notin E$. Nếu dãy $v_0, v_1, \ldots, v_k$ là một đường đi trên G thì $\sum_{i=1}^{p} A[v_{i-1}, v_i]$ được gọi là độ dài của đường đi.

Bài toán tìm đường đi ngắn nhất trên đồ thị dưới dạng tổng quát có thể được phát biểu dưới dạng sau: tìm đường đi ngắn nhất từ một đỉnh xuất phát $s\in V$ (đỉnh nguồn) đến đỉnh cuối $t\in V$ (đỉnh đích). Đường đi như vậy được gọi là đường đi ngắn nhất từ *s* đến *t*, độ dài của đường đi *d(s,t)* được gọi là khoảng cách ngắn nhất từ *s* đến *t* (trong trường hợp tổng quát *d(s,t)* có thể âm). Nếu như không tồn tại đường đi từ *s* đến *t* thì độ dài đường đi $d(s,t)=\infty$. Dưới đây là một số thể hiện cụ thể của bài toán.

Trường hợp 1. Nếu *s* cố định và *t* thay đổi, khi đó bài toán được phát biểu dưới dạng tìm đường đi ngắn nhất từ *s* đến tất cả các đỉnh còn lại trên đồ thị. Đối với đồ thị có trọng số không âm, bài toán luôn có lời giải bằng thuật toán Dijkstra. Đối với đồ thị có trọng số âm nhưng không tồn tại chu trình âm, bài toán có lời giải bằng thuật toán Bellman-Ford. Trong trường hợp đồ thị có chu trình âm, bài toán không có lời giải.

Trường hợp 2. Nếu s thay đổi và t cũng thay đổi, khi đó bài toán được phát biểu dưới dạng tìm đường đi ngắn nhất giữa tất cả các cặp đỉnh của đồ thị. Bài toán luôn có lời giải trên đồ thị không có chu trình âm. Đối với đồ thị có trọng số không âm, bài toán được giải quyết bằng cách thực hiện lặp lại n lần thuật toán Dijkstra. Đối với đồ thị không có chu trình âm, bài toán có thể giải quyết bằng thuật toán Floyd-Warshall.

6.6.1. Thuật toán Dijkstra

Thuật toán tìm đường đi ngắn nhất từ đỉnh *s* đến các đỉnh còn lại được Dijkstra đề nghị áp dụng cho trường hợp đồ thị có hướng với trọng số không âm. Thuật toán được thực hiện trên cơ sở gán tạm thời cho các đỉnh. Nhãn của mỗi đỉnh cho biết cận trên của độ dài đường đi ngắn nhất tới đỉnh đó. Các nhãn này sẽ được biến đổi (tính lại) nhờ một thủ tục lặp, mà ở mỗi bước lặp một số đỉnh sẽ có nhãn không thay đổi, nhãn đó chính là độ dài đường đi ngắn nhất từ s đến đỉnh đó. Thuật toán Dijkstra tìm đường đi ngắn nhất từ s đến tất cả các đỉnh còn lại của đồ thị được mô tả chi tiết trong Hình 5.17.

a) Biểu diễn thuật toán

```
Thuật toán Dijkstra (s): //s∈ V là một đỉnh bất kỳ của G = <V,E>
Begin
    Bước 1 (Khởi tạo):
        d[s]=0; //Gán nhãn của đỉnh s là 0
        T = V\{s}; // T là tập đỉnh có nhãn tạm thời
        for each  v∈ V do { //Sử dụng s gán nhãn cho các đỉnh còn lại
            d[v] = A[s,v];
            truoc[v]=s;
        endfor;
    Bước 2 (Lặp):
        while (T≠ ∅ ) do {
            Tìm đỉnh u∈T sao cho d[u] = min { d[z] | z∈T};
            T= T\{u}; //cố định nhãn đỉnh u
            for each v∈ T do { //Sử dụng u, gán nhãn lại cho các đỉnh
                if ( d[v] > d[u] + A[u, v] ) then {
                    d[v] =  d[u] + A[u, v]; //Gán lại nhãn cho đỉnh v;
                    truoc[v] = u;
                endif;
            endfor;
        endwhlie;
    Bước 3 (Trả lại kết quả):
        Return (d[s], truoc[s]);
End.
```

Hình 5.17. Thuật toán Dijkstra

b) Độ phức tạp thuật toán

Độ phức tạp thuật toán là $O(V^2)$, trong đó V là số đỉnh của đồ thị. Bạn đọc tự tìm hiểu và chứng minh độ phức tạp thuật toán Dijkstra trong các tài liệu liên quan.

c) Thử nghiệm thuật toán

Bạn đọc tự tìm hiểu phương pháp thử nghiệm thuật toán Dijkstra trong các tài liệu liên quan.

d) Cài đặt thuật toán

```
#include <iostream>
#include <fstream>
#define MAX 100
using namespace std;
//hàm tìm đỉnh có nhãn nhỏ nhất
int minDistance(int dist[], bool sptSet[], int n){
        // thiết lập nhãn nhỏ nhất ban đầu
        int min = INT_MAX, min_index;
        for (int v = 1; v <= n; v++){
                if (sptSet[v] == false && dist[v] <= min){
                        min = dist[v]; min_index = v;
                }
        }
        return min_index;//đỉnh có nhãn nhỏ nhất
}
// hàm in ra độ dài đường đi ngắn nhất
int printSolution(int dist[], int n){
        printf("Đường đi ngắn: \n");
        for (int i = 1; i <= n; i++)
                printf("%d \t\t %d\n", i, dist[i]);
}

// thuật toán Dijkstra sử dụng ma trận kề
void dijkstra(int graph[MAX][MAX], int src, int n) {
        int dist[MAX];     // dist[i] sẽ là độ dài đường đi ngắn nhất từ src đến i
        bool sptSet[MAX]; //lưu trữ đường đi ngắn nhất từ src đến i
        //Bước 1 (khởi tạo): thiết lập tất cả độ dài là INF.
        for (int i = 1; i <= n; i++){
                dist[i] = INT_MAX;
                 sptSet[i] = false;
        }
        dist[src] = 0; //nhãn đỉnh xuất phát lấy là 0

        //Bước 2(lặp): tìm đường đi ngắn nhất từ src đến tất cả các đỉnh
        for (int count = 1; count <= n-1; count++){ //lặp trên tập đỉnh V
                int u = minDistance(dist, sptSet,n);//lấy u là đỉnh có nhãn nhỏ nhất
                sptSet[u] = true; //đánh dấu đỉnh u đã là đường đi ngắn nhất
                 // cập nhật nhãn các đỉnh còn lại
                for (int v = 1; v <= n; v++) {
                        if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX
                            && dist[u]+graph[u][v] < dist[v])
                                dist[v] = dist[u] + graph[u][v];
```

```
                }
        }
        printSolution(dist, n); // in ra đường đi ngắn nhất
}
int main() {
        int n, m, dau, cuoi, trongso; //khai báo số đỉnh và số cạnh: n, m
        int graph[MAX][MAX]; //ma trận kề biểu diễn đồ thị
        ifstream fp("dothiw.in"); //file biểu diễn đồ thị dưới dạng danh sách cạnh
        fp>>n>>m; //đọc số cạnh và số đỉnh của đồ thị
        for(int i=1; i<=m; i++){ //đọc m cạnh: (dau, cuoi, trongso)
                fp>>dau>>cuoi>>trongso;
                graph[dau][cuoi]= trongso;
                graph[cuoi][dau]= trongso;//bỏ đi nếu là đồ thị có hướng
        }
        fp.close();
        dijkstra(graph, 1, 9);
}
```

6.6.2. Thuật toán Bellman-Ford

Thuật toán Bellman-Ford dùng để tìm đường đi ngắn nhất trên đồ thị không có chu trình âm. Do vậy, trong khi thực hiện thuật toán Bellman-Ford ta cần kiểm tra đồ thị có chu trình âm hay không. Trong trường hợp đồ thị có chu trình âm, bài toán sẽ không có lời giải. Thuật toán được thực hiện theo k = n - 2 vòng lặp trên tập đỉnh hoặc tập cạnh tùy thuộc vào dạng biểu diễn của đồ thị. Nếu đồ thị được biểu diễn dưới dạng ma trận kề, độ phức tạp thuật toán là $O(V^3)$, với V là số đỉnh của đồ thị. Trong trường hợp đồ thị được biểu diễn dưới dạng danh sách cạnh, độ phức tạp thuật toán là O(V.E), với V là số đỉnh của đồ thị, E là số cạnh của đồ thị. Thuật toán được mô tả chi tiết trong Hình 5.18 cho đồ thị biểu diễn dưới dạng ma trận kề.

a) Biểu diễn thuật toán

```
Thuật toán Bellman-Ford (s): //s∈ V là đỉnh bất kỳ của đồ thị
Begin:
    Bước 1 (Khởi tạo):
        for v∈V do { //Sử dụng s gán nhãn cho các đỉnh v∈V
            D[v] = A[s][v];
            Truoc[v] = s;
        }
    Bước 2 (Lặp) :
        D[s] = 0; K=1;
        while (K<=N-2 ) { //N-2 vòng lặp
            for v∈V\{s} do { //Lấy mỗi đỉnh v∈V\s
                for u∈V do { //Gán nhãn cho v
                    if (D[v] > D[u] + A[u][v] ) {
                        D[v]= D[u] + A[u][v];
                        Truoc[v] = u;
                    endif;
                endfor;
            endfor;
        endwlie;
    Bước 3 (Trả lại kết quả):
        Return( D[v], Truoc[v]: v∈U);
End.
```

Hình 5.18. Thuật toán Bellman-Ford

b) Độ phức tạp thuật toán

Độ phức tạp thuật toán là O(VE), trong đó V, E là số đỉnh và số cạnh của đồ thị. Bạn đọc tự tìm hiểu và chứng minh độ phức tạp thuật toán Bellman-Ford trong các tài liệu liên quan.

c) Thử nghiệm thuật toán

Bạn đọc tự tìm hiểu phương pháp thử nghiệm thuật toán Bellman-Ford trong các tài liệu liên quan.

d) Cài đặt thuật toán

```
#include <iostream.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#define MAX  100
#define  MAXC 10000
```

```
int C[MAX][MAX]; //Ma tran trong so bieu dien do thi
int D[MAX];      //Do dai duong di
int Trace[MAX];   //Luu lai vet duong di
int n, m, S, F;     // n:So dinh; S: Dinh bat dau; F: Dinh ket thuc
FILE *fp;
void Read_Data(void){
        int i, u, v;fp = fopen("dothi.in","r");
        fscanf(fp,"%d%d%d%d",&n,&m,&S,&F);
        for(u=1; u<=n; u++)
                for(v=1; v<=n; v++)
                        if (u==v) C[u][v]=0;
                                else C[u][v]=MAXC;
         for(i=1; i<=m; i++)
                fscanf(fp,"%d%d%d",&u,&v,&C[u][v]);
        fclose(fp);
}
void Init(void){
        int i;
        for( i=1; i<=n; i++){
                D[i] = C[S][i];
                Trace[i]=S;
        }
}
void Result(void){
        if (D[F]==MAXC) printf("\n Khong co duong di");
        else {
                printf("\n Do dai %d den %d: %d", S, F, D[F]);
                while (F!=S ){
                        printf("%d <--",F);
                        F = Trace[F];
                }
        }
}
void Ford_Bellman(void){
        int k, u, v;D[S]=0;
        for( k=1; k<=n-2; k++){
                for(v=1; v<=n; v++){
                        for( u=1; u<=n; u++){
                                if (D[v]>D[u]+C[u][v]){
                                        D[v] = D[u]+C[u][v];
                                        Trace[u]=v;
                                }
                        }
```

```
                }
            }
    }
    int main() {
        Read_Data();Init();
        Ford_Bellman(); Result();
        system("PAUSE");
        return 0;
    }
```

e) Thuật toán Bellman-Ford cho đồ thị biểu diễn dưới dạng danh sách cạnh

```
#include <iostream>
#include <fstream>
using namespace std;
struct Edge{ //biểu diễn cạnh trọng số của đồ thị
        int src; //đỉnh đầu của cạnh
        int  dest; //đỉnh cuối của cạnh
        int  weight; //trọng số của cạnh
};
struct Graph{ //biểu diễn đồ thị bằng danh sách cạnh
        int V, E; //số đỉnh và số cạnh của đồ thị
        Edge* edge;//danh sách các cạnh của đồ thị
};
struct Graph* createGraph(int V, int E){//tạo đồ thị G=<V,E>
        Graph* graph = new Graph; //graph là con trỏ đến Graph
        graph->V = V; //thiết lập số đỉnh là V
        graph->E = E; //thiết lập số cạnh là E
        graph->edge = new Edge[E]; //E cạnh được lưu trữ trong Edge[E]
        return graph; //đồ thị được tạo ra
}
void printArr(int dist[], int n) {//hàm đưa ra độ dài đường đi ngắn nhất
        printf("Đường đi ngắn nhất\n");
        for (int i = 1; i < n; ++i)
                printf("%d \t\t %d\n", i, dist[i]);
}
 void BellmanFord( Graph* graph, int src){ //thuật toán Bellman-Ford
        int V = graph->V; //V là tập đỉnh
        int E = graph->E; //E là tập cạnh
        int dist[V]; //khoảng cách ngắn nhất
```

```
        // Bước 1 (khởi tạo): thiết đường đi ngắn nhất lúc đầu là INF
        for (int i = 1; i < V; i++)
                dist[i]   = INT_MAX;
        dist[src] = 0; //nhãn đỉnh xuất phát là 0
        // Bước 2 (lặp): lặp V-2 lần trên tập các cạnh
        for (int i = 2; i <= V-1; i++)    { //V-2 bước lặp i
                for (int j = 1; j < E; j++) {//trên tập E cạnh
                        int u = graph->edge[j].src;
                        int v = graph->edge[j].dest;
                        int weight = graph->edge[j].weight;
                        if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
                                dist[v] = dist[u] + weight;
                }
        }
        // Bước 3: kiểm tra chu trình âm
        for (int i = 1; i < E; i++)   {
                int u = graph->edge[i].src;
                int v = graph->edge[i].dest;
                int weight = graph->edge[i].weight;
                if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
                        printf("Đồ thị có chu trình âm"); return;
                }
        }
        printArr(dist, V);
}
int main() {
        int V, E;  // số đỉnh và số cạnh của đồ thị
        int dau, cuoi, trongso;
        ifstream fp("dothich.in"); //mở file đồ thị dưới dạng danh sách cạnh
        fp>>V>>E;V=V; E=E; //đọc số đỉnh và số cạnh
        Graph* graph = createGraph(V+1, E+1); //tạo đồ thị G=<V,E>
        for(int i=1; i<=E; i++){//đọc E cạnh tiếp theo
                fp>>dau>>cuoi>>trongso;
                graph->edge[i].src = dau;
                graph->edge[i].dest = cuoi;
                graph->edge[i].weight = trongso;
        }
```

```
        fp.close();
        BellmanFord(graph, 2);
    }
```

6.6.3. Thuật toán Floyd-Warshall

Để tìm đường đi ngắn nhất giữa tất cả các cặp đỉnh của đồ thị, chúng ta có thể sử dụng *V* lần thuật toán *Ford_Bellman* hoặc *Dijkstra* (trong trường hợp trọng số không âm). Tuy nhiên, trong cả hai thuật toán được sử dụng đều có độ phức tạp tính toán lớn (chí ít là $O(V^3)$). Trong trường hợp tổng quát, người ta thường dùng thuật toán *Floyd-Warshall*. Thuật toán Floy được mô tả chi tiết trong Hình 5.19.

a) Biểu diễn thuật toán

Thuật toán Floyd-Warshall:
Begin:

Bước 1 (Khởi tạo):

```
for (i=1; i≤ n; i++) {
    for (j =1; j≤ n; j++) {
        d[i,j] = a[i, j];
        p[i,j] = i;
    }
}
```

Bước 2 (lặp) :

```
for (k=1; k≤ n; k++) {
    for (i=1; i≤ n; i++){
        for (j =1; j≤ n; j++) {
            if (d[i,j] > d[i, k] + d[k, j]) {
                d[i, j] = d[i, k] + d[k, j];
                p[i,j] = p[k, j];
            }
        }
    }
}
```

Bước 3 (Trả lại kết quả):

```
Return (p([i,j], d[i,j]: i, j∈V);
```

Hình 5.19. Thuật toán Floyd-Warshall.

b) Độ phức tạp thuật toán

Độ phức tạp thuật toán là $O(V^3)$, trong đó V là số đỉnh của đồ thị. Bạn đọc tự tìm hiểu và chứng minh độ phức tạp thuật toán Floyd-Warshall trong các tài liệu liên quan.

c) Kiểm nghiệm thuật toán

Bạn đọc tự tìm hiểu phương pháp kiểm nghiệm thuật toán Floyd-Warshall trong các tài liệu liên quan.

d) Cài đặt thuật toán

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#define MAX 10000
#define	TRUE 1
#define FALSE	0
int A[50][50], D[50][50], S[50][50];
int n, u, v, k;FILE *fp;
void Init(void){
	int i, j, k;
	fp=fopen("FLOY.IN","r");
	if(fp==NULL){
		printf("\n Khong co file input");
		getch(); return;
	}
	for(i=1; i<=n; i++)
		for(j=1; j<=n; j++)
			A[i][j]=0;
	fscanf(fp,"%d%d%d",&n,&u,&v);
	printf("\n So dinh do thi:%d",n);
	printf("\n Di tu dinh:%d den dinh %d:",u,v);
	printf("\n Ma tran trong so:");
	for(i=1; i<=n; i++){
		printf("\n");
		for(j=1; j<=n; j++){
			fscanf(fp,"%d", &A[i][j]);
			printf("%5d",A[i][j]);
			if(i!=j && A[i][j]==0)
				A[i][j]=MAX;
		}
	}
	fclose(fp);getch();
}
void Result(void){
	if(D[u][v]>=MAX) {
		printf("\n Khong co duong di");
```

```
                getch(); return;
        }
        else {
                printf("\n Duong di ngan nhat:%d", D[u][v]);
                printf("\n Dinh %3d", u);
                while(u!=v) {
                        printf("%3d",S[u][v]);
                        u=S[u][v];
                }
        }
}
void Floy(void){
        int i, j,k, found;
        for(i=1; i<=n; i++){
                for(j=1; j<=n;j++){
                        D[i][j]=A[i][j];
                        if (D[i][j]==MAX) S[i][j]=0;
                        else S[i][j]=j;
                }
        }
        /* Mang D[i,j] la mang chua cac gia tri khoan cach ngan nhat tu i den j
        Mang S la mang chua gia tri phan tu ngay sau cua i tren duong di
        ngan nhat tu i->j  */
        for (k=1; k<=n; k++){
                for (i=1; i<=n; i++){
                        for (j=1; j<=n; j++){
                                if (D[i][k]!=MAX && D[i][j]>(D[i][k]+D[k][j]) ){
                                // Tim D[i,j] nho nhat co the co
                                        D[i][j]=D[i][k]+D[k][j];
                                        S[i][j]=S[i][k];
                                        //ung voi no la gia tri cua phan tu ngay sau i
                                }
                        }
                }
        }
}
void main(void){
        clrscr();Init();
        Floy();Result();
}
```

BÀI TẬP

BÀI 1. DFS TRÊN ĐỒ THỊ VÔ HƯỚNG

Cho đồ thị vô hướng G=<V, E> được biểu diễn dưới dạng danh sách cạnh. Hãy viết thuật toán duyệt theo chiều sâu bắt đầu tại đỉnh u∈V (DFS(u)=?)

Input:

- Dòng đầu tiên đưa vào T là số lượng bộ test.
- Những dòng tiếp theo đưa vào các bộ test. Mỗi bộ test gồm |E| +1 dòng: dòng đầu tiên đưa vào ba số |V|, |E| tương ứng với số đỉnh và số cạnh của đồ thị, và u là đỉnh xuất phát; |E| dòng tiếp theo đưa vào các bộ đôi u∈V, v∈V tương ứng với một cạnh của đồ thị.
- T, |V|, |E| thỏa mãn ràng buộc: $1 \le T \le 200$; $1 \le |V| \le 10^3$; $1 \le |E| \le |V|(|V|-1)/2$;

Output:

- Đưa ra danh sách các đỉnh được duyệt theo thuật toán DFS(u) của mỗi test theo khuôn dạng của ví dụ dưới đây.

Ví dụ:

Input:	Output:
1 6 9 5 1 2 1 3 2 3 2 4 3 4 3 5 4 5 4 6 5 6	5 3 1 2 4 6

BÀI 2. BFS TRÊN ĐỒ THỊ VÔ HƯỚNG

Cho đồ thị vô hướng G=<V, E> được biểu diễn dưới dạng danh sách cạnh. Hãy viết thuật toán duyệt theo chiều rộng bắt đầu tại đỉnh u∈V (BFS(u)=?)

Input:

- Dòng đầu tiên đưa vào T là số lượng bộ test.
- Những dòng tiếp theo đưa vào các bộ test. Mỗi bộ test gồm 2 dòng: dòng đầu tiên đưa vào ba số |V|, |E|, u∈V tương ứng với số đỉnh, số cạnh và đỉnh bắt đầu duyệt; Dòng tiếp theo đưa vào các bộ đôi u∈V, v∈V tương ứng với một cạnh của đồ thị.
- T, |V|, |E| thỏa mãn ràng buộc: $1 \le T \le 200$; $1 \le |V| \le 10^3$; $1 \le |E| \le |V|(|V|-1)/2$;

Output:

- Đưa ra danh sách các đỉnh được duyệt theo thuật toán BFS(u) của mỗi test theo khuôn dạng của ví dụ dưới đây.

Ví dụ:

Input:	**Output:**

1 6 9 1 1 2 1 3 2 3 2 5 3 4 3 5 4 5 4 6 5 6	1 2 3 5 4 6

BÀI 3. TÌM ĐƯỜNG ĐI THEO DFS VỚI ĐỒ THỊ VÔ HƯỚNG

Cho đồ thị vô hướng G=<V, E> được biểu diễn dưới dạng danh sách cạnh. Hãy tìm đường đi từ đỉnh s∈V đến đỉnh t∈V trên đồ thị bằng thuật toán DFS.

Input:

- Dòng đầu tiên đưa vào T là số lượng bộ test.
- Những dòng tiếp theo đưa vào các bộ test. Mỗi bộ test gồm 2 dòng: dòng đầu tiên đưa vào bốn số |V|, |E|, s∈V, t∈V tương ứng với số đỉnh, số cạnh, đỉnh u, đỉnh v; Dòng tiếp theo đưa vào các bộ đôi u∈V, v∈V tương ứng với một cạnh của đồ thị.
- T, |V|, |E| thỏa mãn ràng buộc: $1 \le T \le 100$; $1 \le |V| \le 10^3$; $1 \le |E| \le |V|(|V|-1)/2$;

Output:

- Đưa ra đường đi từ đỉnh s đến đỉnh t của mỗi test theo thuật toán DFS của mỗi test theo khuôn dạng của ví dụ dưới đây. Nếu không có đáp án, in ra -1.

Ví dụ:

Input:	**Output:**
1 6 9 1 6 1 2 1 3 2 3 2 5 3 4 3 5 4 5 4 6 5 6	1 2 3 4 5 6

BÀI 4. TÌM ĐƯỜNG ĐI THEO DFS VỚI ĐỒ THỊ CÓ HƯỚNG

Cho đồ thị có hướng G=<V, E> được biểu diễn dưới dạng danh sách cạnh. Hãy tìm đường đi từ đỉnh s∈V đến đỉnh t∈V trên đồ thị bằng thuật toán DFS.

Input:

- Dòng đầu tiên đưa vào T là số lượng bộ test.
- Những dòng tiếp theo đưa vào các bộ test. Mỗi bộ test gồm 2 dòng: dòng đầu tiên đưa vào bốn số |V|, |E|, s∈V, t∈V tương ứng với số đỉnh, số cạnh, đỉnh u, đỉnh v; Dòng tiếp theo đưa vào các bộ đôi u∈V, v∈V tương ứng với một cạnh của đồ thị.
- T, |V|, |E| thỏa mãn ràng buộc: $1 \le T \le 100$; $1 \le |V| \le 10^3$; $1 \le |E| \le |V|(|V|-1)/2$;

Output:

- Đưa ra đường đi từ đỉnh s đến đỉnh t của mỗi test theo thuật toán DFS của mỗi test theo khuôn dạng của ví dụ dưới đây. Nếu không có đáp án, in ra -1.

Ví dụ:

Input:	**Output:**
1 6 9 1 6 1 2 2 5 3 1 3 2 3 5 4 3 5 4 5 6 6 4	1 2 5 6

BÀI 5. ĐẾM SỐ THÀNH PHẦN LIÊN THÔNG VỚI DFS.

Cho đồ thị vô hướng G=<V, E> được biểu diễn dưới dạng danh sách cạnh. Hãy tìm số thành phần liên thông của đồ thị bằng thuật toán DFS.

Input:

- Dòng đầu tiên đưa vào T là số lượng bộ test.
- Những dòng tiếp theo đưa vào các bộ test. Mỗi bộ test gồm 2 dòng: dòng đầu tiên đưa vào hai số |V|, |E| tương ứng với số đỉnh và số cạnh; Dòng tiếp theo đưa vào các bộ đôi u∈V, v∈V tương ứng với một cạnh của đồ thị.
- T, |V|, |E| thỏa mãn ràng buộc: $1 \le T \le 100$; $1 \le |V| \le 10^3$; $1 \le |E| \le |V|(|V|-1)/2$;

Output:

- Đưa ra số thành phần liên thông của đồ thị bằng thuật toán DFS.

Ví dụ:

Input:	**Output:**
1 6 6 1 2 1 3 2 3 3 4 3 5 4 5	2

BÀI 6. LIỆT KÊ ĐỈNH TRỤ VỚI BFS

Cho đồ thị vô hướng liên thông G=<V, E> được biểu diễn dưới dạng danh sách cạnh. Sử dụng thuật toán BFS, hãy đưa ra tất cả các đỉnh trụ của đồ thị?

Input:

- Dòng đầu tiên đưa vào T là số lượng bộ test.
- Những dòng tiếp theo đưa vào các bộ test. Mỗi bộ test gồm 2 dòng: dòng đầu tiên đưa vào hai số |V|, |E| tương ứng với số đỉnh và số cạnh; Dòng tiếp theo đưa vào các bộ đôi u∈V, v∈V tương ứng với một cạnh của đồ thị.
- T, |V|, |E| thỏa mãn ràng buộc: $1 \le T \le 100$; $1 \le |V| \le 10^3$; $1 \le |E| \le |V|(|V|-1)/2$;

Output:

- Đưa ra danh sách các đỉnh trụ của mỗi test theo từng dòng.

Ví dụ:

Input:	**Output:**
1 5 5 1 2 1 3 2 3 2 5 3 4	2 3

BÀI 7. ĐƯỜNG ĐI HAMILTON

Đường đi đơn trên đồ thị có hướng hoặc vô hướng đi qua tất cả các đỉnh của đồ thị mỗi đỉnh đúng một lần được gọi là đường đi Hamilton. Cho đồ thị vô hướng G = <V, E>, hãy kiểm tra xem đồ thị có đường đi Hamilton hay không?

Input:

- Dòng đầu tiên đưa vào số lượng bộ test T.

- Những dòng kế tiếp đưa vào các bộ test. Mỗi bộ test gồm hai phần: phần thứ nhất đưa vào hai số V, E tương ứng với số đỉnh, số cạnh của đồ thị; phần thứ hai đưa vào các cạnh của đồ thị.
- T, V, E thỏa mãn ràng buộc: 1≤T ≤100; 1≤V≤10; 1≤ E ≤15.

Output:

- Đưa ra 1 hoặc 0 tương ứng với test có hoặc không có đường đi Hamilton theo từng dòng.

Ví dụ:

Input	Output
2 4 4 1 2 2 3 3 4 2 4 4 3 1 2 2 3 2 4	1 0

BÀI 8. CÂY KHUNG CỦA ĐỒ THỊ THEO THUẬT TOÁN BFS

Cho đồ thị vô hướng G=(V, E). Hãy xây dựng một cây khung của đồ thị G với đỉnh u ∈ V là gốc của cây bằng thuật toán BFS.

Input

Dòng đầu tiên gồm một số nguyên T (1 ≤ T ≤ 20) là số lượng bộ test.

Tiếp theo là T bộ test, mỗi bộ test có dạng sau:

- Dòng đầu tiên gồm 3 số nguyên N=|V|, M=|E|, u ($1 \le N \le 10^3$, $1 \le M \le 10^5$, 1 ≤ u ≤ N).
- M dòng tiếp theo, mỗi dòng gồm 2 số nguyên a, b (1 ≤ a, b ≤ N, a ≠ b) tương ứng cạnh nối hai chiều từ a tới b.
- Dữ liệu đảm bảo giữa hai đỉnh chỉ tồn tại nhiều nhất một cạnh nối.

Output

Với mỗi bộ test, nếu tồn tại cây khung thì in ra N – 1 cạnh của cây khung với gốc là đỉnh u trên N – 1 dòng theo thứ tự duyệt của thuật toán BFS. Ngược lại nếu không tồn tại cây khung thì in ra -1.

Ví dụ

Input	Output
2 4 4 2 1 2 1 3 2 4 3 4 4 2 2 1 2 3 4	2 1 2 4 1 3 -1

BÀI 9. KRUSKAL

Cho đồ thị vô hướng có trọng số G=<V, E, W>. Nhiệm vụ của bạn là hãy xây dựng một cây khung nhỏ nhất của đồ thị bằng thuật toán Kruskal.

Input:

- Dòng đầu tiên đưa vào số lượng bộ test T.
- Những dòng kế tiếp đưa vào các bộ test. Mỗi bộ test gồm hai phần: phần thứ nhất đưa vào hai số V, E tương ứng với số đỉnh và số cạnh của đồ thị; phần thứ 2 đưa vào E cạnh của đồ thị, mỗi cạnh là một bộ 3: đỉnh đầu, đỉnh cuối và trọng số của cạnh.
- T, S, D thỏa mãn ràng buộc: $1 \le T \le 100$; $1 \le V \le 100$; $1 \le E, W \le 1000$.

Output:

- Đưa ra kết quả mỗi test theo từng dòng.

Ví dụ:

Input	Output
2 3 3 1 2 5 2 3 3 1 3 1 2 1 1 2 5	4 5

BÀI 10. DIJKSTRA.

Cho đồ thị có trọng số không âm G=<V, E> được biểu diễn dưới dạng danh sách cạnh trọng số. Hãy viết chương trình tìm đường đi ngắn nhất từ đỉnh $u \in V$ đến tất cả các đỉnh còn lại trên đồ thị.

Input:

- Dòng đầu tiên đưa vào T là số lượng bộ test.
- Những dòng tiếp theo đưa vào các bộ test. Mỗi bộ test gồm |E|+1 dòng: dòng đầu tiên đưa vào hai ba số |V|, |E| tương ứng với số đỉnh và $u \in V$ là đỉnh bắt đầu; |E| dòng tiếp theo mỗi dòng đưa vào bộ ba $u \in V$, $v \in V$, w tương ứng với một cạnh cùng với trọng số canh của đồ thị.
- T, |V|, |E| thỏa mãn ràng buộc: $1 \le T \le 100$; $1 \le |V| \le 10^3$; $1 \le |E| \le |V|(|V|-1)/2$;

Output:

- Đưa ra kết quả của mỗi test theo từng dòng. Kết quả mỗi test là trọng số đường đi ngắn nhất từ đỉnh u đến các đỉnh còn lại của đồ thị theo thứ tự tăng dần các đỉnh.

Ví dụ:

Input:	Output:
1 9 12 1 1 2 4 1 8 8 2 3 8 2 8 11 3 4 7 3 6 4 3 9 2	0 4 12 19 21 11 9 8 14

4 5 9 4 6 14 5 6 10 6 7 2 6 9 6	

TÀI LIỆU THAM KHẢO

[1]. Nguyễn Đức Nghĩa., "*Cấu trúc dữ liệu và giải thuật*,". Nhà Xuất Bản KHKT Hà Nội, 2013.

[2]. Đỗ Xuân Lôi. "*Cấu trúc dữ liệu và giải thuật*,". Nhà Xuất Bản KHKT Hà Nội ,2009.

[3]. Donald E. Knuth., "The Art of Computer Programming", Addison-Wesley, Vol 1, 2, 3, 4, 5, 6.

[4]. Niklaus Wirth, Data Structures and Algorithms, Prentice Hall, 2004.

[5]. https://www.geeksforgeeks.org/

[6]. https://codeforces.com/