

BAN HỌC TẬP
CÔNG NGHỆ PHẦN MỀM



SỔ TAY KIẾN THỨC

CẤU TRÚC DỮ LIỆU
VÀ
GIẢI THUẬT



LỜI GIỚI THIỆU

Xin chào các bạn sinh viên thân mến,

Sau những tháng ngày hoạt động và đồng hành cùng mọi người qua rất nhiều mùa thi, chúng mình nhận thấy mọi người cần một nguồn tư liệu ngắn gọn, dễ hiểu nhưng phải đầy đủ. Chính vì vậy Ban học tập Đoàn khoa Công nghệ Phần mềm đã bắt tay biên soạn cuốn sách này, sách sẽ gồm những phần như: khái quát nội dung, trọng tâm chương trình học và đề thi mẫu kèm lời giải chi tiết nhất.

Đây là dự án mà Ban học tập Đoàn khoa Công nghệ Phần mềm đã ấp ủ từ rất lâu. Và với phương châm: "Dễ hiểu nhất và tường tận nhất", chúng mình hy vọng rằng cuốn sách này sẽ là trợ thủ đắc lực nhất cho các bạn sinh viên UIT trong việc học tập và giúp các bạn đạt thành tích cao nhất trong các kì thi.

Môn Cấu trúc dữ liệu và giải thuật (Data Structure and Algorithms - DSA) là một trong những kiến thức căn bản và quan trọng nhất. Nắm vững môn này là cơ sở để bạn thiết kế, xây dựng phần mềm, cũng như sử dụng các công cụ lập trình một cách hiệu quả. Những kiến thức về cấu trúc dữ liệu, thuật toán sẽ giúp bạn nâng cao khả năng làm việc của bạn và những lập trình viên có kiến thức lập trình tốt và nắm vững các cấu trúc dữ liệu hay thuật toán sẽ là điểm cộng để chinh phục tất cả các nhà tuyển dụng. Và đó là lý do Ban học tập Công nghệ Phần mềm chúng mình muốn hoàn thành một cuốn sổ tay có thể cung cấp đầy đủ kiến thức đến với mọi người.

Nếu sách có những điểm gì thắc mắc hãy liên hệ lại với chúng mình nhé!

Thông tin liên hệ của BHTCNPM:

Website: <https://www.bhtcnpm.com/>

Gmail: bht.cnpm.uit@gmail.com

Fanpage: <https://www.facebook.com/bhtcnpm>

Group BHT NNSC: <https://www.facebook.com/groups/bht.cnpm.uit>

Trân trọng cảm ơn các bạn đã quan tâm.



NGƯỜI BIÊN SOẠN

- 22521104 Trần Bảo Phú
- 22520254 Lê Hữu Độ
- 21520519 Lê Thanh Tuấn
- 22521697 Trương Đoàn Vũ
- Các thành viên và CTV của Ban học tập Đoàn khoa Công nghệ Phần mềm



MỤC LỤC

PHẦN 1. GIẢI THUẬT	3
Chương 1. Giới thiệu về giải thuật	3
1.1. Khái quát về giải thuật.....	3
1.2. Phân tích thuật toán.	3
1.3. Độ phức tạp của thuật toán.....	3
Chương 2. Thuật toán tìm kiếm	6
2.1. Định nghĩa về tìm kiếm.....	6
2.2. Tại sao chúng ta cần tìm kiếm?	6
2.3. Các thuật toán tìm kiếm cơ bản.....	6
2.4. Ví dụ	8
Chương 3. Các thuật toán sắp xếp	15
3.1. Sắp xếp chọn (Selection sort).....	15
3.2. Sắp xếp chèn (Insertion sort)	19
3.3. Sắp xếp nổi bọt (Bubble Sort)	23
3.4. Sắp xếp nhanh (Quick sort).....	27
3.5. Sắp xếp trộn (Merge sort)	30
3.6. Sắp xếp vun đống (Heap sort).....	33
PHẦN 2. CẤU TRÚC DỮ LIỆU	37
Chương 4. Danh sách liên kết đơn (Linked List)	37
4.1. Sơ lược về lịch sử ra đời Linked list	37
4.2. Nguyên Lý Hoạt Động của Linked.....	37
4.3. Định nghĩa Node và LIST	37
4.4. Các thao tác với danh sách liên kết đơn	38
Chương 5. Ngăn xếp (Stack)	40
5.1. Định nghĩa	40
5.2. Các thao tác cơ bản trên stack	40
5.3. Một số ứng dụng của stack.....	40



TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN – ĐHQG-HCM

BẢN HỌC TẬP CÔNG NGHỆ PHẦN MỀM

5.4. Các thao tác trên ngăn xếp (stack)	41
Chương 6. Hàng đợi (Queue)	50
6.1. Nội dung	Error! Bookmark not defined.
6.2. Khái niệm hàng đợi	50
6.3. Khởi tạo	50
6.4. Ứng dụng của Queue.....	51
6.5. Thao tác trên Queue	52
6.6. Bài toán thường gặp	58
6.7. Bài tập	59
Chương 7. Cây (Tree)	61
7.1. Cấu trúc cây.....	61
7.2. Cây nhị phân.....	63
7.3. Cây nhị phân tìm kiếm.....	69
7.4. B-Tree.....	77
7.5. Bài tập	87
Chương 8. Đồ thị (Graph)	89
8.1. Sơ lược về lịch sử hình thành của đồ thị	89
8.2. Định nghĩa và các khái niệm liên quan	89
8.3. Biểu diễn đồ thị trên máy tính.....	100
8.4. Các thao tác duyệt đồ thị DFS, BFS	108
8.5. Ứng dụng đồ thị (Bài toán tô màu,...)	118
Chương 9. Bảng băm (Hash table)	120
9.1. Giới thiệu cơ bản về bảng băm.....	120
9.2. Hàm băm.....	121
9.3. Giải quyết – xử lý va chạm (đụng độ).....	122
Phần 3. ĐỀ THI MẪU	139



Phần 1. GIẢI THUẬT

Chương 1. Giới thiệu về giải thuật

1.1. Khái quát về giải thuật.

1.1.1. Định nghĩa

Giải thuật là một quy trình rõ ràng, được xác định và hiểu được để giải quyết một vấn đề tính toán. Để được coi là một giải thuật, một quy trình phải có ba tính chất sau:

- Đi đến một kết quả chính xác.
- Kết thúc trong một số bước hữu hạn.
- Áp dụng cho một lớp vấn đề đầu vào tiêu chuẩn.

Tóm lại : Giải thuật là một phương pháp hoặc quy trình giải quyết một vấn đề tính toán bằng cách mô tả một tập hữu hạn các bước đơn giản, mà sau khi thực hiện sẽ giải quyết vấn đề đó với đầu vào bất kỳ.

1.1.2. Sự cần thiết của việc học giải thuật

- Giải thuật là cơ sở của khoa học máy tính
- Tối ưu hóa hiệu suất
- Giải quyết vấn đề phức tạp
- Cải thiện khả năng logic và tư duy
- Sự cần thiết trong các lĩnh vực khác nhau
- Hiểu sâu về các cấu trúc dữ liệu
- Phát triển kỹ năng gỡ lỗi và phân tích

1.2. Phân tích thuật toán

- Dự đoán hiệu suất.
- So sánh thuật toán.
- Cung cấp bảo đảm.
- Hiểu cơ sở lý thuyết.

(?) Một số câu hỏi được đặt ra

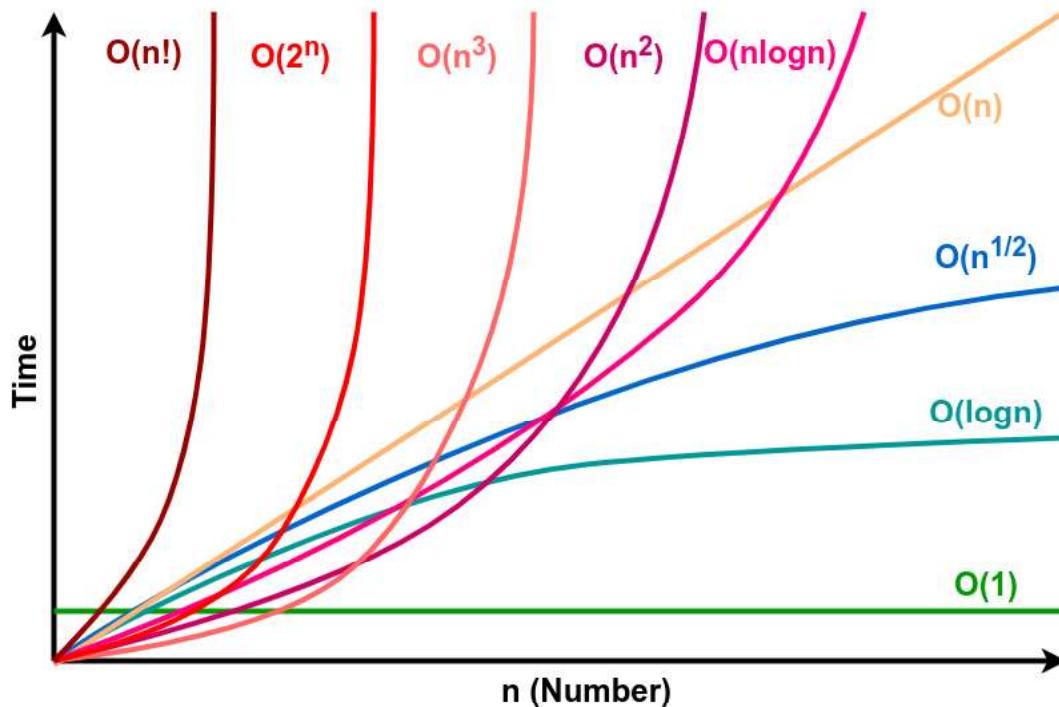
- Liệu chương trình của tôi có thể giải quyết được một đầu vào thực tế lớn không?
- Tại sao chương trình của tôi lại chạy chậm?
- Tại sao chương trình của tôi lại hết bộ nhớ?

1.3. Độ phức tạp của thuật toán.

1.3.1. Một số khái niệm căn bản:

- $T(N)$ là một hàm số để mô tả thời gian thực thi của một thuật toán, dựa trên kích thước đầu vào của nó.
- Big O là ký hiệu đại số để biểu diễn độ phức tạp thời gian của một thuật toán, biểu thị tốc độ tăng của hàm thời gian $T(N)$ khi kích thước đầu vào N tiến đến vô hạn.

- Ký hiệu của Big O là "O", sau đó là một biểu thức đại số biểu thị hàm tốn thời gian của thuật toán, ví dụ như $O(N)$ hoặc $O(N \log N)$.



1.3.2. Một số độ phức tạp của các giải thuật phổ biến:

Tìm kiếm tuyến tính (Linear Search):

- Độ phức tạp trung bình và tốt nhất là $O(n)$
- Độ phức tạp xấu nhất cũng là $O(n)$

1.3.2.1. Tìm kiếm nhị phân (Binary Search):

- Độ phức tạp trung bình và tốt nhất là $O(\log n)$
- Độ phức tạp xấu nhất cũng là $O(\log n)$

1.3.2.2. Tìm kiếm nội suy (Interpolation Search):

- Độ phức tạp trung bình là $O(\log(\log n))$
- Tuy nhiên, trong trường hợp phân bố đều các phần tử thì độ phức tạp có thể lên đến $O(n)$.

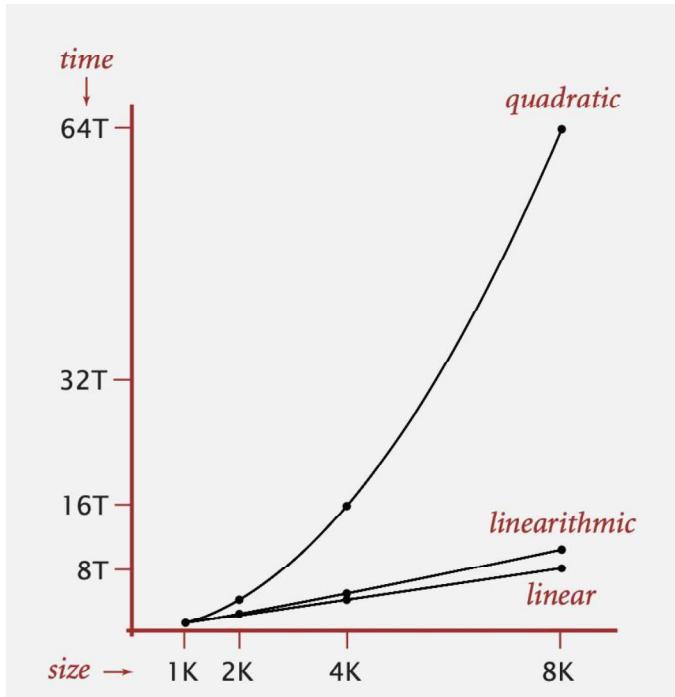
1.3.2.3. Thuật toán tìm kiếm tuyến tính cải tiến

- Sử dụng phương pháp "skip" để tăng tốc độ tìm kiếm.
- Trong trường hợp tốt nhất, độ phức tạp là $O(1)$
- Trong trường hợp xấu nhất là $O(n)$,
- Trung bình là $O(n/m)$, trong đó m là số phần tử mà thuật toán "skip" qua.

1.3.2.4. Độ phức tạp của một số giải thuật sắp xếp

- Sắp xếp nổi bọt (Bubble Sort): $O(n^2)$

- ii) Sắp xếp chọn (Selection Sort): $O(n^2)$
- iii) Sắp xếp chèn (Insertion Sort): $O(n^2)$
- iv) Sắp xếp nhanh (Quick Sort): $O(n \log n)$
- v) Sắp xếp trộn (Merge Sort): $O(n \log n)$
- vi) Sắp xếp vun đống (Heap Sort): $O(n \log n)$



N^2	quadratic
$N \log N$	linearithmic
N	linear

1.3.3. Phát triển Câu 1 a Đề minh họa cuối Kỳ DSA

Câu 1:

- a) Hãy cho biết độ phức tạp của thuật toán **Heap sort** theo định nghĩa Big-O (0.25đ)
- b) Hãy cho biết độ phức tạp của thuật toán **Quick sort** theo định nghĩa Big-O (0.25đ)
- c) Hãy cho biết độ phức tạp của thuật toán **Insertion sort** theo định nghĩa Big-O (0.25đ)
- d) Hãy cho biết độ phức tạp của thuật toán **Selection sort** theo định nghĩa Big-O (0.25đ)
- e) Hãy cho biết độ phức tạp của thuật toán **Merge sort** theo định nghĩa Big-O (0.25đ)
- f) Hãy cho biết độ phức tạp của thuật toán **Tìm kiếm tuyến tính (Linear Search)** theo định nghĩa Big-O (0.25đ)
- g) Hãy cho biết độ phức tạp của thuật toán **Tìm kiếm nhị phân (Binary Search)** theo định nghĩa Big-O (0.25đ)



Chương 2. Thuật toán tìm kiếm

2.1. Định nghĩa về tìm kiếm

Trong môn khoa học máy tính, tìm kiếm là một quá trình tìm kiếm một đối tượng với những thuộc tính đặc trưng trong tập hợp các đối tượng. Đối tượng tìm kiếm ấy có thể là bản ghi trong một cơ sở dữ liệu, những dữ liệu cơ bản trong một mảng, đoạn văn trong files, node trên cây,...

2.2. Tại sao chúng ta cần tìm kiếm?

Tìm kiếm là một trong những phần quan trọng nhất của thuật toán. Chúng ta đều biết rằng máy tính lưu trữ rất nhiều thông tin. Và để tìm kiếm thông tin ấy một cách nhanh chóng, chúng ta cần phải nhờ thuật toán tìm kiếm. Có rất nhiều cách để sắp xếp thông tin giúp cho việc tìm kiếm trở nên nhanh chóng. Trong chương này, chúng ta sẽ tìm hiểu một số thuật toán tìm kiếm cơ bản.

2.3. Các thuật toán tìm kiếm cơ bản

2.3.1. Thuật toán tìm kiếm tuyến tính

2.3.1.1. Bài toán tìm kiếm

Cho trước mảng một chiều có **n** phần tử. Tìm phần tử **data** có giá trị cho trước trong mảng. Nếu phần tử đó **tồn tại** trong danh sách thì **xuất vị trí của nó trong danh sách**. Nếu phần tử đó **không tồn tại** trong danh sách thì trả về giá trị **-1**.

2.3.1.2. Ý tưởng thuật toán

Như chúng ta đã biết, tìm kiếm tuyến tính còn được gọi là tìm kiếm tuần tự, vì thế, chúng ta sẽ so sánh tuần tự giá trị của x với phần tử thứ 0, phần tử thứ 1,... của danh sách cho đến khi tìm được phần tử cần tìm. Nếu không tìm thấy thì trả về giá trị -1.

2.3.1.3. Các bước của thuật toán

Bước 1: Gán $i = 0$ // Giá trị khởi đầu

Bước 2: So sánh $a[i]$ với $data$. Có 2 khả năng xảy ra

- Nếu $a[i] = data$ thì trả về giá trị i là vị trí cần tìm
- Nếu $a[i] \neq data$ thì tiếp tục bước 3

Bước 3: $i = i + 1$ // Xét tiếp phần tử tiếp theo trong mảng

- Nếu $i == n$ thì không tìm thấy à Trả về giá trị -1.
- Ngược lại: Tiếp tục bước 2

2.3.1.4. Code minh họa:

```
int linearSearch(int a[], int n, int data) {  
    for (int i = 0; i < n; i++) {  
        if (a[i] == data) {  
            return i;  
        }  
    }  
    return -1;  
}
```

```

        }
    }
    return -1;
}

```

- Độ phức tạp thuật toán: O(n)
- Trường hợp xấu nhất: Duyệt hết tất cả phần tử trong mảng
- Space complexity: O(1)
-

2.3.2. Thuật toán tìm kiếm nhị phân

2.3.2.1. Bài toán tìm kiếm

Cho trước một mảng một chiều có **n** phần tử đã được sắp xếp. Tìm phần tử **data** có giá trị cho trước trong mảng. Nếu phần tử đó **tồn tại** trong danh sách thì **xuất vị trí của nó trong danh sách**. Nếu phần tử đó **không tồn tại** trong danh sách thì trả về giá trị **-1**.

2.3.2.2. Ý tưởng thuật toán:

Chúng ta sẽ so sánh data với giá trị phần tử chính giữa của mảng. Nếu như giá trị đó bằng data thì xuất ra vị trí ấy, nếu nhỏ hơn data thì quá trình tìm kiếm sẽ tiếp tục ở nửa trước của mảng, nếu lớn hơn data thì quá trình tìm kiếm sẽ tiếp tục ở nửa sau của mảng. Bằng cách này, ở mỗi phép lặp của thuật toán, chúng ta có thể loại bỏ được nửa mảng mà giá trị data chắc chắn sẽ không xuất hiện.



2.3.2.3. Các bước của thuật toán

Bước 1: Gán start = 0, end = n – 1

Bước 2: So sánh start với end

- Nếu start > end thì tìm kiếm thất bại à Trả về giá trị -1
- Ngược lại
 - Gán mid = start + (end – start) / 2 // Tránh bị tràn mảng
 - So sánh a[mid] với data
- Nếu a[mid] = data thì trả về giá trị mid là vị trí cần tìm à Kết thúc bài toán
- Nếu a[mid] < data thì gán start = mid + 1
- Nếu a[mid] > data thì gán end = mid – 1
 - Tiếp tục thực hiện bước 2

2.3.2.4. Code minh họa

- Code trực tiếp:



```
int binarySearch(int a[], int n, int data) {  
    int start = 0;  
    int end = n - 1;  
    while (start < end) {  
        int mid = start + (end - start) / 2;  
        if (a[mid] == data) {  
            return mid;  
        }  
        else if (a[mid] < data) {  
            start = mid + 1;  
        }  
        else {  
            end = mid - 1;  
        }  
    }  
    return -1;  
}
```

- Code bằng đệ quy (recursion):

```
int binarySearchRecursion(int a[], int start, int end, int data) {  
    int mid = start + (end - start) / 2;  
    if (start > end) {  
        return -1;  
    }  
    else if (a[mid] == data) {  
        return mid;  
    }  
    else if (a[mid] < data) {  
        binarySearchRecursion(a, mid + 1, end, data);  
    }  
    else {  
        binarySearchRecursion(a, start, end - 1, data);  
    }  
    return -1;  
}
```

- Độ phức tạp : O(logn)
- Trường hợp xấu nhất: Không tìm thấy phần tử data trong mảng
- Space complexity: O(1) (code trực tiếp), O(logn) (code bằng đệ quy)

2.4. Ví dụ

- **Ví dụ 1:** Cho mảng a[7] = {10, 20, 30, 40, 50, 60, 70}. Hỏi phần tử data bằng 55 có tồn tại trong mảng không?



Cách 1: Dùng tìm kiếm tuyến tính, duyệt từng phần tử trong mảng

```
#include<iostream>
using namespace std;
int main(){
    int a[] = { 10, 20, 30, 40, 50, 60, 70 };
    int find = 55;
    int pos = -1;
    int n = (sizeof(a) / sizeof(a[0])) - 1;
    for (int i = 0; i < n; i++) {
        if (a[i] == find) {
            pos = i;
            break;
        }
    }
    if (a[pos] == find) {
        cout << pos << endl;
    }
    else {
        cout << "Not Find" << endl;
    }
    return 0;
}
```

- Độ phức tạp: $O(n)$

Cách 2: Dùng tìm kiếm nhị phân

```
#include<iostream>
using namespace std;
int main() {
    int a[] = { 10, 20, 30, 40, 50, 60, 70 };
    int find = 55;
    int start = 0;
    int end = (sizeof(a) / sizeof(a[0])) - 1;
    int mid;
    while (start <= end) {
        mid = (start + end) / 2;
        if (a[mid] == find) {
            break;
        }
        else if (a[mid] < find) {
            start = mid + 1;
        }
    }
}
```

```

        else {
            end = mid - 1;
        }
    if (a[mid] == find) {
        cout << mid << endl;
    }
    else {
        cout << "Not Find" << endl;
    }
    return 0;
}

```

- Độ phức tạp: $O(n \log n)$

Cách 3: Cải tiến thuật toán nhị phân (Tham khảo)

- Ở thuật toán tìm kiếm nhị phân, chúng ta gán

$mid = start + (end - start)/2.$

- Nhưng chúng ta có cải tiến nó bằng cách gán

$mid = ((find - a[start]) * (end - start)) / (a[end] - a[start]).$

(Với $find$ là số cần tìm)

- Việc gán như thế làm giảm phạm vi tìm kiếm và làm cho độ phức tạp thuật toán giảm xuống

```
#include<iostream>
using namespace std;
int main() {
    int a[] = { 10, 20, 30, 40, 50, 60, 70 };
    int find = 55;
    int start = 0;
    int end = (sizeof(a) / sizeof(a[0])) - 1;
    int mid;
    while (start <= end) {
        if (find < a[start]) {
            break;
        }
        else {
            mid = ((find - a[start]) * (end - start)) / (a[end] - a[start]);
        }
        if (a[mid] == find) {
            break;
        }
    }
}
```

```
        }
        else if (a[mid] < find) {
            start = mid + 1;
        }
        else {
            end = mid - 1;
        }
    }
    if (a[mid] == find) {
        cout << mid << endl;
    }
    else {
        cout << "Not Find" << endl;
    }
    return 0;
}
```

Độ phức tạp: $O(\log(\log n))$

- **Ví dụ 2:** Cho một mảng số nguyên gồm n phần tử. Viết hàm kiểm tra xem mảng đó có tồn tại cặp phần tử giống nhau hay không ? Nếu có trả về 1, ngược lại trả về 0.

Cách 1: Dùng 2 vòng lặp for để kiểm tra

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

bool kiemTraPhanTuTrungLap(int a[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (a[i] == a[j]) {
                return 1;
            }
        }
    }
    return 0;
}

int main() {
    int n;
    cin >> n;
    int* a;
    a = new int[n];
```

```
for (int i = 0; i < n; i++) {
    cin >> a[i];
}
if (kiemTraPhanTuTrungLap(a, n) == 1) {
    cout << '1' << endl;
}
else {
    cout << '0' << endl;
}
return 0;
}
```

- Độ phức tạp: $O(n^2)$

Cách 2: Dùng thư viện stl sort để sắp xếp mảng tăng dần, sau đó dùng tìm kiếm tuyến tính để kiểm tra

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

bool kiemTraPhanTuTrungLap(int a[], int n) {
    for (int i = 0; i < n - 1; i++) {
        if (a[i] == a[i + 1])
            return 1;
    }
    return 0;
}
int main()
{
    int n;
    cin >> n;
    int* a;
    a = new int[n];
    for (int i = 0; i < n; i++) {
        cin >> a[i];
    }
    sort(a, a + n);
    if (kiemTraPhanTuTrungLap(a, n) == 1) {
        cout << '1' << endl;
    }
    else {
        cout << '0' << endl;
    }
}
```

```

    }
    return 0;
}

```

Độ phức tạp: $O(n \log n)$

- **Ví dụ 3:** Cho một mảng số nguyên gồm n phần tử. Gọi x là tổng của 2 phần tử bất kỳ tổng mảng. Viết chương trình tìm x sao cho x có giá trị lớn nhất và x bé hơn k .

Cách 1: Dùng 2 vòng for để kiểm tra

```

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

int main() {
    int n, x;
    cin >> n >> x;
    int* a;
    a = new int[n];
    for (int i = 0; i < n; i++) {
        cin >> a[i];
    }
    int money = 0;
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            int y = a[i] + a[j];
            if (y <= x) {
                money = max(money, y);
            }
        }
    }
    cout << money << endl;
}

```

- Độ phức tạp: $O(n^2)$

Cách 2: Dùng STL Sort để sắp xếp mảng tăng dần, sau đó dùng tìm kiếm tuyến tính để kiểm tra

```

#include<iostream>
#include<algorithm>

```



```
using namespace std;

int main() {
    int n, x;
    cin >> n >> x;
    int* a;
    a = new int[n];
    for (int i = 0; i < n; i++) {
        cin >> a[i];
    }
    sort(a, a + n);
    int money = 0;
    int start = 0;
    int end = n - 1;
    while (start < end) {
        int y = a[start] + a[end];
        if (y <= x) {
            money = max(money, y);
            start++;
        }
        else {
            end--;
        }
    }
    cout << money << endl;
}
```

- Độ phức tạp: $O(n \log n)$



Chương 3. Các thuật toán sắp xếp

3.1. Sắp xếp chọn (Selection sort)

3.1.1. Sơ lược về nguồn gốc

Selection sort là một thuật toán sắp xếp đơn giản và cổ điển, được phát triển bởi nhà toán học Robert W. Floyd vào năm 1964. Tuy nhiên, trước đó, nó đã được phát triển bởi nhà toán học người Anh John von Neumann vào những năm 1945.

Thuật toán selection sort là một trong những thuật toán đơn giản nhất để sắp xếp một danh sách các phần tử. Nó hoạt động bằng cách lần lượt tìm kiếm phần tử nhỏ nhất trong danh sách và đưa nó về vị trí đầu tiên. Sau đó, thuật toán tiếp tục tìm kiếm phần tử nhỏ nhất trong phần còn lại của danh sách và đưa nó về vị trí thứ hai. Thuật toán tiếp tục thực hiện như vậy cho tới khi toàn bộ danh sách được sắp xếp.

Mặc dù selection sort có hiệu quả không cao như các thuật toán sắp xếp khác, nhưng nó vẫn được sử dụng trong một số trường hợp đơn giản hoặc khi các dữ liệu đầu vào có kích thước nhỏ.

3.1.2. Ý tưởng

- Tìm phần tử nhỏ nhất trong danh sách.
- Đổi chỗ phần tử nhỏ nhất với phần tử ở vị trí đầu tiên trong danh sách.
- Tìm phần tử nhỏ nhất trong phần còn lại của danh sách (trừ phần tử ở vị trí đầu tiên).
- Đổi chỗ phần tử nhỏ nhất với phần tử ở vị trí thứ hai trong danh sách.
- Tiếp tục quá trình tìm kiếm và đổi chỗ phần tử nhỏ nhất cho đến khi danh sách được sắp xếp.

3.1.3. Pseudocode:

- **Bước 1:** $i=1$.
- **Bước 2:** Tìm phần tử $a[min]$ nhỏ nhất trong dãy hiện hành từ $a[i]$ đến $a[n]$.
- **Bước 3:** Hoán vị $a[min]$ và $a[i]$
- **Bước 4:** Nếu $i <= n-1$ thì $i=i+1$; Lặp lại bước 2.

Ngược lại: Dừng. $n-1$ phần tử đã nằm đúng vị trí.

3.1.4. Cài đặt hàm

```
void selectionSort(int arr[], int n) {  
    int i, j, min_idx;  
    for (i = 0; i < n - 1; i++) {  
        min_idx = i;  
        for (j = i + 1; i < n; j++) {  
            if (arr[j] < arr[min_idx]) {  
                min_idx = j;  
            }  
        }  
        swap(arr[min_idx], arr[i]);  
    }  
}
```

```
    }
}
```

Trong đó:

- **arr:** là mảng chứa các phần tử cần sắp xếp.
- **n:** là số lượng phần tử trong mảng.
- **i:** là biến lặp bên ngoài, duyệt từ đầu đến cuối mảng để tìm vị trí phần tử đang xét.
- **j:** là biến lặp bên trong, duyệt từ vị trí $i+1$ đến cuối mảng để tìm phần tử nhỏ nhất trong phần còn lại của mảng.
- **min_idx:** là biến lưu vị trí của phần tử nhỏ nhất trong phần còn lại của mảng.
- **Hàm swap():** là hàm trao đổi giá trị của hai biến.

3.1.5. Biểu diễn chi tiết thuật toán

//Hình vẽ minh họa thống nhất sau

Sơ lược bố cục:

- Một cái mảng
- Chạy từng bước của thuật toán (biểu diễn sự thay đổi của mảng)

3.1.6. Ví dụ

Sắp xếp mảng một chiều a gồm có các phần tử: 3, 1, 2, 7, 5.

- Khởi tạo mảng:

3	1	2	7	5
---	---	---	---	---

- $i=0, a[i]=3$

3	1	2	7	5
---	---	---	---	---

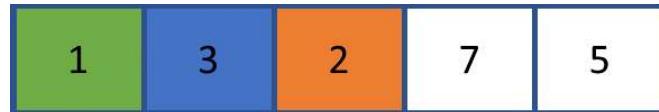
- $\text{min_idx}=1, a[\text{min_idx}]=1$
 $\Rightarrow \text{swap}(a[i], a[\text{min_idx}])$ ta được phần tử $i = 0$ đã được sắp xếp

3	1	2	7	5
1	3	2	7	5

- Bước tiếp theo, $i=1, a[i]=3$

1	3	2	7	5
---	---	---	---	---

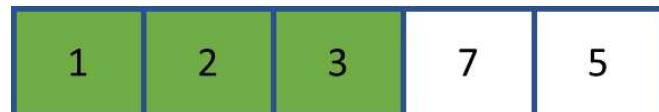
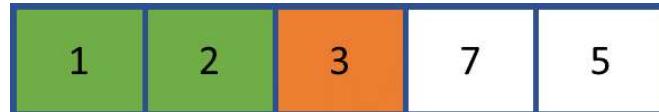
- $\text{min_idx}=2$, $a[\text{min_idx}]=2 \Rightarrow \text{swap}(a[i], a[\text{min_idx}])$ ta được phần tử $i = 1$ đã được sắp xếp



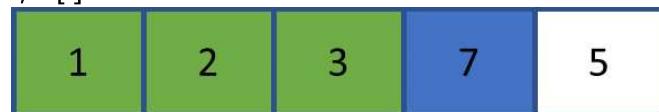
- Bước tiếp theo, $i=2$, $a[i]=3$



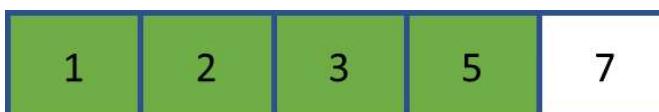
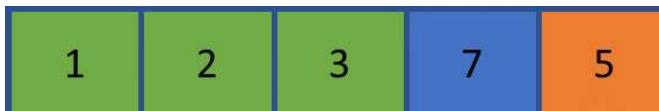
- $\text{min_idx}=2$, $a[\text{min_idx}]=3 \Rightarrow \text{swap}(a[i], a[\text{min_idx}])$ ta được phần tử $i = 2$ đã được sắp xếp



- Bước tiếp theo, $i=3$, $a[i]=7$



- $\text{min_idx}=4$, $a[\text{min_idx}]=5 \Rightarrow \text{swap}(a[i], a[\text{min_idx}])$ ta được phần tử $i = 3$ đã được sắp xếp



- Kết thúc chương trình, ta được mảng đã sắp xếp.



3.1.7. Bài tập luyện tập

BT 3.1.7.1 Sắp xếp mảng số nguyên tăng dần, giảm dần

BT 3.1.7.2 Sắp xếp mảng số thực tăng dần, giảm dần



BT 3.1.7.3 Sắp xếp mảng phân số tăng dần, giảm dần

BT 3.1.7.4 Sắp xếp mảng chuỗi tăng dần, giảm dần và đếm số lần truy cập vào các phần tử

3.2. Sắp xếp chèn (Insertion sort)

3.2.1. Giới Thiệu

- Thuật toán Insertion Sort, một thuật toán hiệu quả để sắp xếp một số lượng nhỏ các phần tử.
- Ta đã vô tình áp dụng thuật toán Insertion Sort trong trò chơi bài tây mỗi khi chúng ta vừa bốc bài và vừa xếp nó trên tay.

3.2.2. Mô tả cách hoạt động của thuật toán Insertion sort bằng việc xếp các lá bài.

Bước 1: Bắt đầu với một tay trống và các lá bài úp mặt xuống trên bàn.

Bước 2: Lấy từng lá bài từ bàn và xem nó như một phần tử cần được sắp xếp.

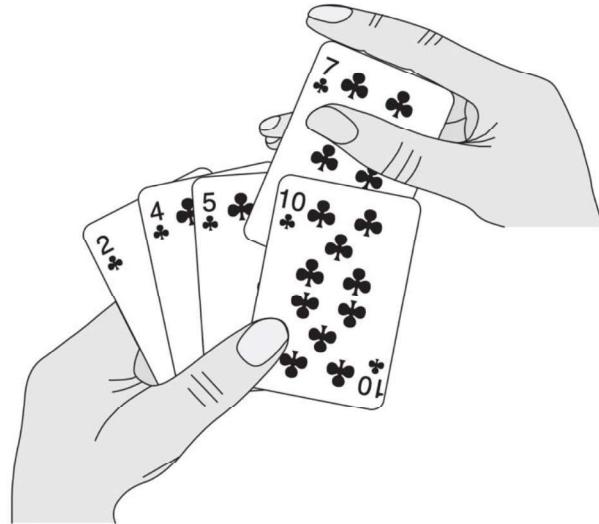
Bước 3 : Chèn nó vào vị trí đúng trong tay trái

Để tìm vị trí đúng cho một lá bài, chúng ta so sánh nó với mỗi lá bài đã có trong tay trái, từ phải sang trái.

Luôn luôn, các lá bài nằm trong tay trái đã được sắp xếp.

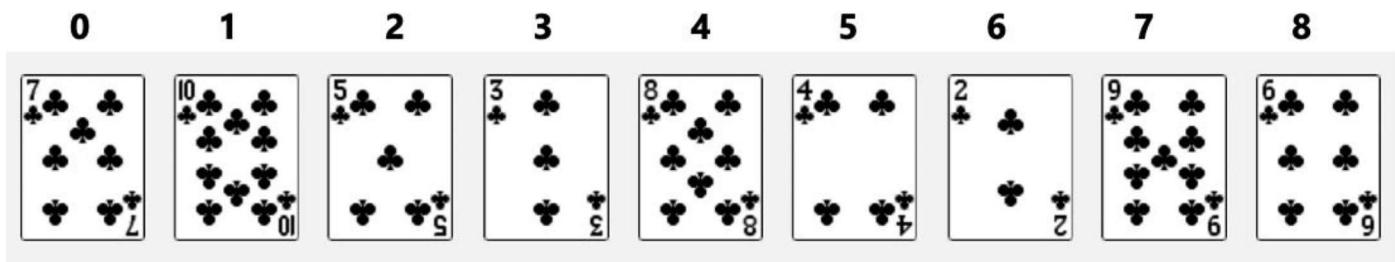
Bước 4 : Lặp lại các bước 2 đến 3 cho đến khi tất cả các lá bài được chèn vào tay.

Tương tự, thuật toán Insertion Sort sẽ sắp xếp các phần tử trong một mảng theo thứ tự tăng dần bằng cách chèn từng phần tử vào vị trí đúng của nó trong mảng đã được sắp xếp trước đó.



3.2.3. Ví dụ minh họa

$A = \{7, 10, 5, 3, 8, 4, 2, 9, 6\}$ và $n=9$



Bước 1: Chèn A[1]=10 vào đoạn [0,0] sao cho ta được đoạn [0,1] có thứ tự

0	1	2	3	4	5	6	7	8
7	10	5	3	8	4	2	9	6

j i



Bước 2: Chèn A[2]=5 vào đoạn [0,1] sao cho ta được đoạn [0,2] có thứ tự

0	1	2	3	4	5	6	7	8
5	7	10	3	8	4	2	9	6



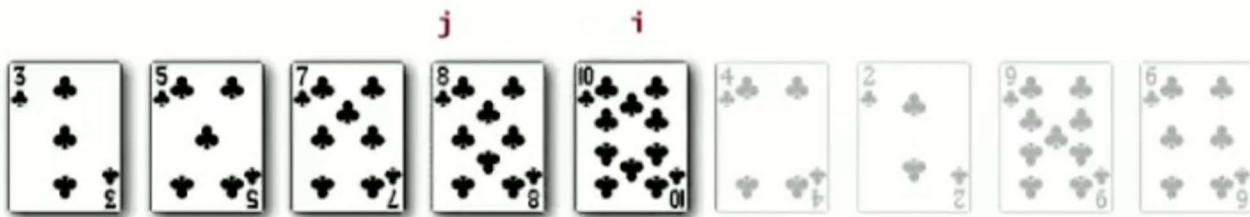
Bước 3: Chèn A[3]=3 vào đoạn [0,2] sao cho ta được đoạn [0,3] có thứ tự

0	1	2	3	4	5	6	7	8
3	5	7	10	8	4	2	9	6



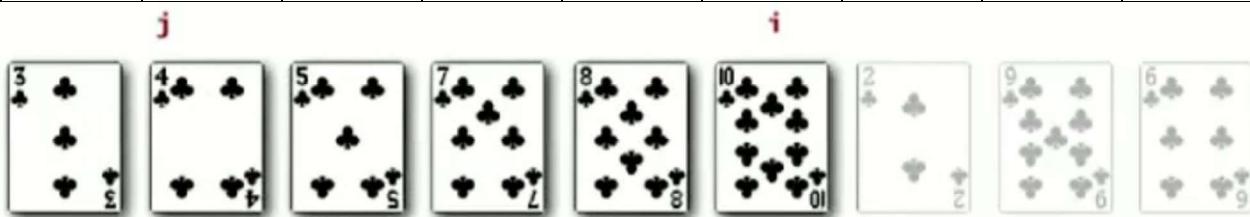
Bước 4: Chèn A[4]=8 vào đoạn [0,3] sao cho ta được đoạn [0,4] có thứ tự

0	1	2	3	4	5	6	7	8
3	5	7	8	10	4	2	9	6



Bước 5: Chèn A[5]=4 vào đoạn [0,4] sao cho ta được đoạn [0,5] có thứ tự

0	1	2	3	4	5	6	7	8
3	4	5	7	8	10	2	9	6



Bước 6: Chèn A[6]=2 vào đoạn [0,5] sao cho ta được đoạn [0,6] có thứ tự

0	1	2	3	4	5	6	7	8
2	3	4	5	7	8	10	9	6



Bước 7: Chèn A[7]=9 vào đoạn [0,6] sao cho ta được đoạn [0,7] có thứ tự

0	1	2	3	4	5	6	7	8
2	3	4	5	7	8	9	10	6



Bước 8: Chèn $A[8]=6$ vào đoạn $[0,7]$ sao cho ta được đoạn $[0,8]$ có thứ tự

0	1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	9	10



sorted

- Kết thúc giải thuật, dãy số kết quả là : 2 3 4 5 6 7 8 9 10.

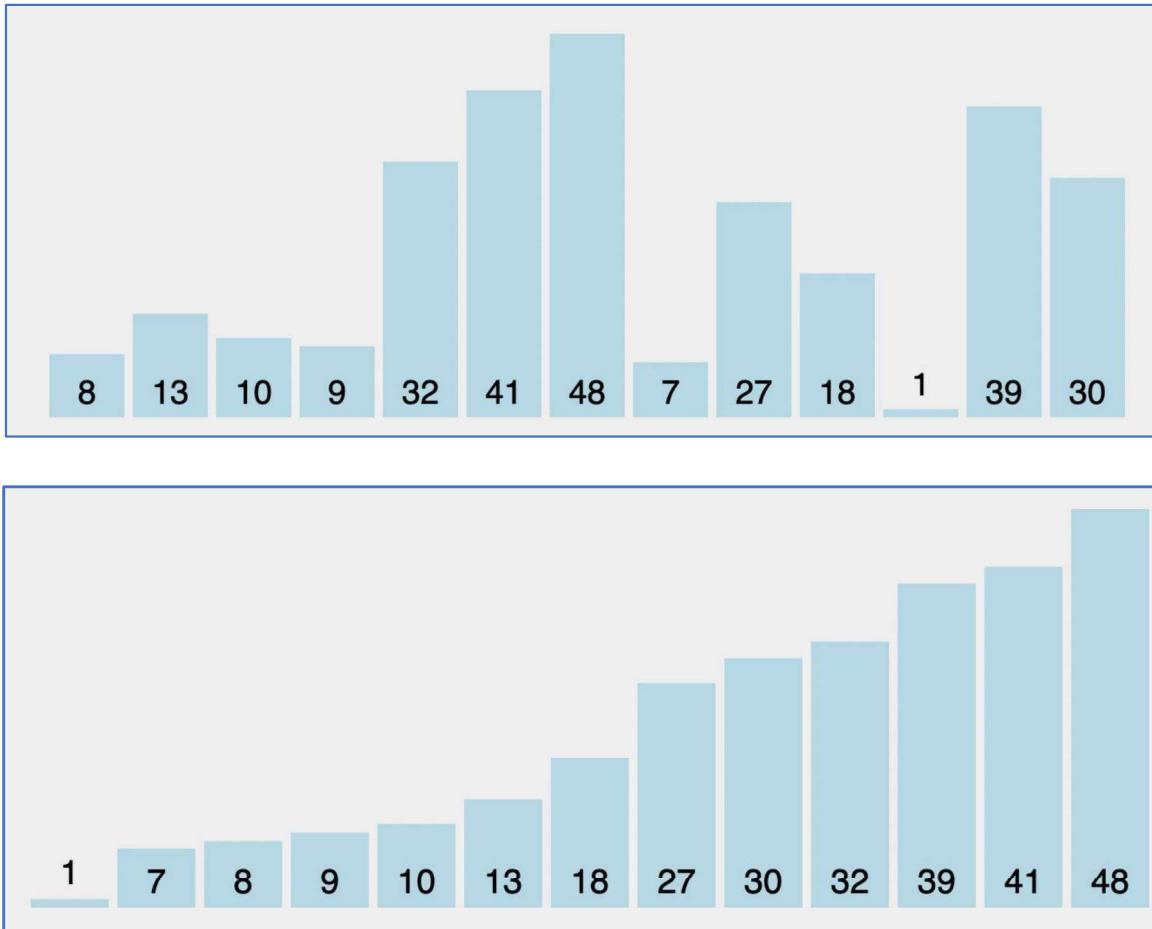
3.2.4. Đoạn code c++ cho thuật toán Insertion sort

```
void insertionSort(int arr[], int n) {

    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

3.2.5. Insertion sort: animation

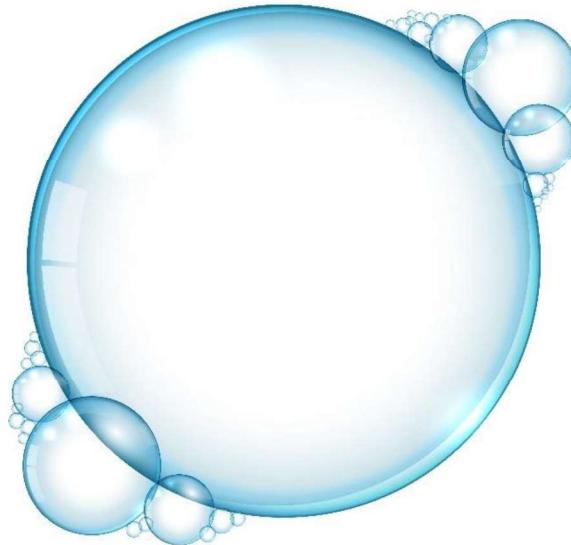
13 Random items



- Web tham khảo chạy từng bước thuật toán: <https://visualgo.net/en/sorting>

3.3. Sắp xếp nổi bọt (Bubble Sort)

- Bubble Sort, hay còn gọi là sắp xếp nổi bọt, là một thuật toán sắp xếp đơn giản và cổ điển.
- Được phát triển vào những năm 1950, thuật toán này đã trở thành một phần quan trọng trong việc giảng dạy các thuật toán sắp xếp cơ bản trong lĩnh vực khoa học máy tính.

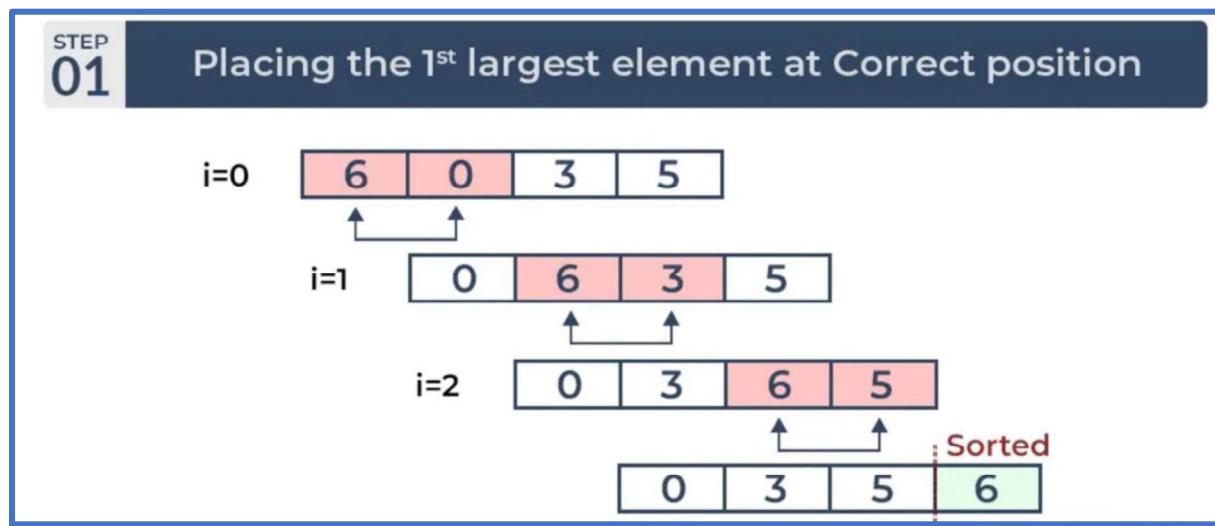


3.3.1. Mô tả cách hoạt động của thuật toán Bubble Sort

- **Input:** arr[] = {6, 3, 0, 5}

Bước 1: Vòng lặp thứ nhất:

Phần tử lớn nhất được đặt ở vị trí chính xác của nó, tức là ở cuối mảng.

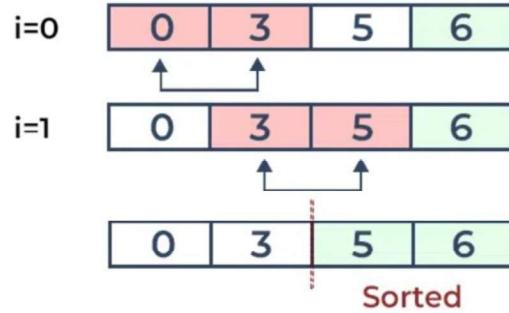


Bước 2 : Lần lặp thứ hai:

Đặt phần tử lớn thứ hai ở vị trí đúng

STEP
02

Placing 2nd largest element at Correct position

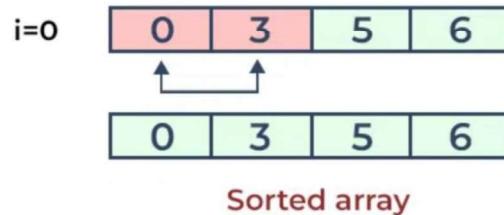


Bước 3 : Lần lặp thứ ba:

Đặt hai phần tử còn lại ở vị trí đúng của chúng.

STEP
03

Placing 3rd largest element at Correct position





3.3.2. Đoạn code c++ cho thuật toán Bubble sort

```
void bubbleSort(int arr[], int n) {  
    for (int i = 0; i < n - 1; i++) {  
        for (int j = 0; j < n - i - 1; j++) {  
            if (arr[j] > arr[j + 1]) {  
                swap(arr[j], arr[j + 1]);  
            }  
        }  
    }  
}
```



3.4. Sắp xếp nhanh (Quick sort)

3.4.1. Sơ lược về nguồn gốc

- Quick sort là một thuật toán sắp xếp được phát minh bởi nhà khoa học máy tính người Anh Tony Hoare vào năm 1959 khi ông đang làm việc tại câu lạc bộ Turing ở Luân Đôn.
- Thuật toán này được coi là một trong những thuật toán sắp xếp nhanh nhất và được sử dụng rộng rãi trong thực tế.

3.4.2. Ý tưởng

- Ý tưởng chính của thuật toán Quick sort là dựa trên thuật toán Chia để trị chọn một phần tử làm trục (pivot) và phân vùng mảng đã cho xung quanh trục đã chọn bằng cách đặt trục vào đúng vị trí của nó trong mảng được sắp xếp.
- Thuật toán Quick sort hoạt động bằng cách chọn một phần tử trong dãy đầu vào làm "pivot". Sau đó, các phần tử trong dãy đầu vào được phân thành hai phần: các phần tử nhỏ hơn hoặc bằng pivot được đưa về phía trước của pivot và các phần tử lớn hơn pivot được đưa về phía sau của pivot. Sau khi phân hoạch này được thực hiện, pivot sẽ được đặt vào giữa hai phần được phân chia.
- Sau đó, thuật toán sẽ đệ quy áp dụng phương pháp trên cho các phần tử nằm bên trái và phải của pivot cho đến khi dãy được sắp xếp hoàn toàn. Quá trình đệ quy sẽ kết thúc khi chỉ còn một phần tử hoặc không còn phần tử nào để phân chia.
- (!) Với mỗi đệ quy, thuật toán Quick sort sẽ có thể chọn một pivot khác nhau. Có nhiều cách để chọn pivot, một cách đơn giản là chọn phần tử ở giữa dãy đầu vào. Tuy nhiên, sử dụng cách chọn pivot khác nhau sẽ ảnh hưởng đến hiệu suất của thuật toán Quick sort.

3.4.3. Cài đặt hàm

- Chúng ta cần xác định được các hàm để thực hiện thuật toán Quick sort:
 - Hàm Quick Sort với 2 đối số.
 - Hàm Quick Sort với 3 đối số.
 - Hàm Partition.
- Định nghĩa hàm:
 - Quick sort với 2 đối số:

```
void quickSort(int a[], int n) {  
    quickSort(a, 0, n - 1);  
}
```

- Quick sort với 3 đối số:

```
void quickSort(int a[], int left, int right) {  
    if (left < right) {  
        int id_pivot = Partition(a, left, right);  
        quickSort(a, left, id_pivot - 1);  
    }  
}
```

```

        quickSort(a, id_pivot + 1, right);
    }
}

```

- Partition:

Hàm partition có chức năng chia mảng thành hai phần, một phần có giá trị nhỏ hơn hoặc bằng pivot (giá trị được chọn để so sánh và phân chia mảng) và một phần có giá trị lớn hơn pivot.

```

int partition(int a[], int left, int right) {
    int pivot = a[right];
    int id = left - 1;
    for (int i = left; i < right; i++) {
        if (a[i] < pivot) {
            id++;
            swap(a[id], a[i]);
        }
    }
    id++;
    swap(a[id], a[right]);
    return id;
}

```

3.4.4. Biểu diễn chi tiết thuật toán

Cho mảng số nguyên gồm các phần tử {10, 5, 30, 70, 40, 80, 90} trình bày từng bước giải thuật Quick Sort để sắp xếp mảng giảm:

- Mảng ban đầu:

10	5	30	70	40	80	90
----	---	----	----	----	----	----

- Phân hoạch lần 1: (chọn 70 làm phần tử phân hoạch)

90	80	70	30	40	5	10
----	----	----	----	----	---	----

- Phân hoạch lần 2: (chọn 80 làm phần tử phân hoạch [đoạn 90,80])

90	80	70	30	40	5	10
----	----	----	----	----	---	----

- Phân hoạch lần 3: (chọn 5 làm phần tử phân hoạch [đoạn 30,40,5,10])

90	80	70	30	40	10	5
----	----	----	----	----	----	---

- Lần 4 : Chọn phần tử 40 làm phần tử phân hoạch [đoạn 30,40,10]

90	80	70	40	30	10	5
----	----	----	----	----	----	---

- Kết thúc phân hoạch ta được mảng đã sắp giảm {90, 80, 70, 40, 30, 10, 5}



3.4.5. Đánh giá thuật toán

3.4.5.1. Độ phức tạp

- Trung bình: $O(n\log n)$
- Tốt nhất: $O(n\log n)$ (khi mảng được chia đều)
- Xấu nhất: $O(n^2)$ (khi mảng đã sắp xếp hoặc đảo ngược)

3.4.5.2. Ưu điểm

- Quick sort là một trong những thuật toán sắp xếp nhanh nhất.
- Quick sort dễ dàng được cài đặt và tối ưu hóa.
- Quick sort hoạt động hiệu quả trên bộ nhớ do không cần phải sử dụng thêm bộ nhớ phụ.

3.4.5.3. Nhược điểm

- Quick sort có thể trở nên chậm nếu mảng đã được sắp xếp hoặc đảo ngược.
- Trong trường hợp xấu nhất, khi mảng đã được sắp xếp hoặc đảo ngược, quick sort có thể làm việc với độ phức tạp $O(n^2)$.
- Quick sort không ổn định, nghĩa là vị trí tương đối của các phần tử bằng nhau có thể bị thay đổi sau khi được sắp xếp.

3.4.6. Bài tập luyện tập

- Sắp xếp một mảng số nguyên tăng dần bằng thuật toán quick sort.
- Sắp xếp một mảng số nguyên giảm dần bằng thuật toán quick sort.
- Sắp xếp một mảng số nguyên có phần tử trùng lặp bằng thuật toán quick sort.
- Sắp xếp một mảng số thực bằng thuật toán quick sort.
- Sắp xếp một mảng số nguyên theo thứ tự từ xa đến gần với một giá trị xác định bằng thuật toán quick sort.
- Sắp xếp một mảng số nguyên theo thứ tự từ gần đến xa với một giá trị xác định bằng thuật toán quick sort.
- Sắp xếp một mảng số nguyên bằng thuật toán quick sort sử dụng phương pháp "Hoare partition".
- Sắp xếp một mảng số nguyên bằng thuật toán quick sort sử dụng phương pháp "Lomuto partition".
- Sắp xếp một danh sách liên kết đơn có chứa các số nguyên bằng thuật toán quick sort.

3.5. Sắp xếp trộn (Merge sort)

3.5.1. Sơ lược về nguồn gốc

- Merge sort là một thuật toán sắp xếp đệ quy được phát minh bởi nhà khoa học máy tính John von Neumann vào năm 1945. Thuật toán merge sort được xây dựng trên nguyên tắc "chia để trị" (divide and conquer) và được thiết kế để giải quyết các vấn đề sắp xếp dữ liệu trên máy tính.
- Ban đầu, thuật toán merge sort được sử dụng để giải quyết vấn đề sắp xếp các bộ dữ liệu trên đĩa từ trong các máy tính đầu tiên. Sau đó, nó được ứng dụng rộng rãi trong các lĩnh vực khác như kinh doanh, khoa học, công nghệ và các lĩnh vực khác.
- Vì tính hiệu quả của nó, merge sort là một trong những thuật toán sắp xếp phổ biến nhất trong lĩnh vực khoa học máy tính và được sử dụng trong nhiều ứng dụng khác nhau.

3.5.2. Ý tưởng

- Ý tưởng chính của thuật toán merge sort là chia một danh sách cần sắp xếp thành các danh sách con nhỏ hơn, rồi sắp xếp từng danh sách con đó và sau đó trộn các danh sách con đã sắp xếp lại với nhau cho đến khi nhận được danh sách ban đầu đã sắp xếp.
- Thuật toán merge sort được thực hiện theo các bước sau:
 - Chia danh sách cần sắp xếp thành các danh sách con độc lập. Điều này được thực hiện bằng cách **chia đôi** danh sách ban đầu và tiếp tục chia đôi cho đến khi chỉ còn **một phần tử trong mỗi danh sách con**.
 - Sắp xếp các danh sách con này bằng cách sử dụng đệ quy. Tức là tiếp tục chia danh sách con đó thành các danh sách con nhỏ hơn và sắp xếp chúng cho đến khi chỉ còn một phần tử trong mỗi danh sách con.
 - Trộn các danh sách con đã sắp xếp lại với nhau để tạo ra danh sách lớn hơn, được sắp xếp đúng thứ tự. Quá trình trộn này được thực hiện bằng cách so sánh phần tử đầu tiên của mỗi danh sách con và chọn phần tử nhỏ hơn để đưa vào danh sách kết quả. Tiếp tục thực hiện cho đến khi một trong hai danh sách con đã được trộn hết.
 - Trả về danh sách đã được sắp xếp.

3.5.3. Cài đặt hàm

- Chúng ta cần xác định được các hàm để thực hiện thuật toán Merge sort:
 - Hàm Quick Sort với 2 đối số.
 - Hàm Quick Sort với 3 đối số.
 - Hàm Merge (Để trộn các danh sách lại với nhau).
- Định nghĩa hàm:
 - Merge sort với 2 đối số:

```
void mergeSort(int a[], int n) {
    mergeSort(a, 0, n - 1);
}
```

- Merge sort với 3 đối số:

```
void mergeSort(int a[], int left, int right)
{
    if (left < right)
    {
        int mid = (left + right) / 2;
        mergeSort(a, left, mid);
        mergeSort(a, mid + 1, right);
        merge(a, left, mid, right);
    }
}
```

- Merge:

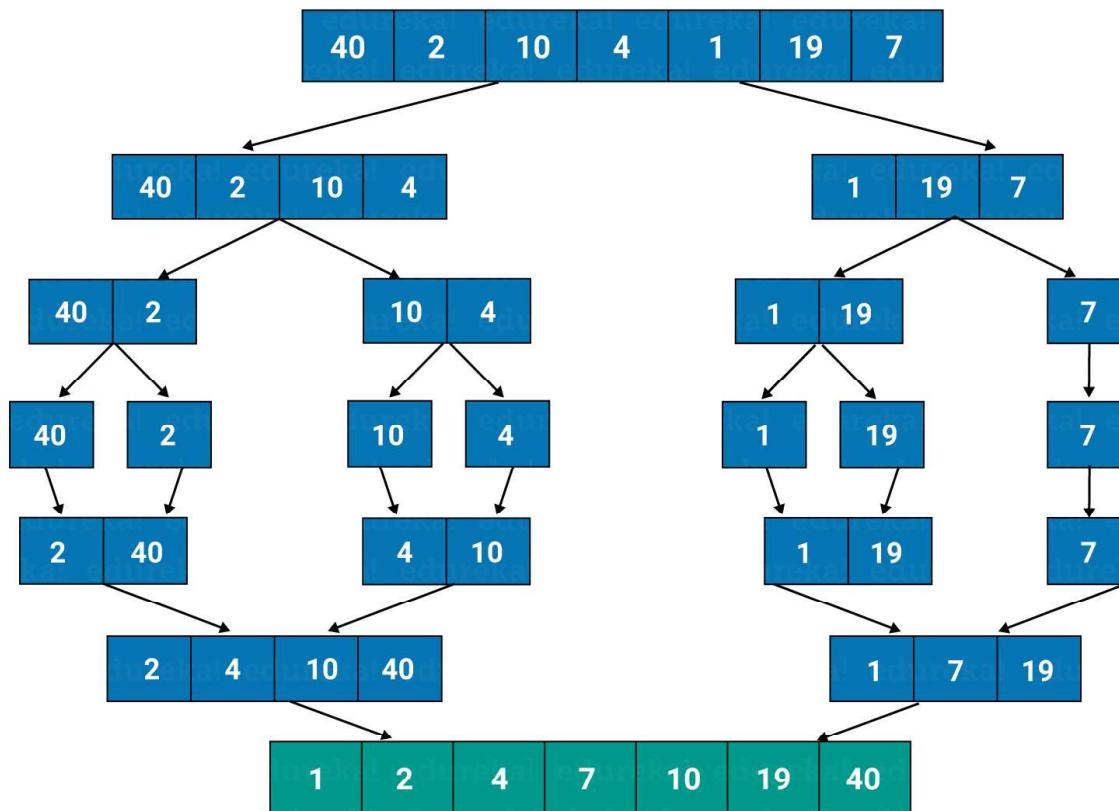
```
void merge(int a[], int left, int mid, int right)
{
    int m = left;
    int n = mid + 1;
    int k = 0;
    int* temp = new int[right - left + 1];
    while (!(m > mid) && (n > right))
    {
        if ((a[m] < a[n] && m <= mid && n <= right) || (n > right))
            temp[k++] = a[m++];
        else
            temp[k++] = a[n++];
    }
    for (int i = 0; i < k; i++)
        a[left + i] = temp[i];
    delete[] temp;
}
```

- Trong đó:

- **left, mid, right:** là các chỉ số của phần tử bên trái, giữa, bên phải mảng cần được sắp xếp
- **m:** là biến chạy trên mảng con thứ nhất
- **n:** là biến chạy trên mảng con thứ hai
- **temp:** là mảng tạm thời để lưu trữ các phần tử

3.5.4. Biểu diễn chi tiết thuật toán

- **Ví dụ:** Cho mảng a = {40,2,10,4,1,19,7}



3.5.5. Đánh giá thuật toán

3.5.5.1. Độ phức tạp

- Trung bình: $O(n \log n)$
- Tốt nhất: $O(n \log n)$
- Xấu nhất: $O(n \log n)$

3.5.5.2. Ưu điểm

- Merge sort là một thuật toán ổn định, nghĩa là nó không đổi vị trí các phần tử bằng nhau trong mảng đầu vào.
- Merge sort có thể được thực hiện trên các dữ liệu liên tục, bởi vì việc chia mảng thành các mảng con không cần phải dùng đến con trỏ hoặc thêm bộ nhớ phụ.
- Merge sort rất hiệu quả trong việc sắp xếp các danh sách liên kết, vì nó không cần truy cập phần tử bất kỳ đâu trong danh sách cho đến khi nó được sắp xếp.

3.5.5.3. Nhược điểm

- Merge sort sử dụng một mảng phụ để lưu trữ các giá trị đã được sắp xếp, do đó nó cần thêm không gian bộ nhớ để lưu trữ các mảng con và mảng phụ này.
- Merge sort yêu cầu một số lượng lớn các thao tác so sánh, điều này có thể khiến nó chậm hơn các thuật toán sắp xếp nhanh hơn như Quick sort.
- Merge sort không hiệu quả với dữ liệu nhỏ, vì chi phí của việc chia và ghép dữ liệu có thể vượt quá chi phí của việc sắp xếp các phần tử đó trực tiếp.

3.5.6. Bài tập luyện tập

- Sắp xếp một mảng số nguyên tăng dần bằng thuật toán merge sort.
- Sắp xếp một mảng số nguyên giảm dần bằng thuật toán merge sort.
- Sắp xếp một mảng số nguyên có phần tử trùng lặp bằng thuật toán merge sort.
- Sắp xếp một mảng số thực bằng thuật toán merge sort.
- Sắp xếp một mảng số nguyên theo thứ tự từ xa đến gần với một giá trị xác định bằng thuật toán merge sort.
- Sắp xếp một mảng số nguyên theo thứ tự từ gần đến xa với một giá trị xác định bằng thuật toán merge sort.
- Sắp xếp một danh sách liên kết đơn có chứa các số nguyên bằng thuật toán merge sort.
- Sắp xếp một danh sách liên kết đôi có chứa các số nguyên bằng thuật toán merge sort.

3.6. Sắp xếp vun đống (Heap sort)

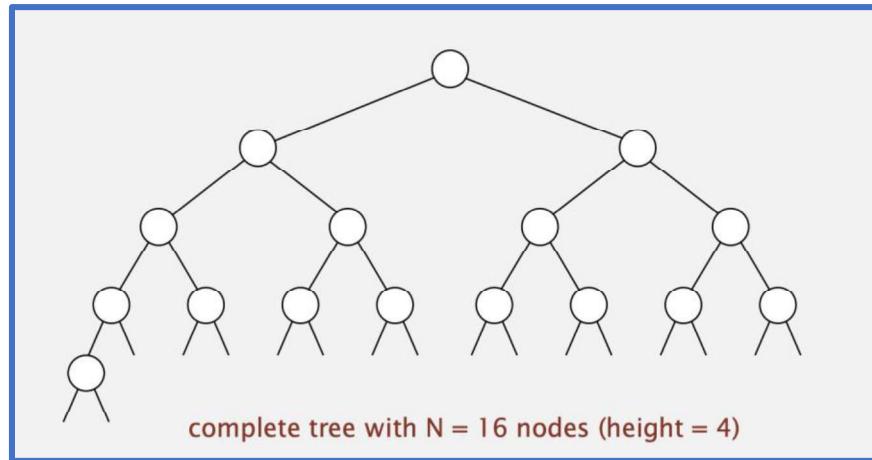
3.6.1. Sơ lược về lịch sử hình thành

HeapSort được đề xuất bởi J.W.J.Williams năm 1981, thuật toán không những đóng góp một phương pháp sắp xếp hiệu quả mà còn xây dựng một cấu trúc dữ liệu quan trọng để biểu diễn hàng đợi có độ ưu tiên: Cấu trúc dữ liệu Heap.

3.6.2. Cấu Trúc dữ liệu Binary Heaps

- **Complete binary tree (Cây nhị phân đầy đủ)**

Cây nhị phân (Binary tree) là một cấu trúc dữ liệu dạng cây, bao gồm các nút (node) liên kết với các cây nhị phân con bên trái và bên phải (left và right binary trees). Nếu cây rỗng, nó không có nút nào



- **Cây nhị phân đầy đủ (Complete binary tree):** Được cân bằng hoàn hảo, trừ mức cuối cùng.
- **Thuộc tính:** Chiều cao của cây nhị phân đầy đủ có N nút là $\lfloor \log_2 n \rfloor$
- **Chứng minh:** Chiều cao chỉ tăng lên khi N là một lũy thừa của 2.

3.6.3. Thao tác Heapify

```
void heapify(int arr[], int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] > arr[largest]) {
        largest = left;
    }
    if (right < n && arr[right] > arr[largest]) {
        largest = right;
    }
    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}
```

- Giải thích ý nghĩa của từng câu lệnh:

Bước 1 : Hàm heapify được khai báo với ba đối số: một mảng số nguyên arr, kích thước của mảng n, và chỉ số i của nút gốc cần heapify:

```
void heapify(int arr[], int n, int i)
```



TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN – ĐHQG-HCM BẢN HỌC TẬP CÔNG NGHỆ PHẦN MỀM

Bước 2: Một biến largest được khởi tạo với giá trị bằng chỉ số i, đại diện cho nút có giá trị lớn nhất trong ba nút gốc và hai nút con của nó.

```
int largest = i;
```

Bước 3 : Hai biến **left** và **right** được khởi tạo để lưu chỉ số của hai nút con của nút gốc đang xét. Biểu thức **2 * i + 1** và **2 * i + 2** tính toán chỉ số của hai nút con trong mảng dựa trên chỉ số của nút gốc i.

```
int left = 2 * i + 1;
int right = 2 * i + 2;
```

Bước 4: Nếu chỉ số của nút con trái (left) nhỏ hơn kích thước của mảng và giá trị của nút con trái lớn hơn giá trị của nút có giá trị lớn nhất (largest), thì largest được cập nhật thành chỉ số của nút con trái.

```
if (left < n && arr[left] > arr[largest]) {
    largest = left;
}
```

Bước 5 : Tương tự bước 4

```
if (right < n && arr[right] > arr[largest]){
    largest = right;
}
```

Bước 6: Nếu chỉ số của nút lớn nhất (largest) khác với chỉ số của nút gốc ban đầu (i), tức là giá trị nút con lớn hơn giá trị nút gốc, thì hoán đổi giá trị nút gốc và nút lớn nhất, sau đó đệ quy hàm heapify cho nút con có giá trị lớn nhất.

```
if (largest != i) {
    swap(arr[i], arr[largest]);
    heapify(arr, n, largest);
}
```

Lưu ý: Để xây dựng hàm Heap ta gọi hàm heapify với mọi nốt không phải là nốt lá:



```
for (int i = (n/2)-1; i >= 0; i--) {  
    heapify(arr, n, i);  
}
```

3.6.4. Cài đặt thuật toán Heap Sort

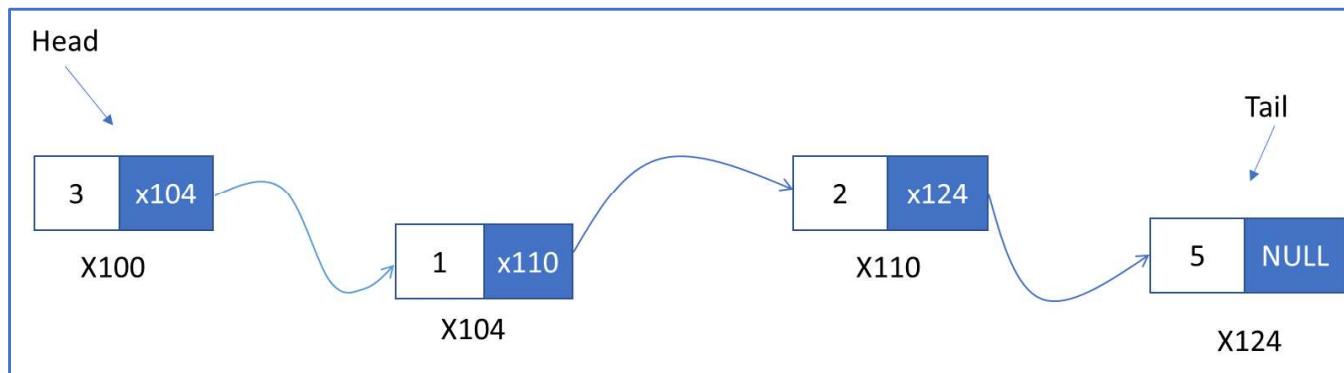
```
void heapSort(int arr[], int n) {  
    for (int i = n / 2 - 1; i >= 0; i--) {  
        heapify(arr, n, i);  
    }  
    for (int i = n - 1; i >= 0; i--) {  
        swap(arr[0], arr[i]);  
        heapify(arr, i, 0);  
    }  
}
```

Phần 2. CẤU TRÚC DỮ LIỆU

Chương 4. Danh sách liên kết đơn (Linked List)

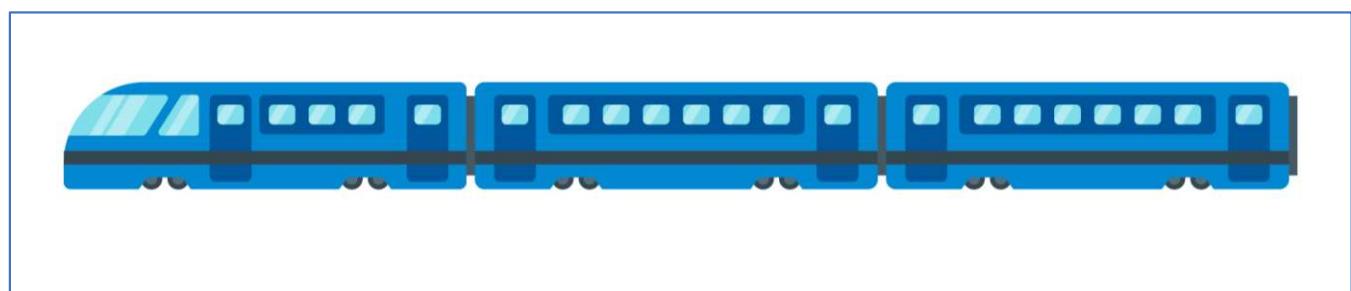
4.1. Sơ lược về lịch sử ra đời Linked list

Danh sách liên kết (Linked List) là một cấu trúc dữ liệu cổ điển trong khoa học máy tính. Nó xuất hiện từ thời kỳ đầu của máy tính và lập trình. Dù không thể xác định chính xác thời điểm đầu tiên danh sách liên kết được phát minh, nhưng có thể nói rằng nó ra đời cùng với ngành khoa học máy tính.



4.2. Nguyên Lý Hoạt Động của Linked

Danh sách liên kết (Linked List) có thể được minh họa giống như một **đoàn tàu hỏa**, trong đó **mỗi toa tàu là một nút (node)** trong danh sách, và các toa tàu được liên kết với nhau bằng **dây xích (con trỏ next)**.



- **Toa đầu (Head node):** Toa đầu tương trưng cho node đầu, con trỏ head sẽ trỏ đến node này.
- **Toa cuối (Tail node):** Toa cuối đại diện cho node cuối và node cuối trỏ đến NULL
- **Dây xích (Link):** các pointer (con trỏ next)

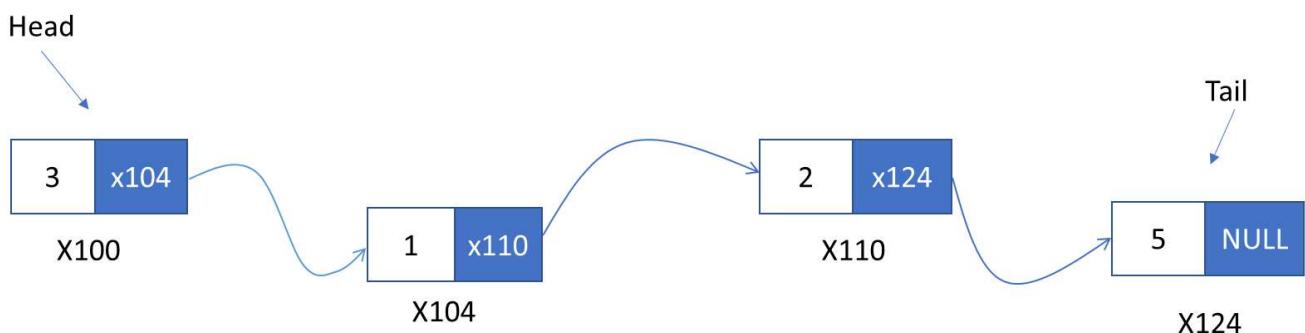
4.3. Định nghĩa Node và LIST

```
struct Node {
    int data;
    Node* next;
};

struct LIST {
    Node* head;
    Node* tail;
};
```

- **Cấu trúc Node:** gồm hai thành phần là dữ liệu (data – trong ví dụ là một dữ liệu kiểu int) và một con trỏ (pointer) đến nút kế tiếp trong danh sách.
- **Cấu trúc LIST:** gồm hai thành phần là **con trỏ đầu** danh sách (head) và **con trỏ đuôi** (tail).

- **Ví dụ:** Minh họa về 1 Danh sách liên kết đơn



4.4. Các thao tác với danh sách liên kết đơn

4.4.1. Cách tạo 1 danh sách liên kết đơn rỗng

```
void createEmptyList(LIST& L) {
    L.head = L.tail = NULL;
}
```

4.4.2. Hàm thêm vào đầu một danh sách liên kết đơn

```
void addHead(LIST& L, int x) {
    Node* p = new Node();
    p->data = x;
    p->next = NULL;
    if (L.head == NULL) {
        L.head = p;
        L.tail = L.head;
    }
    else {
```

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

4.4.3. Hàm in ra Linked List

```
void printList(LIST L) {
    Node* p;
    if (L.head == NULL) {
        cout << "Empty List.";
    }
    else {
        p = L.head;
        while (p) {
            cout << p->data << "\t";
            p = p->next;
        }
    }
    cout << endl;
}
```

4.4.4. Hàm xoá phần tử ở vị trí k

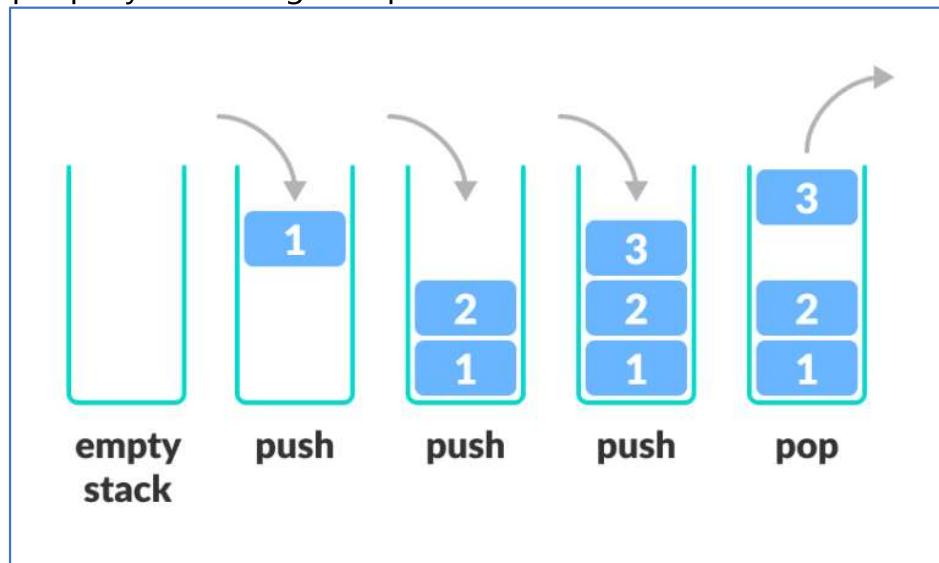
```
void Remove_k_th(LIST& L, int k) {
    if (L.head == NULL) {
        return;
    }
    Node* p = L.head;
    if (k == 1) {
        L.head = p->next;
        delete p;
        return;
    }
    Node* p1 = L.head;
    for (int i = 0; i < k - 2 && p1 != NULL; i++) {
        p1 = p1->next;
    }
    if (p1 == NULL || p1->next == NULL) {
        return;
    }
    Node* p2 = p1->next;
    p1->next = p2->next;
    delete p2;
}
```

Chương 5. Ngăn xếp (Stack)

5.1. Định nghĩa

Stack (ngăn xếp) là một cấu trúc dữ liệu trừu tượng, hoạt động theo nguyên lý "vào sau ra trước" (*Last In First Out – LIFO*, nghĩa là dữ liệu được nạp vào ngăn xếp trước sẽ được xử lý sau cùng và dữ liệu được nạp vào sau cùng được xử lý đầu tiên).

Có thể hình dung nó như cơ cấu của một hộp chứa đạn súng trường hoặc súng tiểu liên. Lắp đạn vào hay lấy đạn ra cũng chỉ ở 1 đầu hộp. Viên đạn mới nạp vào sẽ nằm ở đỉnh (top), còn viên nạp vào đầu tiên sẽ nằm ở đáy (bottom), điều này cũng có nghĩa là trong stack, các đối tượng có thể được thêm vào bất kỳ lúc nào, nhưng chỉ có đối tượng thêm vào sau cùng mới được phép lấy ra khỏi ngăn xếp.



Hình 5.1. Cấu trúc ngăn xếp

5.2. Các thao tác cơ bản trên stack

Trên stack sẽ có 2 thao tác cơ bản chính là **push** và **pop**. Thao tác **push** sẽ bổ sung 1 phần tử vào đỉnh của ngăn xếp, nghĩa là sau các phần tử đã có trong ngăn xếp. Trong khi đó, thao tác **pop** lấy ra và trả về phần tử đang đứng ở đỉnh của ngăn xếp.

Hình 5.1 mô tả cách hoạt động của stack với 2 thao tác **push** và **pop** trên stack các phần tử là số nguyên.

Ngoài những thao tác cơ bản trên thì còn có 1 số thao tác khác:

- Thao tác **top()**: trả về vị trí phần tử nạp sau cùng vào ngăn xếp
- Thao tác **size()**: trả về số lượng phần tử được lưu trong ngăn xếp
- Thao tác **isEmpty()**: kiểm tra xem ngăn xếp có phải rỗng hay không

5.3. Một số ứng dụng của stack



Dựa theo cách thức tổ chức, cơ chế hoạt động của ngăn xếp, ta có một số ứng dụng sau:

- Đổi cơ số trong hệ nhị phân
- Biểu thức số học và ký pháp Ba Lan
- Xây dựng ứng dụng theo dõi lịch sử trình duyệt web
- Xây dựng chức năng undo/redo trong các phần mềm
- Kiểm tra tính hợp lệ của các dấu ngoặc trong biểu thức
- Sử dụng khử đệ quy khi lập trình

...

5.4. Các thao tác trên ngăn xếp (stack)

Trong phần này, chúng ta sẽ giới thiệu 2 cách cài đặt chính trên ngăn xếp (stack) là dùng mảng và dùng danh sách liên kết, đồng thời cũng giới thiệu cách xây dựng, cách sử dụng 2 thao tác push và pop trên thư viện STL

5.4.1. Cài đặt ngăn xếp (stack) bằng cách sử dụng mảng:

Đây là cách đơn giản nhất để cài đặt trên ngăn xếp. Ta nạp các phần tử theo thứ tự từ trái sang phải. Có biến lưu giữ chỉ số các phần tử ở đầu ngăn xếp. Thao tác cài đặt ngăn xếp dùng mảng được thực hiện qua các bước sau:

Bước 1: Khai báo cấu trúc dữ liệu của ngăn xếp

Ngăn xếp sẽ bao gồm 1 mảng **a[]** chứa các phần tử trong ngăn xếp và 1 biến **top** để lưu giữ chỉ số của phần tử đầu ngăn xếp

Định nghĩa:

```
struct Stack {  
    int top;  
    int a[MAX];  
};
```

Bước 2: Viết các hàm thực hiện các thao tác trên ngăn xếp, bao gồm các thao tác chính:

- Thao tác khởi tạo ngăn xếp
- Thao tác kiểm tra ngăn xếp có rỗng hay không
- Thao tác kiểm tra ngăn xếp có đầy hay không
- Thao tác thêm phần tử vào ngăn xếp
- Thao tác xóa một phần tử ra khỏi ngăn xếp
- Thao tác lấy giá trị trên đỉnh của ngăn xếp

Hàm khởi tạo ngăn xếp:

```
void createStack(Stack& s) {  
    s.top = -1;  
}
```



TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN – ĐHQG-HCM BẢN HỌC TẬP CÔNG NGHỆ PHẦN MỀM

Hàm kiểm tra ngăn xếp có rỗng hay không

```
bool isEmpty(Stack s) {
    if (s.top < 0) { // (s.top == -1)
        return true;
    }
    else {
        return false;
    }
}
```

Hàm kiểm tra ngăn xếp có đầy hay không

```
bool isFull(Stack s) {
    if (s.top >= MAX) {
        return true;
    }
    else {
        return false;
    }
}
```

Hàm thêm phần tử vào ngăn xếp

```
bool push(Stack& s, int x) {
    if (isFull(s) == 0) {
        s.top++;
        s.a[s.top] = x;
        cout << x << " pushed into stack" << endl;
        return true;
    }
    else {
        cout << "Stack Overflow";
        return false;
    }
}
```

Hàm xóa 1 phần tử ra khỏi ngăn xếp

```
int pop(Stack& s) {
    if (isEmpty(s) == 0) {
        int x = s.a[s.top];
        s.top--;
        return x;
    }
    else {
        cout << "Stack Underflow";
        return 0;
    }
}
```



```
}
```

Hàm lấy giá trị trên đỉnh của ngăn xếp

```
int peek(Stack& s) {
    if (isEmpty(s) == 0) {
        int x = s.a[s.top];
        return x;
    }
    else {
        cout << "Stack is Empty";
        return 0;
    }
}
```

Thực thi chương trình:

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

#define MAX 1000

struct Stack {
    int top;
    int a[MAX];
};

void createStack(Stack& s) {
    s.top = -1;
}

bool isEmpty(Stack s) {
    if (s.top < 0) { // (s.top == -1)
        return true;
    }
    else {
        return false;
    }
}

bool isFull(Stack s) {
    if (s.top >= MAX) {
        return true;
    }
}
```



```
        else {
            return false;
        }

bool push(Stack& s, int x) {
    if (isFull(s) == 0) {
        s.top++;
        s.a[s.top] = x;
        cout << x << " pushed into stack" << endl;
        return true;
    }
    else {
        cout << "Stack Overflow";
        return false;
    }
}

int pop(Stack& s) {
    if (isEmpty(s) == 0) {
        int x = s.a[s.top];
        s.top--;
        return x;
    }
    else {
        cout << "Stack Underflow";
        return 0;
    }
}

int peek(Stack& s) {
    if (isEmpty(s) == 0) {
        int x = s.a[s.top];
        return x;
    }
    else {
        cout << "Stack is Empty";
        return 0;
    }
}

int main()
```

```
{  
    Stack s;  
    createStack(s);  
    push(s, 5);  
    push(s, 10);  
    push(s, 15);  
    cout << pop(s) << " popped from stack" << endl;  
  
    cout << "Elements present in stack: ";  
    while (!isEmpty(s))  
    {  
        cout << peek(s) << " ";  
        pop(s);  
    }  
    return 0;  
}
```

Output:

```
5 pushed into stack  
10 pushed into stack  
15 pushed into stack  
15 popped from stack  
Elements present in stack: 10 5
```

Ưu điểm của việc cài đặt trên:

- Dễ dàng cài đặt
- Không có liên quan đến con trỏ

Nhược điểm:

- Do đây là mảng tĩnh nên bị giới hạn số phần tử, vì thế cần phải xác định số lượng phần tử trước.
=> Từ đó, chúng ta có 1 cách cải tiến hơn cho việc cài đặt này.

5.4.2. Cài đặt ngăn xếp (stack) sử dụng danh sách liên kết (Linked List)

Thao tác cài đặt ngăn xếp dùng danh sách liên kết được thực hiện qua các bước sau:

Bước 1: Khai báo cấu trúc dữ liệu của ngăn xếp

```
struct stackNode {  
    int data;  
    stackNode* next;  
};
```

Cấp phát bộ nhớ một node trên ngăn xếp



```
stackNode* newNode(int data) {
    stackNode* p = new stackNode();
    p->data = data;
    p->next = NULL;
    return p;
}
```

Bước 2: Viết các hàm thực hiện các thao tác trên ngăn xếp, bao gồm các thao tác chính:

- Thao tác kiểm tra ngăn xếp có rỗng hay không
- Thao tác thêm phần tử vào ngăn xếp
- Thao tác xóa một phần tử ra khỏi ngăn xếp
- Thao tác lấy giá trị trên đỉnh của ngăn xếp

Hàm kiểm tra ngăn xếp có rỗng hay không

```
int isEmpty(stackNode* root) {
    return !root;
}
```

Hàm thêm phần tử vào ngăn xếp

```
void push(stackNode** root, int data) {
    stackNode* p = newNode(data);
    p->next = *root;
    *root = p;
    cout << data << " pushed to stack\n";
}
```

Hàm xóa một phần tử ra khỏi ngăn xếp

```
int pop(stackNode** root) {
    if (isEmpty(*root)) {
        return INT_MIN;
    }
    stackNode* temp = *root;
    *root = (*root)->next;
    int popped = temp->data;
    free(temp);
    return popped;
}
```

Hàm lấy giá trị trên đỉnh của ngăn xếp

```
int peek(stackNode* root) {
    if (isEmpty(root)) {
        return INT_MIN;
    }
    return root->data;
}
```



Thực thi chương trình:

```
#include<iostream>
#include <stdio.h>
#include <conio.h>
using namespace std;

struct stackNode {
    int data;
    stackNode* next;
};

stackNode* newNode(int data) {
    stackNode* p = new stackNode();
    p->data = data;
    p->next = NULL;
    return p;
}

int isEmpty(stackNode* root) {
    return !root;
}

void push(stackNode** root, int data) {
    stackNode* p = newNode(data);
    p->next = *root;
    *root = p;
    cout << data << " pushed to stack\n";
}

int pop(stackNode** root) {
    if (isEmpty(*root)) {
        return INT_MIN;
    }
    stackNode* temp = *root;
    *root = (*root)->next;
    int popped = temp->data;
    free(temp);
    return popped;
}

int peek(stackNode* root) {
    if (isEmpty(root)) {
```

```

        return INT_MIN;
    }
    return root->data;
}

int main() {
    stackNode* root = NULL;

    push(&root, 10);
    push(&root, 20);
    push(&root, 30);

    cout << pop(&root) << " popped from stack" << endl;
    cout << "Top element is " << peek(root) << endl;
    cout << "Elements present in stack : ";

    while (!isEmpty(root)) {
        cout << peek(root) << " ";
        pop(&root);
    }

    return 0;
}

```

5.4.3. Cách sử dụng các thao tác trên ngăn xếp sử dụng thư viện STL trong C++

- Tên thư viện: stack
- Cách khai báo thư viện:

```
#include<stack>
```

- Cách khai báo 1 ngăn xếp (stack) có tên là s, chứa các phần tử là số nguyên: stack<int>s;
- Cách khai báo các thao tác trên thư viện stack:
 - Thao tác kiểm tra ngăn xếp có rỗng hay không: s.empty()
 - Thao tác thêm phần tử vào ngăn xếp: s.push(num);
 - Thao tác xóa phần tử ra khỏi ngăn xếp: s.pop();
 - Thao tác lấy phần tử trên đỉnh của ngăn xếp: s.top();
 - Thao tác lấy số phần tử trên ngăn xếp: s.size();

***Thực thi chương trình mẫu:**

```
#include <iostream>
#include <stack>
using namespace std;
int main() {
```



TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN – ĐHQG-HCM BẢN HỌC TẬP CÔNG NGHỆ PHẦN MỀM

```
stack<int> stack;
stack.push(1);
stack.push(2);
stack.push(4);
stack.push(5);
int num = 0;
stack.push(num);
stack.pop();
stack.pop();
stack.pop();

while (!stack.empty()) {
    cout << stack.top() << " ";
    stack.pop();
}
}
```

Output:

```
2 1
```

Chương 6. Hàng đợi (Queue)

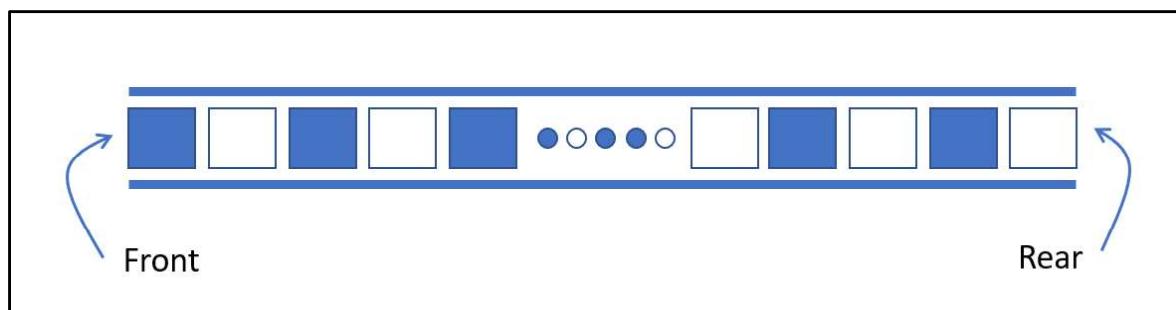
6.1. Khái niệm hàng đợi

6.1.1. Sơ lược về lịch sử hình thành:

- Khái niệm về hàng đợi đã tồn tại từ rất lâu trong cuộc sống hàng ngày trước khi nó được áp dụng vào lĩnh vực khoa học máy tính. Tuy nhiên, việc formalize và áp dụng hàng đợi vào lĩnh vực khoa học máy tính bắt đầu từ những năm 1940 và 1950.
- Trong lĩnh vực toán học, một số nhà toán học đã nghiên cứu về hàng đợi trong thế kỷ 19, bao gồm Agner Krarup Erlang, người đã nghiên cứu về xếp hàng trong lĩnh vực lưu lượng cuộc gọi điện thoại.
- Trong khoa học máy tính, hàng đợi được phát triển để giải quyết các vấn đề liên quan đến việc quản lý và tổ chức dữ liệu. Vào những năm 1950, hàng đợi đã trở thành một khái niệm quan trọng trong lĩnh vực lập trình và tính toán.
- Ngôn ngữ lập trình FORTRAN, được phát triển bởi IBM vào những năm 1950, đã cung cấp các phương pháp để triển khai hàng đợi trong chương trình. Điều này đã tạo điều kiện thuận lợi cho việc phát triển và sử dụng hàng đợi trong các ứng dụng tính toán.

6.1.2. Khái niệm

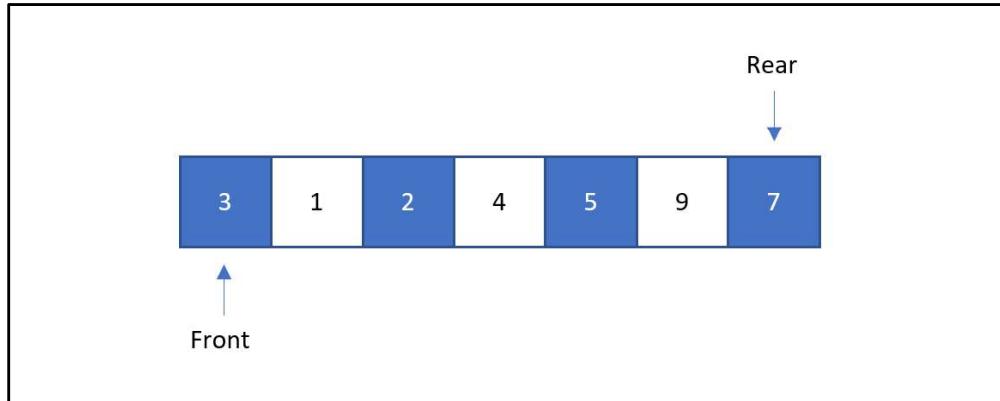
- Hàng đợi là cấu trúc dữ liệu tuyến tính mở ở cả hai đầu và các thao tác được thực hiện theo thứ tự First In First Out (FIFO – Vào Trước Ra Trước) nghĩa là phần tử nào được đẩy vào hàng đợi đầu tiên thì các thao tác của hàng đợi sẽ thực hiện với phần tử đó đầu tiên.



6.2. Khởi tạo

6.2.1. Khởi tạo queue trên mảng (array):

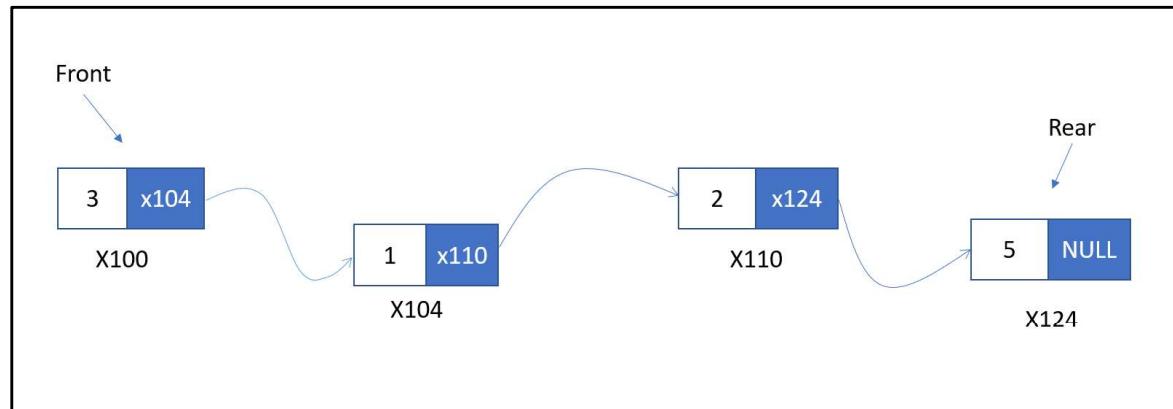
```
typedef struct queue {  
    char a[MAX];  
    int front; //chỉ số của phần tử đầu trong Queue  
    int rear; //chỉ số của phần tử cuối trong Queue  
} QUEUE;
```



6.2.2. Khởi tạo queue trên danh sách liên kết đơn (singly linked-list):

```
// Cấu trúc của một node
struct Node {
    int data;
    Node* next;
};

// Cấu trúc của một DSLK
struct QUEUE {
    Node* Front;
    Node* Rear;
};
```



6.3. Ứng dụng của Queue

6.3.1. Trong các thuật toán:

- Quay lui (Backtracking): Trong thuật toán quay lui, queue có thể được sử dụng để lưu trữ các trạng thái tạm thời hoặc các lựa chọn tiếp theo để thử.
- Vét cạn (Breadth-First Search): Trong thuật toán vét cạn, queue được sử dụng để duyệt qua các trạng thái hoặc khả năng theo cấp độ (level).

- Queue cũng có thể được sử dụng trong các thuật toán khác như tìm kiếm theo chiều sâu (Depth-First Search), thuật toán Dijkstra và nhiều thuật toán tìm kiếm và duyệt đồ thị khác.
- Ngoài ra Queue còn được sử dụng để khử đệ quy trong hầu hết các bài toán

6.3.2. Trong thực tế:

- Trong hệ thống tin nhắn: Để lưu trữ và sắp xếp các tin nhắn ngắn nắp.
- Trong hệ thống hàng hóa vận chuyển: Giúp kiểm soát lưu lượng và đảm bảo đúng thứ tự vận chuyển.
- Trong hệ thống giao thông: Quản lý phương tiện để đảm bảo lưu thông thuận lợi.
- ...

6.4. Thao tác trên Queue

6.4.1. Thao tác trên mảng

6.4.1.1. Khởi tạo queue rỗng

```
void createQueue(QUEUE& q) {
    q.front = -1;
    q.rear = -1;
}
```

Front = Rear = -1

6.4.1.2. Kiểm tra queue rỗng

```
bool isEmpty(QUEUE q) {
    if (q.front == -1) {
        return 1;
    }
    return 0;
}
```

(*) Queue rỗng hàm trả về true (1) ngược lại trả về false (0)



6.4.1.3. Kiểm tra queue đầy

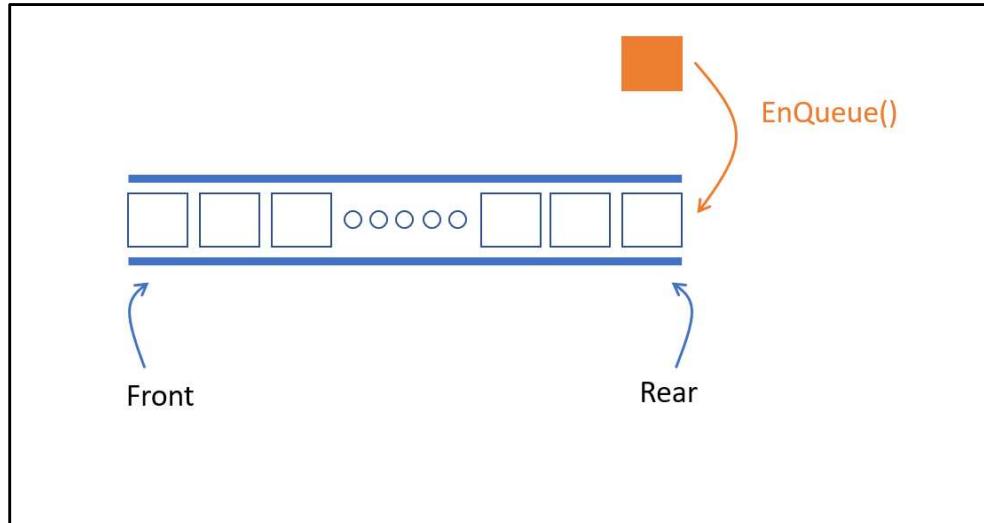
```
bool isFull(QUEUE q) {
    if (q.rear - q.front + 1 == MAX) {
        return 1;
    }
    return 0;
}
```

(*) Queue đầy hàm trả về true (1) ngược lại trả về false (0)

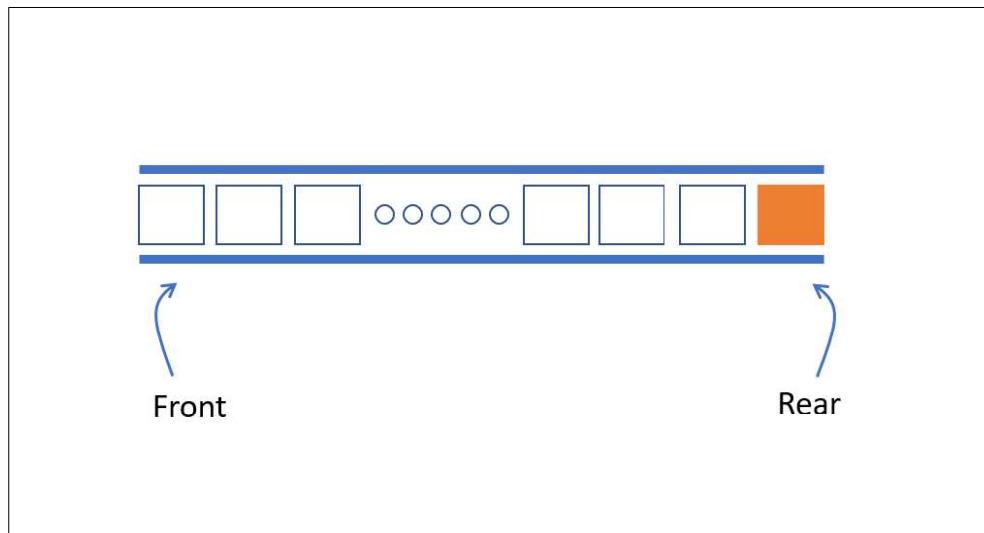
6.4.1.4. Thêm một phần tử vào queue

```
void enqueue(QUEUE& q, char x) {
    int f, r;
    if (isFull(q) == 1) {
        cout << "Queue da day, khong the them phan tu!\n";
    }
    else {
        if (q.front == -1) {
            q.front = 0;
            q.rear = -1;
        }
        if (q.rear == MAX - 1) { //Queue đầy ảo
            f = q.front;
            r = q.rear;
            for (int i = f; i <= r; i++) {
                q.a[i - f] = q.a[i];
            }
            q.front = 0;
            q.rear = r - f;
        }
        q.rear++;
        q.a[q.rear] = x;
    }
}
```

- Thực hiện EnQueue



- Queue sau khi EnQueue



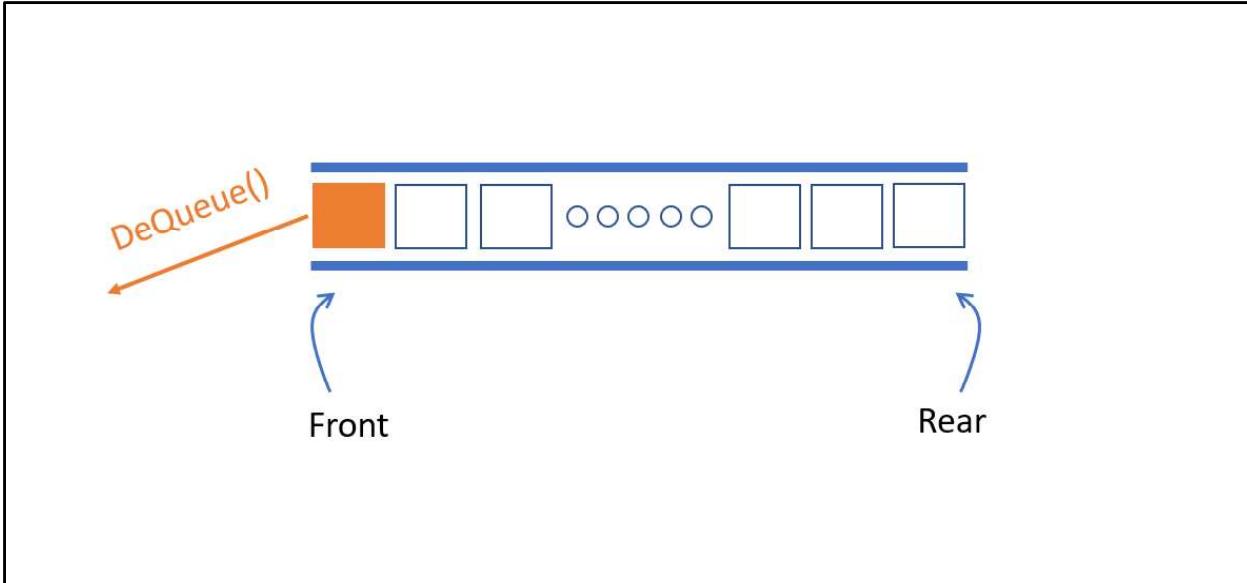
(*) Sau khi thực hiện EnQueue() độ dài queue tăng thêm 1

6.4.1.5. Lấy một phần tử ra khỏi queue

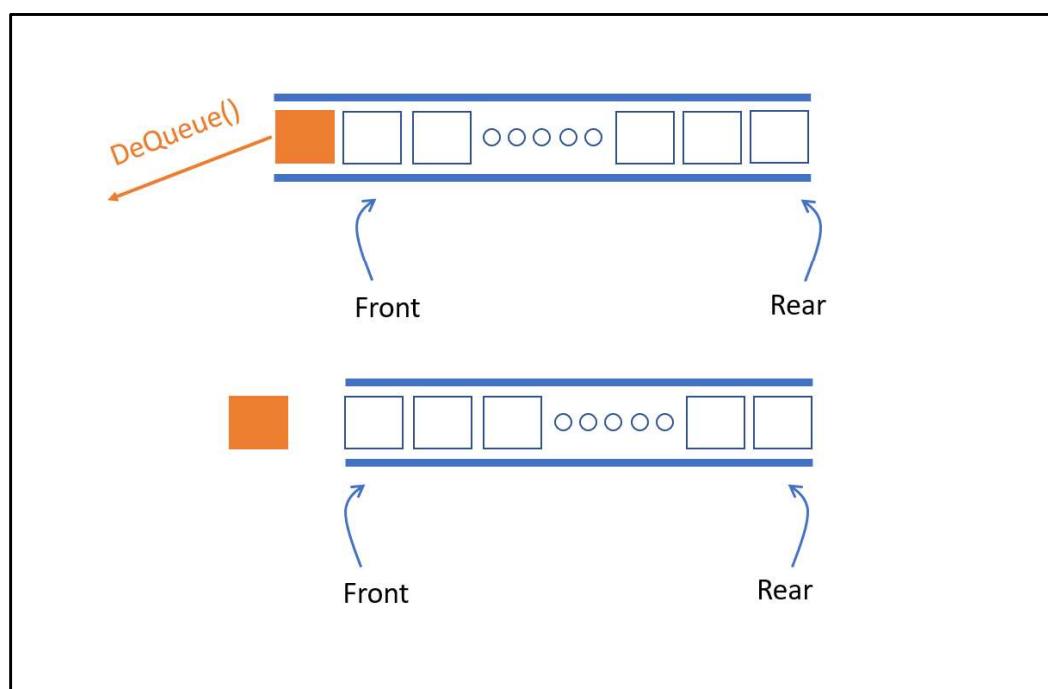
```
bool DeQueue(QUEUE& q, int& x)
{
    if (isEmpty(q) == 0)
    {
        x = q.a[q.Front];
        q.Front++;
        //Trường hợp queue chỉ có 1 phần tử => khi lấy ra queue rỗng
        if (q.Front > q.Rear)
        {
            q.Front = -1;
            q.Rear = -1;
        }
    }
}
```

```
        }
        return 1;
    }
    cout << "Queue rong!";
    return 0;
}
```

- Khi thực hiện DeQueue



- Sau khi thực hiện DeQueue



(*) Sau khi thực hiện DeQueue độ dài queue giảm đi 1

6.4.2. Thao tác trên danh sách liên kết đơn

6.4.2.1. Khởi tạo queue rỗng

```
void createQueue(QUEUE& q) {
    q.front = NULL;
    q.rear = NULL;
}
```

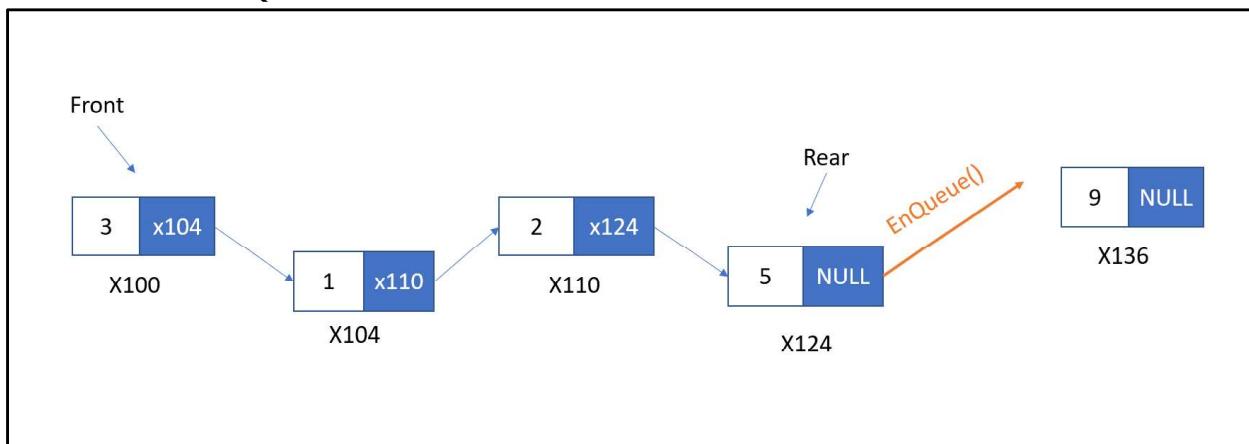
6.4.2.2. Kiểm tra queue rỗng

```
bool isEmpty(QUEUE& Q) {
    if (Q.front == NULL) { //Queue rỗng
        return 1;
    }
    return 0;
}
```

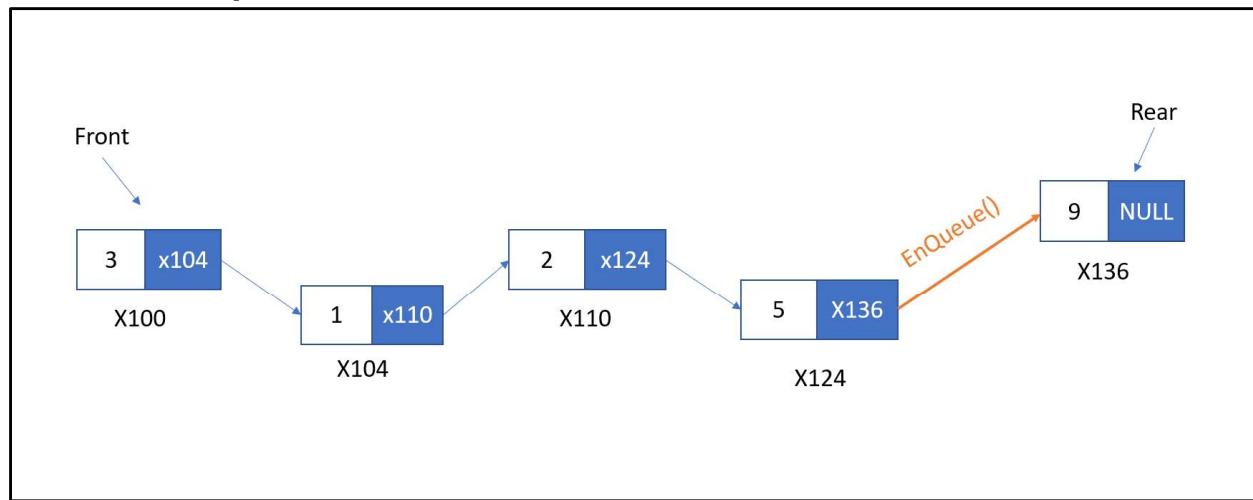
6.4.2.3. Thêm một phần tử vào queue

```
void enqueue(QUEUE& Q, Node* p) {
    if (Q.front == NULL) {
        Q.front = Q.rear = p;
    }
    else {
        Q.rear->next = p;
        Q.rear = p;
    }
}
```

- Trước khi EnQueue



- Sau khi EnQueue

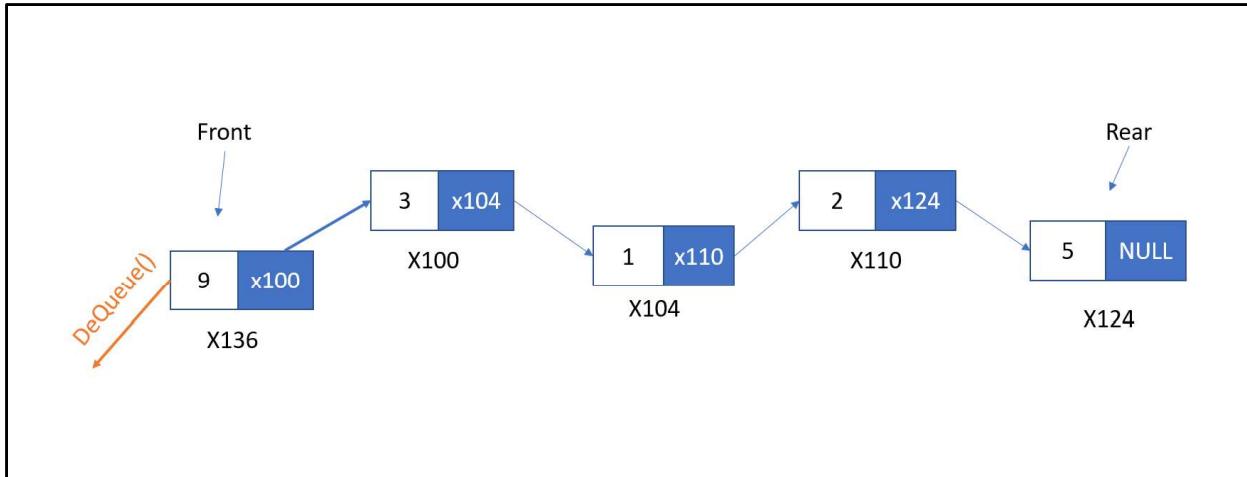


- Node tiếp theo (**pNext**) của node Rear (**X124**) sẽ giữ địa chỉ node được thêm vào (**X136**)
- Node Rear hiện tại chuyển thành node vừa được thêm vào (**X136**)
- Hàm EnQueue trong danh sách liên kết đơn tương tự như hàm AddTail

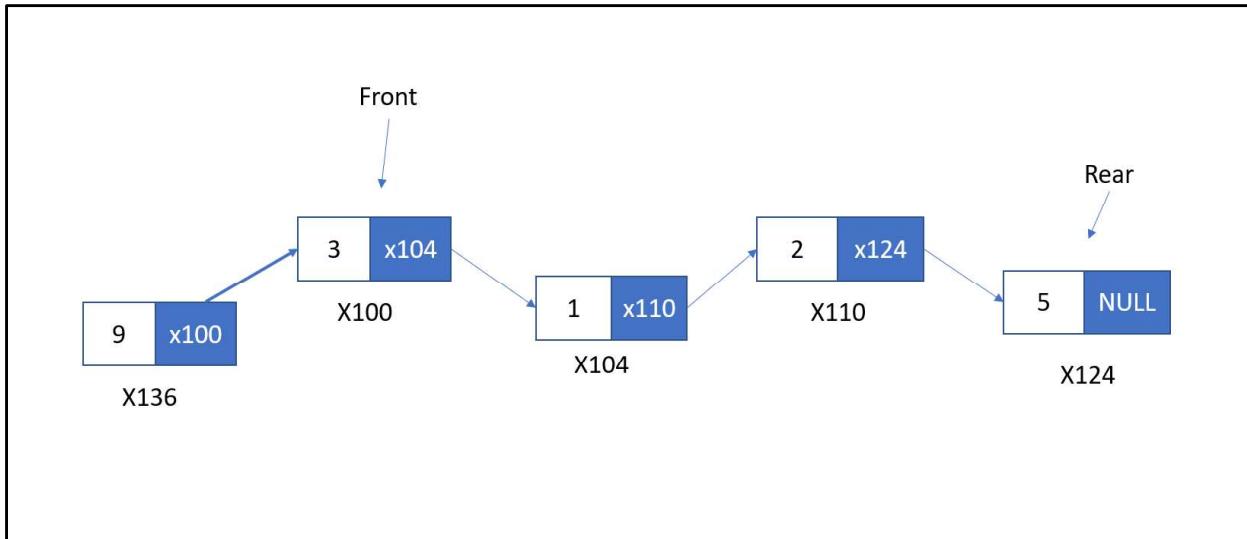
6.4.2.4. Lấy một phần tử ra khỏi queue

```
int deQueue(QUEUE& Q, int& x) {
    Node* p;
    if (isEmpty(Q) == 0) {
        p = Q.front;
        x = p->data;
        Q.front = Q.front->next;
        //Nếu ban đầu queue có 1 phần tử, sau khi DeQueue thì queue rỗng
        if (Q.front == NULL) {
            Q.rear = NULL;
        }
        return 1;
    }
    return 0;
}
```

- Trước khi DeQueue



- Sau khi DeQueue



(!) Nhận xét: Để hoàn thiện chương trình, ngoài các hàm thao tác ra chúng ta cần tạo thêm một số hàm như là CreateNode (Tạo node trong danh sách liên kết đơn), hàm Input (Nhập vào queue), hàm Output (Xuất các phần tử trong queue),...

6.5. Bài toán thường gặp

6.5.1. Chuỗi palindromes

Khái niệm: Một chuỗi palindromes là một chuỗi có thể được đọc xuôi hay ngược mà vẫn cho kết quả giống nhau. Ví dụ, "level" và "radar".

Giải thuật sử dụng queue:

- Ta có thể sử dụng một queue để kiểm tra palindrome bằng cách: đẩy nửa cuối chuỗi vào queue, sau đó so sánh các ký tự còn lại với các ký tự được lấy từ queue. Nếu tất cả các ký tự khớp, chuỗi là palindrome.



- Độ phức tạp: $O\left(\frac{n}{2}\right)$, n: là độ dài của chuỗi.
- Cài đặt hàm tham khảo:

```
bool isPalindromes(string str) {  
    QUEUE q;  
    createQueue(q);  
    int n = str.size();  
    // Đưa nửa sau hàng đợi vào chuỗi  
    for (int i = n - 1; i > n/2; i--) {  
        enqueue(q, str[i]);  
        str.pop_back();  
    }  
    // So sánh với các phần tử còn lại  
    int i = 0;  
    char x;  
    while (!isEmpty(q)) {  
        dequeue(q, x);  
        if (x != str[i]) {  
            return false;  
        }  
        i++;  
    }  
    return true;  
}
```

6.5.2. Giải thuật dermerging:

- Được sử dụng để phân tách một danh sách (hay mảng) gồm các phần tử có thuộc tính nhất định thành hai danh sách riêng biệt, mỗi danh sách chứa các phần tử có thuộc tính khác nhau.
- Thông thường, thuộc tính được sử dụng để phân chia là thuộc tính boolean, ví dụ: giới tính (nam, nữ), trạng thái (đã kích hoạt, chưa kích hoạt),...
- Độ phức tạp: $O\left(\frac{n}{2}\right)$, n: là độ dài của chuỗi.

6.5.3. Chuyển từ biểu thức trung tố sang ký pháp Ba Lan

6.6. Bài tập

BT 6.6.1. Tạo số nhị phân từ 1 đến N bằng cách sử dụng Hàng đợi

BT 6.6.2. Cho một chuỗi s chỉ chứa các ký tự '(', ')', '{', '}', '[' và ']', hãy xác định xem chuỗi đầu vào có hợp lệ hay không.

BT 6.6.3. Một chuỗi đầu vào hợp lệ nếu:



- BT 6.6.4.** Dấu ngoặc mở phải được đóng bằng cùng một loại dấu ngoặc.
- BT 6.6.5.** Dấu ngoặc mở phải được đóng theo đúng thứ tự.
- BT 6.6.6.** Mỗi dấu ngoặc đóng có một dấu ngoặc mở tương ứng cùng loại.
- BT 6.6.7.** Viết chương trình chuyển biểu thức trung tố sang biểu thức ký pháp Ba Lan sử dụng queue.

Chương 7. Cây (Tree)

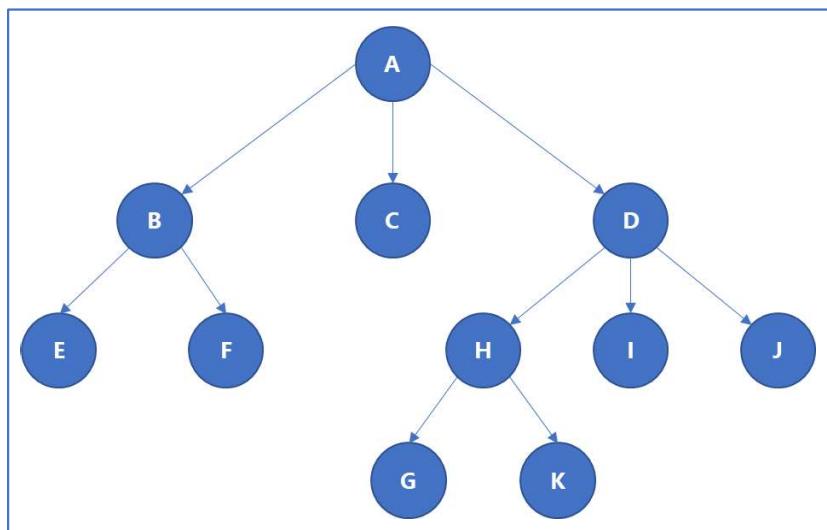
7.1. Cấu trúc cây

7.1.1. Khái niệm cây

Cây là một tập hợp các phần tử, hay còn gọi là các nút, gồm:

- Nút gốc (root).
- Các nút còn lại chia thành các tập con T_1, T_2, \dots, T_n . Trong đó các T_i cũng là một cây (cây con)

- **Ví dụ 7.1:**



Hình 7.1 Ví dụ minh họa 1 cấu trúc cây.

7.1.2. Tính chất cây

7.1.2.1. Độ tuổi của nút:

Là số cây con của nút đó.

- **Ví dụ 7.2:** Xét cây trong **ví dụ 7.1**:

- Độ tuổi của nút A là 3
- Độ tuổi của nút B là 2
- Độ tuổi của nút C là 0

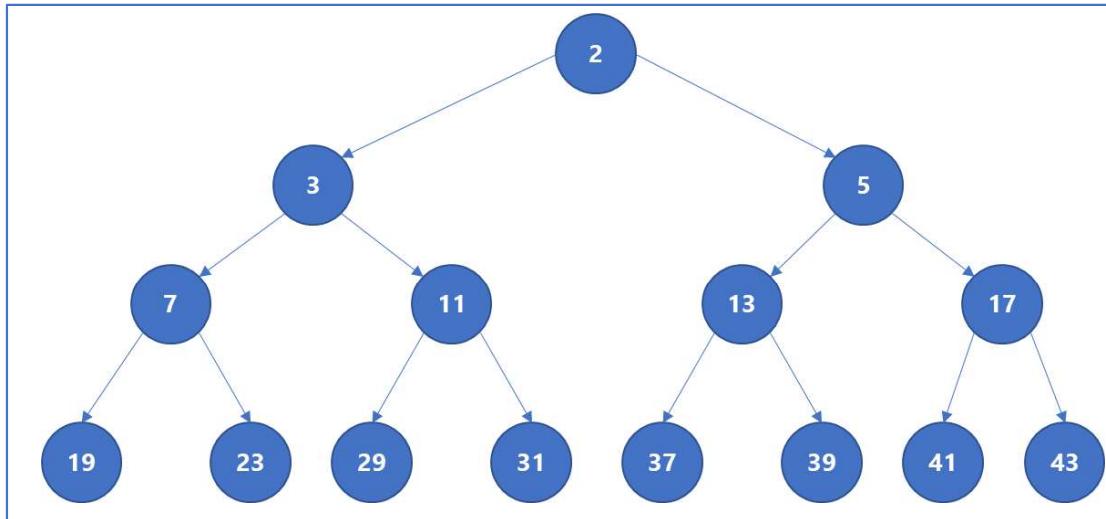
(?) *Câu hỏi: Hãy liệt kê độ tuổi của các nút H, D, G, K của cây ở ví dụ 7.1*

7.1.2.2. Độ tuổi của cây

- Là độ tuổi của nút có độ tuổi lớn nhất trên cây.
- Cây có độ tuổi n được gọi là cây n-phân.

- **Ví dụ 7.3:** Độ tuổi của cây trong **ví dụ 7.1** là 3, gọi là cây tam phân.

(?) *Câu hỏi: Xác định độ tuổi của cây trong hình dưới đây:*



Hình 7.2. Minh họa 1 cấu trúc cây để xác định bậc.

7.1.2.3. Nút lá

Là nút có bậc bằng 0 (không chứa cây con nào).

- **Ví dụ 7.4:** Ở **Hình 7.2** có nút 19, 23 là nút lá.

(?) Câu hỏi:

- Liệt kê tất cả các nút lá của cây trong **Hình 7.2**.
- Nút C của cây ở **Hình 7.1** có phải nút lá không?
- Liệt kê tất cả các nút lá của cây trong **Hình 7.1**.

7.1.2.4. Nút nhánh

- Là nút có bậc khác 0, và không phải là nút gốc.
- Nút nhánh hay còn gọi là nút trung gian.

- **Ví dụ 7.5:** Ở **Hình 7.2** các nút nhánh là 3, 5, 7, 11, 13, 17.

(?) Câu hỏi:

- Nút I của cây trong **Hình 7.1** có phải nút nhánh không?
- Liệt kê tất cả các nút nhánh của cây trong **Hình 7.1**.

7.1.2.5. Mức của nút

- Mức của gốc $T_0 = 0$.
- Xét T_1, T_2, \dots, T_n là các cây con của T_0 .
- Khi đó: Mức (T_1) = Mức (T_2) = ... = Mức (T_n) = Mức(T_0) + 1.

- **Ví dụ 7.6:** Xét cây trong **Hình 7.2**, ta có:

- Mức của nút 2 là 0.
- Mức của nút 3 và nút 5 là 1.
- Mức của nút 7, 11, 13, 17 là 2.

(?) Câu hỏi: Bạn hãy cho biết mức của nút G của cây trong **Hình 7.1**.

7.1.2.6. Chiều cao

Chiều cao (hay còn gọi là chiều sâu) của cây: là mức lớn nhất của các nút lá.

- **Ví dụ 7.7:** Cây trong **Hình 7.2** có mức là 3.

(?) *Câu hỏi: Xác định mức của cây trong **Hình 7.1**.*

7.1.2.7. Độ dài đường đi của nút x

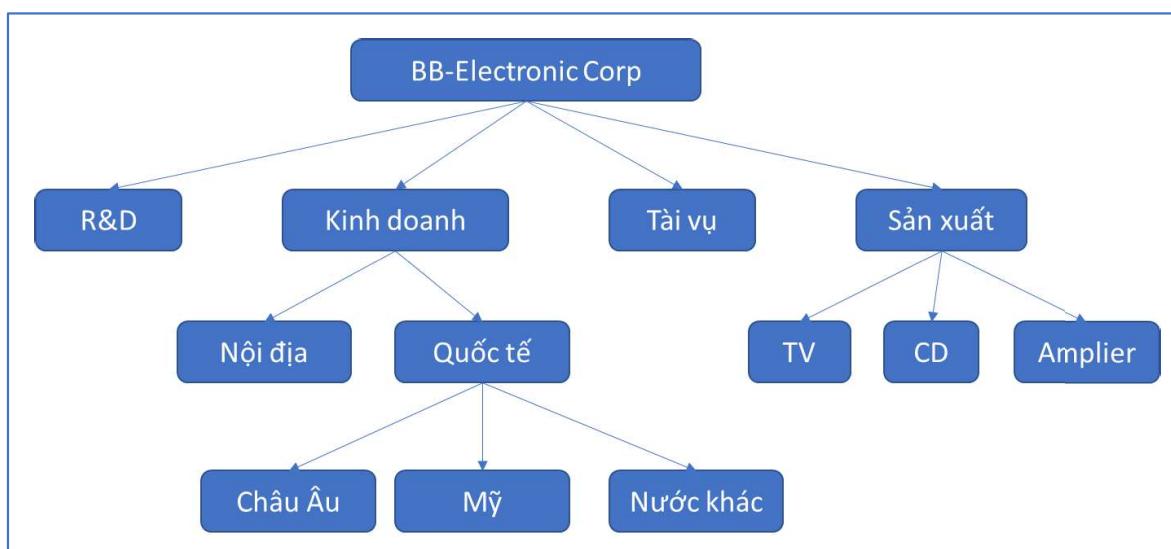
- Là số nhánh (cạnh) cần đi qua kể từ nút gốc (root) đến nút x.
- Một nút tại mức l có độ dài đường đi là i.

- **Ví dụ 7.8:** Trong **Hình 7.2**:

- Nút 13 có độ dài đường đi là 2.
- Nút 37 có độ dài đường đi là 3.

(?) *Câu hỏi: Xác định độ dài đường đi của nút F, I, J, K của cây trong **Hình 7.1**.*

7.1.3. Ví dụ về đối tượng có cấu trúc dạng cây



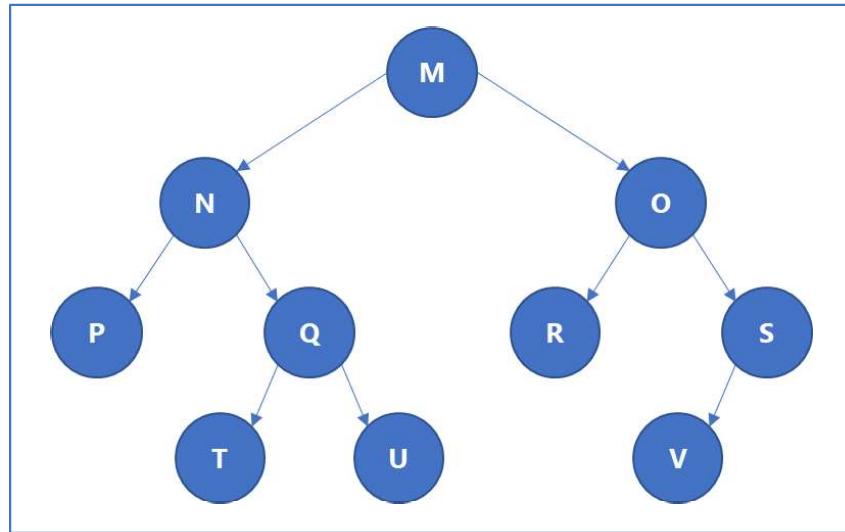
Hình 7.3. Mô hình một công ty được tổ chức dưới dạng cây.

7.2. Cây nhị phân

7.2.1. Khái niệm cây nhị phân

Cây nhị phân là cây mà mỗi nút có tối đa 2 cây con (hoặc là cây rỗng).

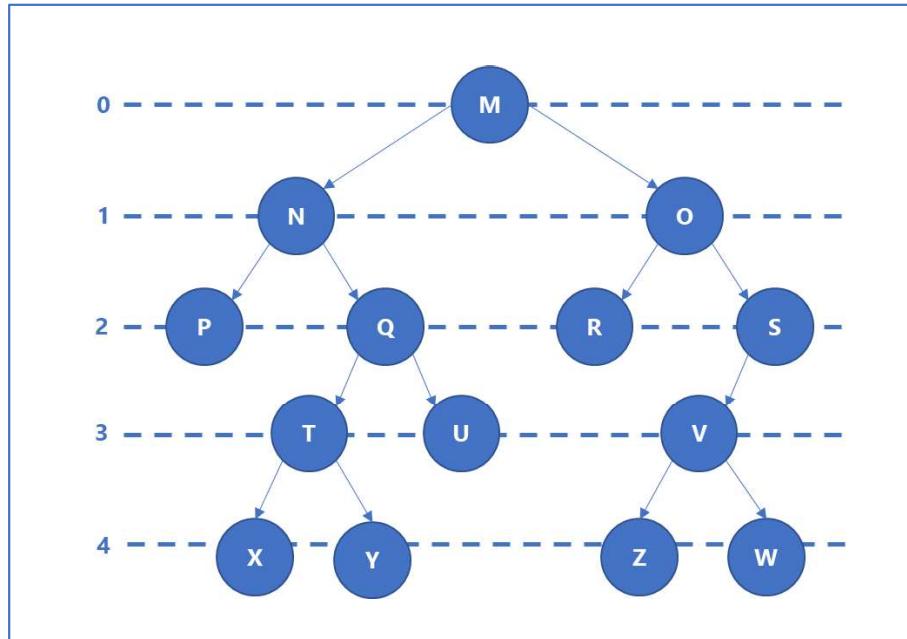
- **Ví dụ 7.9:**



Hình 7.4. Ví dụ cây nhị phân

7.2.2. Tính chất cây nhị phân

- Số nút nằm ở mức $i \leq 2^i$.
- Số nút lá $\leq 2^{(h-1)}$, với h là chiều cao của cây.
- Chiều cao của cây $h \geq \log_2(N)$ (N là số nút trong cây)
- Số nút trong cây $\leq 2^{(h-1)}$.

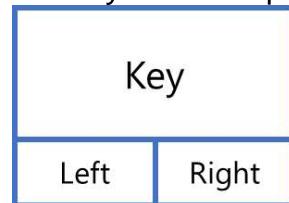


Hình 7.5. Cây nhị phân với minh họa các mức

7.2.3. Biểu diễn cây nhị phân

- Để biểu diễn cây nhị phân, ta dùng cách cấp phát các vùng nhớ liên kết với nhau bằng con trỏ tương tự như biểu diễn danh sách liên kết.

- Cấu trúc 1 nút gồm:
 - Trường chứa dữ liệu của nút.
 - Con trỏ lưu địa chỉ nút gốc của cây con bên trái.
 - Con trỏ lưu địa chỉ nút gốc của cây con bên phải.



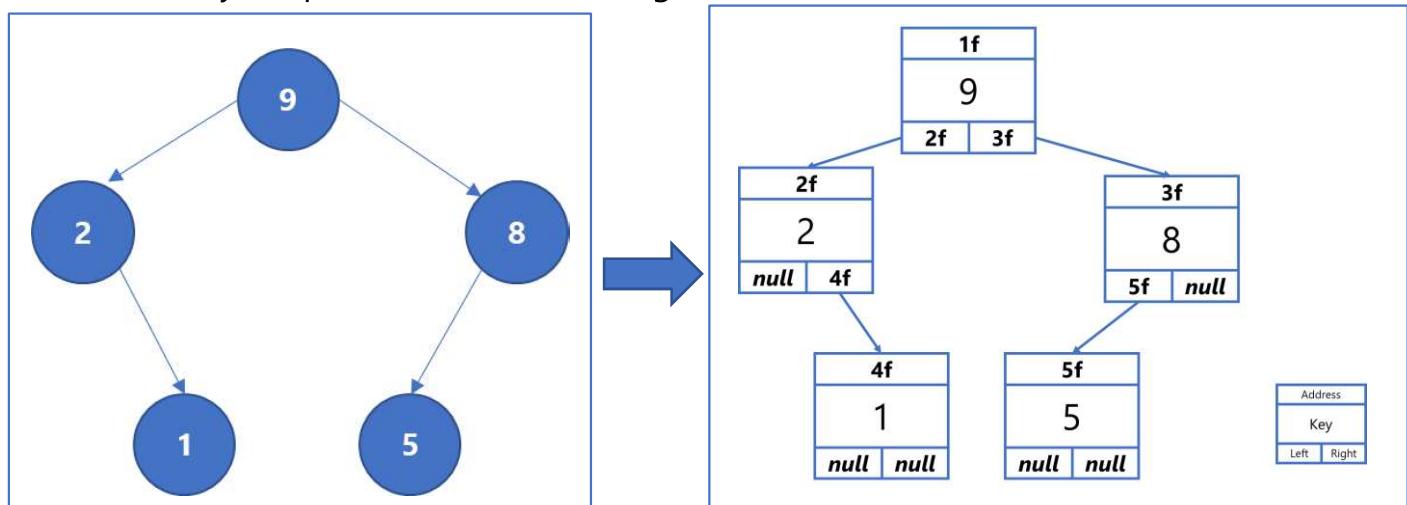
Hình 7.6. Mô hình cấu trúc 1 nút của cây nhị phân

Định nghĩa 1 nút của cây nhị phân bằng C++:

```
struct node {
    string data;
    node* left;
    node* right;
};
```

Hình 7.7. Cấu trúc một nút của cây nhị phân (lưu trữ số nguyên) được định nghĩa bằng C++.

- Cách cây nhị phân được lưu trữ trong bộ nhớ:



Hình 7.8. Mô tả cách cây nhị phân được lưu trữ trong bộ nhớ

7.2.4. Các phép duyệt cây nhị phân

- Duyệt cây là một quá trình truy cập tất cả các nút của một cây và thao tác với các giá trị của cây này (thông thường là in ra giá trị tại các nút).
- Có 3 phép duyệt chính:

- Duyệt trước: Gốc – Trái – Phải (Node – Left – Right)
- Duyệt giữa: Trái – Gốc – Phải (Left – Node – Right)
- Duyệt sau: Trái – Phải – Gốc (Left – Right – Node)

Các phép duyệt cây cho độ phức tạp $O(\log_2 h)$ với h là chiều cao của cây.

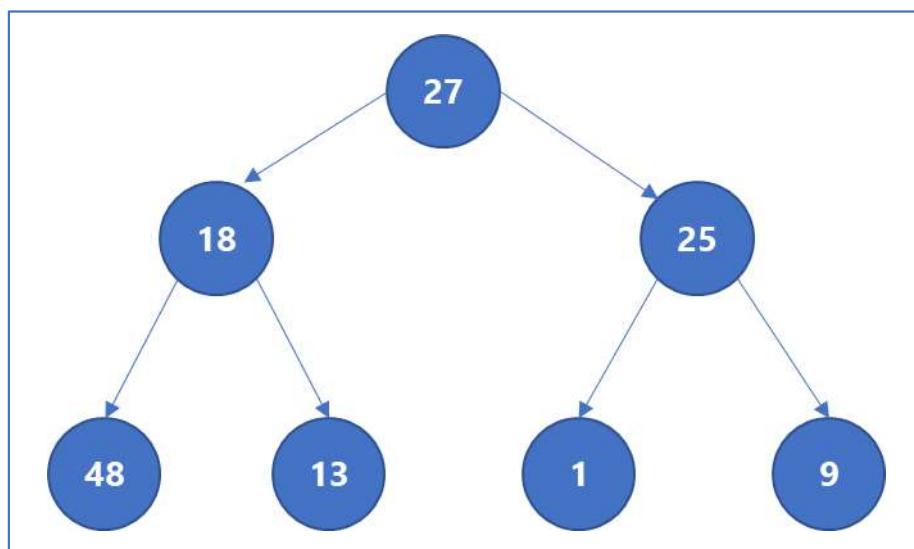
7.2.4.1. Duyệt trước (Node – Left – Right)

- Trong phép duyệt này, nút gốc sẽ được truy cập đầu tiên, sau đó sẽ tiến hành truy cập cây con bên trái và cuối cùng là cây con bên phải. Tiếp tục áp dụng thứ tự duyệt này cho các cây con, ta sẽ truy cập được toàn bộ các nút của cây.
- Mô tả thuật toán duyệt trước (Node – Left – Right) bằng C++:

```
void NLR(tree& t){
    if(t!=NULL){
        cout<<t->data<<" ";
        NLR(t->left);
        NLR(t->right);
    }
}
```

- Ví dụ 7.10:

- Ta sẽ áp dụng thuật toán trên để duyệt trước (Node – Left – Right) cây sau:



- Ta kí hiệu:

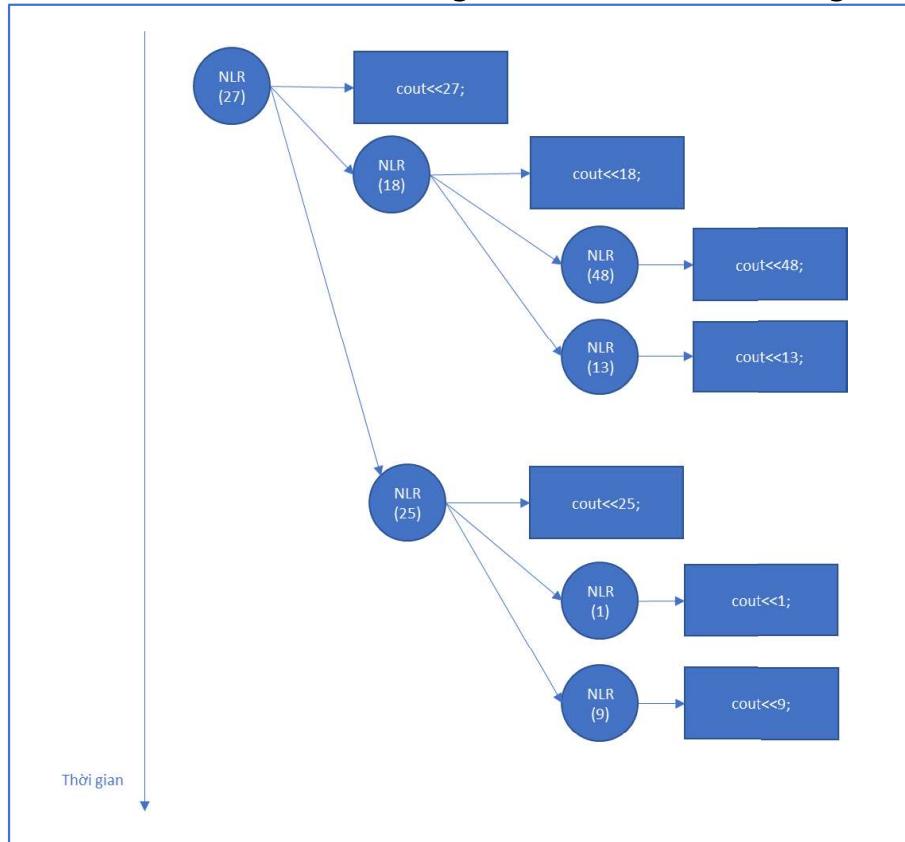


: là thao tác truy cập vào nút có giá trị khóa là 27



: là thao tác in giá trị của nút có giá trị khóa là 27 ra màn hình

- Ta có thể minh họa thuật toán bằng sơ đồ theo thứ tự thời gian như sau:



- Giải thích:

- Đầu tiên, nút 27 sẽ được truy cập, vì đây là phép duyệt trước (Node – Left – Right) nên khóa 27 sẽ được in ra trước.
- Tiếp đó, con trỏ T sẽ tiến hành truy cập cây con bên trái và cây con bên phải.
- Trước tiên, con trỏ T sẽ truy cập vào cây con bên trái, tức là nút 18 và tiến hành in giá trị khóa 18 ra.
- Theo như thuật toán được mô tả bằng C++, thì dòng lệnh `NLR(t->right)` sẽ được thực thi ngay sau dòng lệnh `NLR(t->left)`, cụ thể hơn `NLR(25)` sẽ được thực hiện ngay sau `NLR(18)`.
- Tuy nhiên, `NLR(25)` chính là một phép duyệt cây có nút gốc là nút 25, chứ không phải chỉ là thao tác in ra số 25, vậy nên phải tiến hành duyệt hết các cây con của nút 25, trước khi tiến hành duyệt cây con có nút gốc là nút 18.

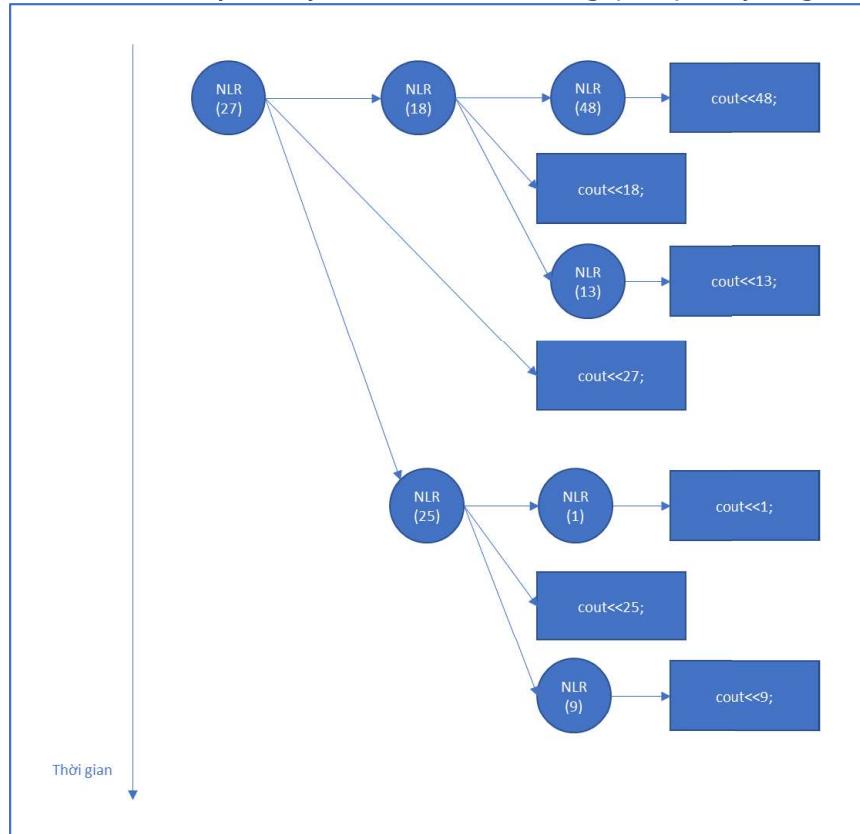
- Kết quả của phép duyệt in ra theo thứ tự thời gian, nút nào được duyệt và in ra giá trị khóa trước sẽ được hiển thị trước. Căn cứ theo sơ đồ duyệt theo thời gian như trên, ta có thứ tự duyệt các nút của cây trên theo thứ tự Node – Left – Right là: **27, 18, 48, 13, 25, 1, 9.**

7.2.4.2. Duyệt giữa (Left – Node – Right)

- Trong phép duyệt này, cây con bên trái sẽ được truy cập đầu tiên, sau đó là nút gốc, và cuối cùng là cây con bên phải. Tiếp tục áp dụng thứ tự truy cập này cho các cây con, ta sẽ truy cập được toàn bộ các nút của cây.
- Mô tả thuật toán duyệt giữa (Left – Node – Right) bằng C++:

```
void LNR(tree& t){
    if(t!=NULL){
        LNR(t->left);
        cout<<t->data<<" ";
        LNR(t->right);
    }
}
```

- **Ví dụ 7.11:** Ta sẽ tiến hành duyệt cây ở Ví dụ 7.10 bằng phép duyệt giữa (LNR).





- Giải thích:
 - Đầu tiên, nút 27 sẽ được truy cập, vì đây là phép duyệt trước (Left – Node – Right) nên cây con bên trái của nút 27 sẽ được truy cập trước thao tác in ra khóa 27.
 - Cây con bên trái của nút 27 là cây có nút gốc là nút 18, cây có nút gốc là nút 18 có cây con bên trái, nên cây con bên trái này cũng sẽ được truy cập trước khi in ra khóa 18.
 - Cây con bên trái của nút 18 là cây có nút gốc là nút 48. Nút 48 này không có cây con trái nên sẽ không có phép duyệt cây con nào của nút 48, vậy nên sẽ tiến hành **in ra khóa 48**.
 - Sau khi in khóa 48, thì nút cha của nút 48 tức là **nút 18 sẽ được in ra**, trước khi truy cập cây con bên phải.
 - Sau khi đã in ra nút 18, thì cây con bên phải của nút 18 sẽ được truy cập. Cây con này có nút gốc là nút 13 và không có cây con, nên **nút 13 sẽ được in ra**.
 - Vậy là ta đã duyệt xong cây con bên trái của nút 27, nên **nút 27 sẽ được in ra**.
 - Tương tự, ta tiến hành duyệt cây con bên phải của nút 27, và lần lượt các nút được in ra theo thứ tự là **1, 25, 9**
- Kết quả của phép duyệt in ra theo thứ tự thời gian, nút nào được duyệt và in ra giá trị khóa trước sẽ được hiển thị trước. Căn cứ theo sơ đồ duyệt theo thời gian như trên, ta có thứ tự duyệt các nút của cây trên theo thứ tự Left – Node – Right là: **48, 18, 13, 27, 1, 25, 9**.

7.2.4.3. Duyệt sau (Left – Right – Node)

- Ở phép duyệt này, nút gốc sẽ được duyệt cuối cùng, đầu tiên ta sẽ tiến hành duyệt cây con bên trái, tiếp theo là đến cây con bên phải và cuối cùng là duyệt nút gốc.
- Mô tả thuật toán duyệt sau (Left – Right – Node) bằng C++:

```
void LRN(tree& t){  
    if(t!=NULL){  
        LRN(t->left);  
        LRN(t->right);  
        cout<<t->data<<" ";  
    }  
}
```

(?) Câu hỏi:

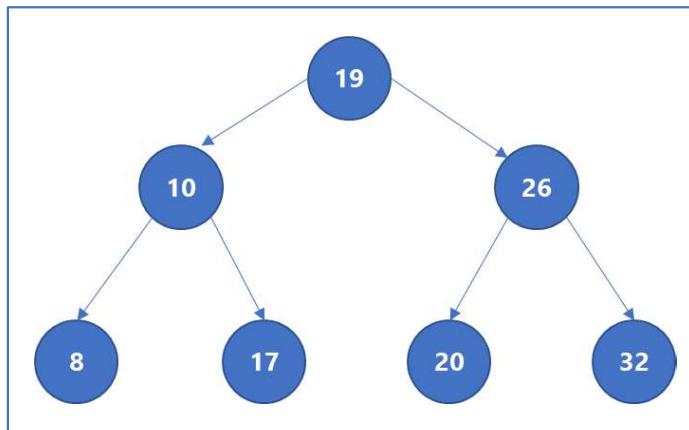
- i) Khi tiến hành duyệt cây ở Ví dụ 7.10, bạn có đoán trước được nút 27 sẽ được in ra ở vị trí nào không?
- ii) Hãy tiến hành duyệt sau cây ở Ví dụ 7.10.

7.3. Cây nhị phân tìm kiếm

7.3.1. Khái niệm

- Là cây nhị phân.
- Bảo đảm nguyên tắc bố trí khóa tại mỗi nút, sao cho:
 - Các nút trong cây con bên trái nhỏ hơn nút hiện hành.
 - Các nút trong cây con bên phải lớn hơn nút hiện hành.
- Ví kiểu dữ liệu được lưu trữ tại khóa của một nút trong cây rất đa dạng, bởi vì vậy phép so sánh “lớn hơn” và “nhỏ hơn” không đơn giản như phép so sánh $>$ (lớn hơn) hay $<$ (nhỏ hơn) của các số nguyên hay số thực. Ở trường hợp nếu muốn lưu trữ các kiểu dữ liệu phức tạp vào các nút của cây, các bạn phải tự định nghĩa phép so sánh giữa 2 giá trị khóa của 2 nút.

- Ví dụ 7.12:



Hình 7.9. Ví dụ cây nhị phân tìm kiếm

(?) Câu hỏi: Nếu thêm 1 nút có giá trị khóa bằng 30 thì nút này sẽ ở vị trí nào?

7.3.2. Các thao tác trên cây

Ta nhận thấy rằng cây con của một nút cũng là một cây nên các thao tác thực hiện với cây cha, cũng có thể thực hiện được với cây con. Do đó, một số thao tác với cấu trúc dữ liệu cây có thể thực hiện một cách dễ dàng bằng đệ quy.

7.3.2.1. Tạo một cây rỗng

```
void createTree(tree& t)
{
    t=NULL;
}
```

- Cây rỗng có giá trị địa chỉ nút gốc bằng NULL.

(?) Câu hỏi: Đây là thao tác đơn giản nhưng nếu không thực hiện thì chương trình sẽ có lỗi. Các bạn hãy giải thích tại sao?



7.3.2.2. Tạo một nút có khóa bằng x

- Việc tạo một nút của cây tương tự như tạo một nút của danh sách liên kết.
- Thao tác bao gồm:
 - Cấp phát một vùng nhớ cho 1 nút (kiểu dữ liệu của nút đã được định nghĩa như mục 4.2.3)
 - Gán giá trị cho khóa của nút cần tạo
 - Gán giá trị cho địa chỉ của nút bên trái nút cần tạo (Nếu chưa xác định địa chỉ thì gán bằng NULL)
 - Gán giá trị cho địa chỉ của nút bên phải nút cần tạo (Nếu chưa xác định địa chỉ thì gán bằng NULL)
- Biểu diễn bằng C++:

```
node* createOneNode(string x){  
    node* p = new node;  
    if(p==NULL){  
        exit(1);  
    }  
    else{  
        p->data = x;  
        p->left = NULL;  
        p->right = NULL;  
    }  
    return p;  
}
```

7.3.2.3. Tìm 1 nút có khóa bằng x trên cây

- Ý tưởng: Xuất phát từ nút gốc của cây, ta sẽ tiến hành tìm vị trí của nút có khóa là x, bằng cách so sánh giá trị của nút hiện hành với x:
 - Nếu $x <$ giá trị của nút hiện hành \Rightarrow Nút có khóa là x sẽ nằm ở cây con bên trái của nút hiện hành (nếu có). Ta sẽ tiến hành duyệt các nút ở cây con bên trái.
 - Nếu $x >$ giá trị của nút hiện hành \Rightarrow Nút có khóa là x sẽ nằm ở cây con bên phải của nút hiện hành (nếu có). Ta sẽ tiến hành duyệt các nút ở cây con bên phải.
 - Nếu $x =$ giá trị của nút hiện hành \Rightarrow Đây là vị trí cần tìm.
- Khi đã duyệt hết các vị trí có thể có (con trỏ duyệt có giá trị là null) mà vẫn không thấy nút nào có khóa là x \Rightarrow Không tồn tại nút có khóa là x trong cây.
- Hiện thực hóa ý tưởng trên bằng C++:

```
node* searchNode(tree& t, const string& x){
```

```

if(t==NULL){
    return t;
}
else{
    if(t->data==x){
        return t;
    }
    else{
        if(t->data>x){
            return searchNode(t->left,x);
        }
        else{
            return searchNode(t->right,x);
        }
    }
}
}

```

Hình 7.14. Biểu diễn thao tác tìm 1 nút có khóa là x bằng C++

7.3.2.4. Thêm một nút vào cây nhị phân tìm kiếm

- Điều kiện: Sau khi thêm nút vào cây, vẫn đảm bảo cây là cây nhị phân tìm kiếm.
- Ý tưởng: Ta cần xác định vị trí của nút cần thêm:
 - Nằm ở cây con bên phải của tất cả các nút có khóa nhỏ hơn nó.
 - Nằm ở cây con bên trái của tất cả các nút có khóa lớn hơn nó.
 - Là nút bên trái của nút không có cây con trái hoặc nút bên phải của nút không có cây con bên phải.
- Xuất phát từ nút gốc. Tiến hành duyệt cây để tìm vị trí thích hợp.
- So sánh giá trị khóa của nút cần thêm với khóa của nút hiện hành.
- Nếu nút hiện hành khác NULL:
 - Nếu khóa của nút cần thêm = khóa của nút hiện hành
=> Thông báo đã tồn tại 1 nút có giá trị cần thêm trong cây.
 - Nếu khóa của nút cần thêm < khóa của nút hiện hành
=> Tiến hành duyệt cây con bên trái của nút hiện hành để tìm vị trí thích hợp.
 - Nếu khóa của nút cần thêm > khóa của nút hiện hành
=> Tiến hành duyệt cây con bên phải của nút hiện hành để tìm vị trí thích hợp.
- Nếu nút hiện hành bằng NULL: Tiến hành thêm khóa cần thêm vào cây vào nút hiện hành.

```
void insertNode(tree& t, const string& x){
```

```
if(t!=NULL){  
    {  
        if(t->data==x){  
            return;  
        }  
        else if (t->data>x) {  
            insertNode(t->left,x);  
        }  
        else{  
            insertNode(t->right,x);  
        }  
    }  
}  
else{  
    t=new node;  
    if(t==NULL){  
        exit(1);  
    }  
    else{  
        t->data = x;  
        t->left = t->right = NULL;  
    }  
}  
}
```

7.3.2.5. Xóa 1 nút có Key bằng x trên cây

- Điều kiện: Sau khi xóa xong, cây vẫn là cây nhị phân tìm kiếm
- Ý tưởng:
 - Đầu tiên, ta cần xác định được vị trí của nút cần xóa (nút chứa khóa x).
 - Tiếp theo ta thực hiện thay thế vị trí của nút cần xóa bằng một nút khác trong cây. Có các trường hợp sau xảy ra:

TH1: Nút cần xóa là nút lá.

Ở trường hợp này, ta xóa nút lá sẽ không gây ảnh hưởng đến cấu trúc cây.

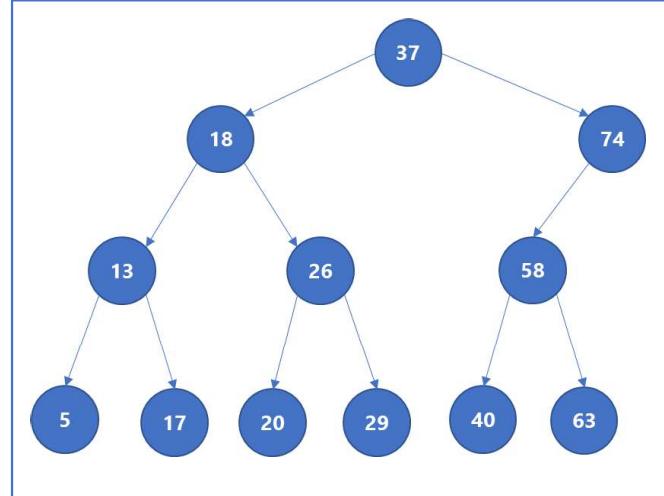
TH2: Nút cần xóa có duy nhất 1 cây con.

Ở trường hợp này, ta chỉ cần nối nút cha của X với nút con duy nhất của X và hủy nút X.

TH3: Nút cần xóa có đầy đủ 2 cây con.

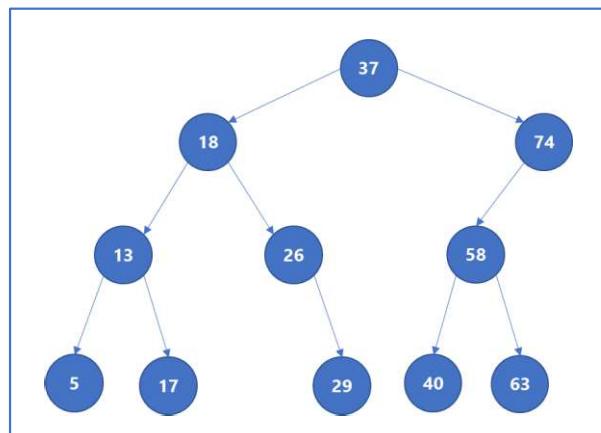
Ở trường hợp này, ta cần tìm nút thế mạng, nút này sẽ thay thế vị trí của nút cần xóa. Thông thường, ta sẽ chọn nút thế mạng là nút lớn nhất (phải nhất) của cây con bên trái, hoặc nút nhỏ nhất (trái nhất) của cây con bên phải.

- **Ví dụ 7.13:** Xét cây nhị phân tìm kiếm dưới đây, ta sẽ tiến hành xóa 3 nút của cây thuộc 3 trường hợp xóa đã nêu ở bên trên.



Hình 7.16. Minh họa thao tác xóa nút cây nhị phân tìm kiếm

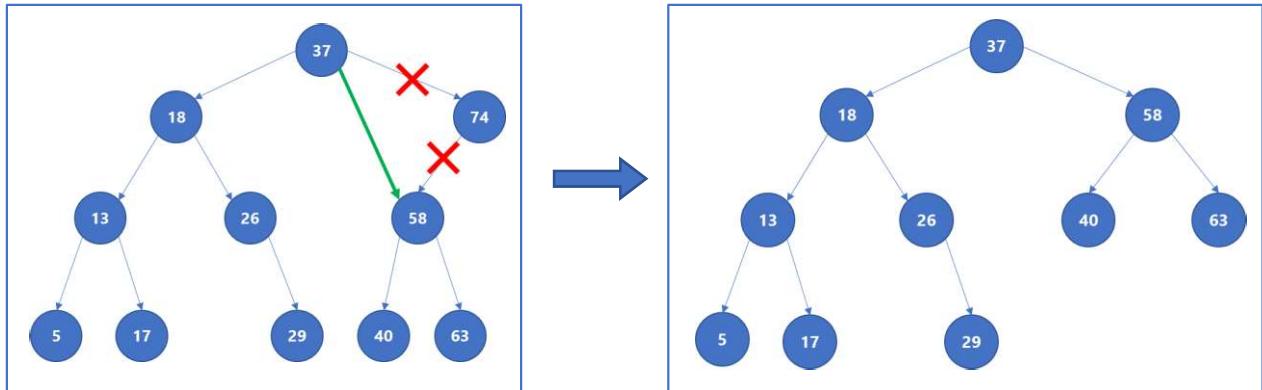
TH1: Xóa nút lá. Ta sẽ tiến hành xóa 1 nút lá bất kỳ, ở đây ta sẽ chọn xóa nút 20. Vì xóa nút lá sẽ không ảnh hưởng đến cấu trúc cây, nên việc xóa nút lá rất đơn giản, ta chỉ cần hủy nút đó đi. Cây sau khi xóa nút 20.



Hình 7.17. Minh họa thao tác xóa nút cây nhị phân tìm kiếm

TH2: Xóa nút có duy nhất 1 cây con. Ví dụ, ta cần xóa nút 74.

Ở trường hợp này, ta sẽ mốc nối nút con của nút cần xóa với nút cha của nút cần xóa. Ở đây nút cần xóa là nút 74. Nút cha của nút cần xóa là nút 37, nút con của nút cần xóa là nút 58.

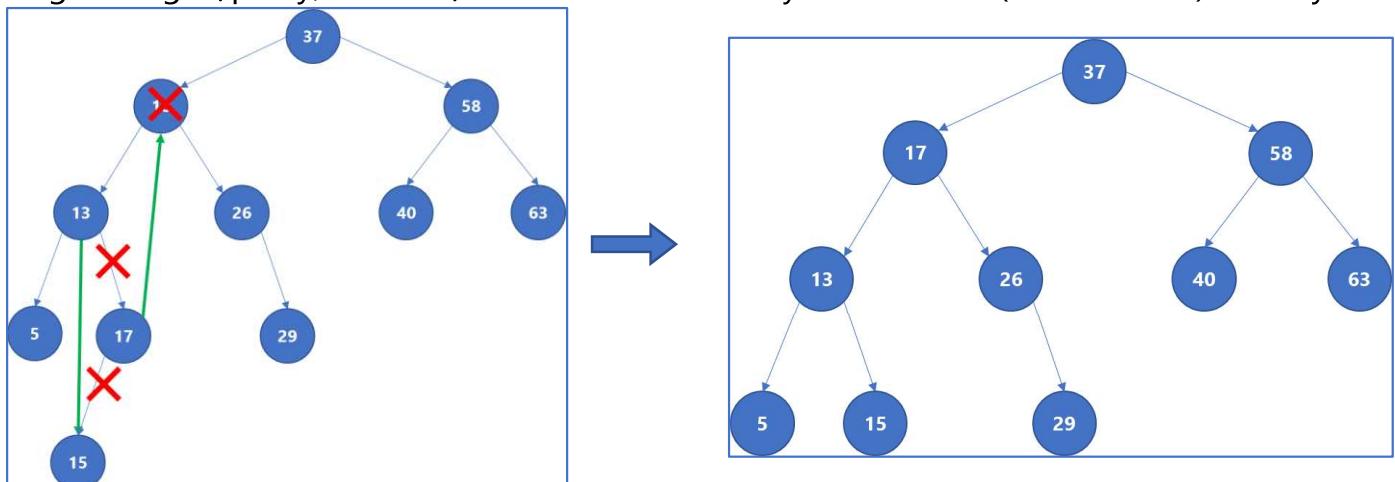


Hình 7.18. Minh họa thao tác xóa nút cây nhị phân tìm kiếm

TH3: Xóa nút có đầy đủ 2 cây con. Ví dụ ở đây ta sẽ xóa nút 18. Ta cần chọn nút thay thế, ở đây có 2 cách chọn:

1. Chọn nút thay thế là nút nhỏ nhất của cây con bên phải (tức là nút 26)
2. Chọn nút thay thế là nút lớn nhất của cây con bên trái (tức là nút 17)

Trong trường hợp này, ta sẽ chọn nút lớn nhất của cây con bên trái (tức là nút 17) để thay thế.



Hình 7.19. Minh họa thao tác xóa nút cây nhị phân tìm kiếm

- Minh họa ý tưởng xóa một nút khóa x bằng C++:
 - Hàm tìm nút thay thế:

```
void replaceNode(tree& p, tree& t){
    if(t->right != NULL){
        replaceNode(p,t->right);
        // Tìm nút phải nhất cây con trái
    }
    else{
        cout<<"Nut thay the: "<<t->data<<endl;
        p->data = t->data;
```

```

        // Nút p là nút cần xóa (ta đã gán địa chỉ của t cho p ở hàm
deleteNodeX),
        // ta tiến hành gán lại data của t (nút thay thế) cho p (nút cần
xóa)
        // bây giờ ta chỉ cần xóa đi vùng nhớ của nút thay thế
        p=t;
        // Ta gán địa chỉ p bằng vị trí của nút thay thế
        t=t->left;
        // Nếu nút thay thế có cây con bên trái,
        // nút trái của nút thay thế sẽ trở thành nút phải của nút cha nút
thay thế
    }
}

```

- Hàm xóa nút khóa x:

```

void deleteNodeX(tree& t, const string& x){
    if(t!=NULL){
        if(t->data<x){
            deleteNodeX(t->right,x);
        }
        else if(t->data>x){
            deleteNodeX(t->left,x);
        }
        else{
            node* p = t;
            // ta gán địa chỉ của t cho p
            // để sau khi tìm được nút thay thế cho t,
            // ta tiến hành cập nhật data của nút thay thế cho vị trí này
            if(t->left == NULL){
                t=t->right;
            }
            else if (t->right == NULL){
                t=t->left;
            }
            else {
                replaceNode(p,t->left);
            }
            delete p;
            //p bây giờ lưu địa chỉ của nút thay thế - không còn cần sử dụng
        nữa
    }
}

```

}

7.4. B-Tree

7.4.1. Định nghĩa

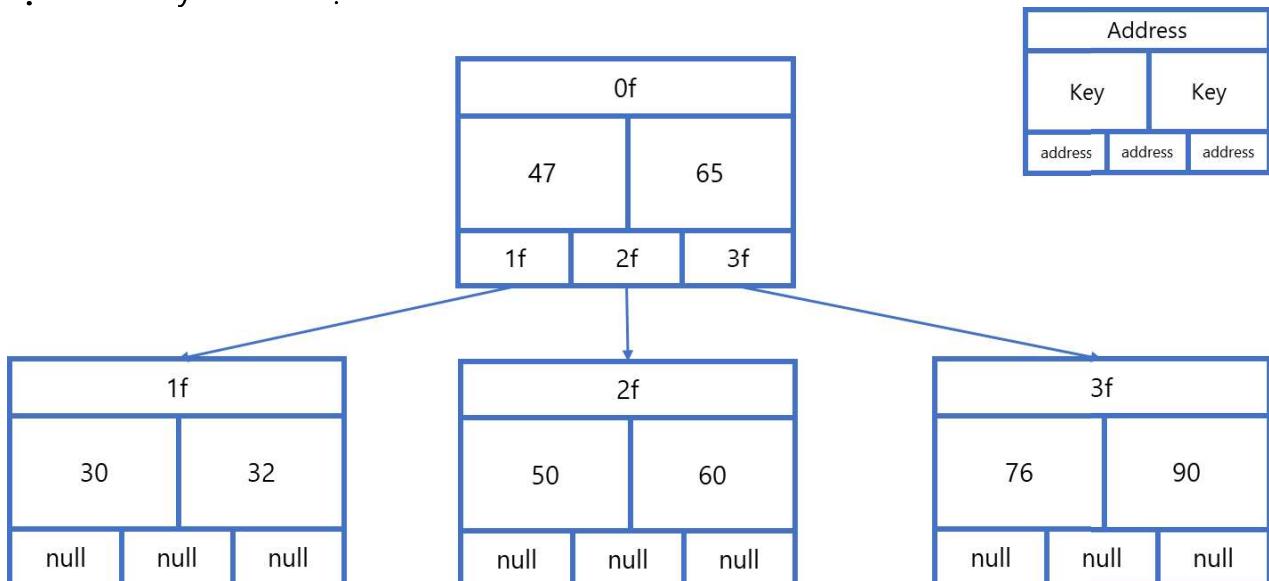
7.4.1.1. Một nút:

- Gồm: khóa và các cây con.
- Số cây con = số khóa + 1.



Hình 7.20. Minh họa 1 nút của cây B-tree với 2 khóa và 3 cây con

- **Ví dụ 7.14:** Cây B-tree bậc 3:



Hình 7.21. Minh họa cấu trúc cây B-tree

7.4.1.2. Các tính chất:

Cho số tự nhiên $k > 0$, B-Trees bậc m (với $m \geq k$) là một cây nhiều nhánh tìm kiếm thỏa mãn các tính chất :

i/ Tất cả các node lá ở cùng một mức.

ii/ Tất cả các node trung gian trừ node gốc có tối đa m cây con ($m-1$ khóa) và tối thiểu $m/2 + 1$ cây con khác rỗng ($m/2$ khóa).

iii/ Mỗi node hoặc là node lá hoặc node có k khóa thì có $k+1$ cây con và phân chia các khóa trong các nhánh con theo cách của cây tìm kiếm.

iv/ Node gốc có nhiều nhất m cây con hoặc ít nhất có 2 cây con khi node gốc không là node lá hoặc không có cây con nào khi cây chỉ có 1 node gốc.

7.4.1.3. Tips :

Cây B-tree bậc m :

- Mỗi nút sẽ chứa khóa và các cây con của nó, số cây con bằng số khóa + 1.
- Nút gốc:
 - Hoặc là NULL
 - Không có cây con nào nếu là nút lá
 - Có ít nhất 2 cây con (nút gốc chứa 1 khóa) nếu không là nút lá
 - Có nhiều nhất m cây con (nút gốc chứa $m-1$ khóa)
- Nút trung gian:
 - Có ít nhất $m/2$ cây con (nếu m chẵn) và $(m+1)/2$ cây con (nếu m lẻ)
 - Có nhiều nhất m cây con
- Nút lá:
 - Tất cả các nút lá sẽ nằm cùng một mức
- Khóa:
 - Các khóa nằm trên các nút sẽ được sắp xếp theo thứ tự nhất định (thường sẽ theo thứ tự tương tự cây nhị phân tìm kiếm)

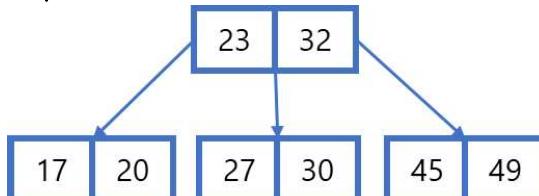
7.4.2. Các thao tác với cây B-tree:

7.4.2.1. Các khái niệm:

*Thao tác tách node (split):

- Là thao tác sẽ được thực hiện khi thêm 1 khóa vào nút mà số khóa của nút sẽ vượt quá số khóa tối đa.
- Tiến hành lấy nút chính giữa đem gộp với nút cha của nút đang xét trở thành nút cha của 2 nút mới, 2 nút mới này chính là nút chứa các khóa bên trái và nút chứa các khóa bên phải của nút chính giữa.
- Nếu nút cha vừa được thêm khóa cũng vượt quá số khóa tối đa, ta sẽ tiếp tục thực hiện thao tác split (tách) cho nút đó.

- **Ví dụ 7.15:** Cho cây B-Tree bậc 3 như sau:



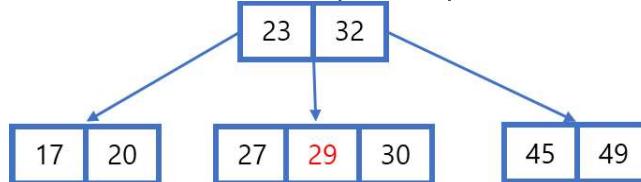
Đầu tiên ta xác định số cây con, số khóa tối đa, tối thiểu.

- Số cây con tối đa: 3
- Số cây con tối thiểu: $(3+1)/2 = 2$

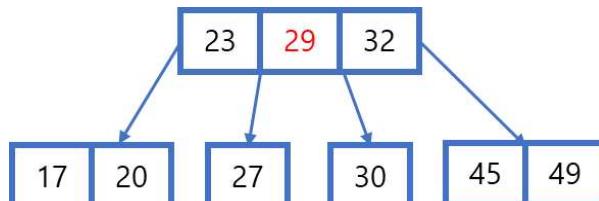
- Số khóa tối đa: $3 - 1 = 2$
- Số khóa tối thiểu $(3-1)/2 = 1$

Hãy tiến hành thêm vào cây khóa có giá trị 29 sao cho cây vẫn là cây B-Tree bậc 3.

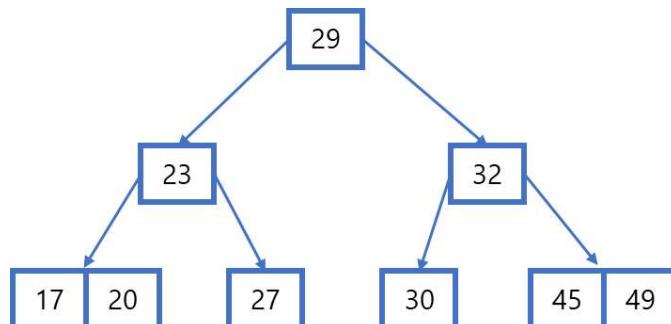
Nhìn vào cây B-Tree, thì ta xác định được vị trí phù hợp với số 29 chính là giữa nút 27, 30.



Tuy nhiên, khi ta thêm khóa 29 vào nút này thì số khóa sẽ vượt quá số khóa tối đa là 3. Vậy nên ta sẽ tiến hành thao tác split node (tách). Khóa 29 sẽ gộp với nút cha của nút đang xét, khóa 27, 30 sẽ tách thành 2 nút mới là con của nút chứa khóa 29.



Tuy nhiên, khi khóa 29 nằm ở nút 23, 32 thì số khóa của nút này cũng vượt quá số khóa tối đa, nên sẽ tiếp tục thực hiện thao tác split (tách). Cây B-Tree sau khi thực hiện xong thêm khóa 29.

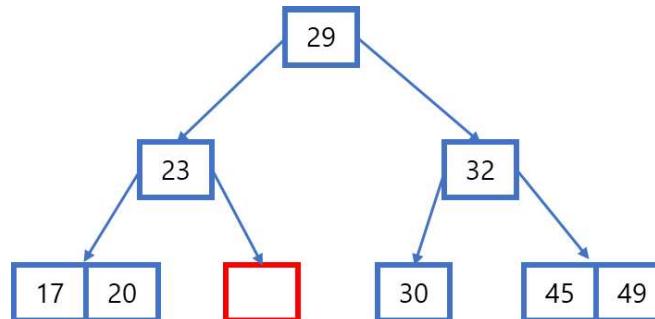


Hình 7.22. Cây B-Tree bậc 3 sau khi hoàn tất thêm khóa 29.

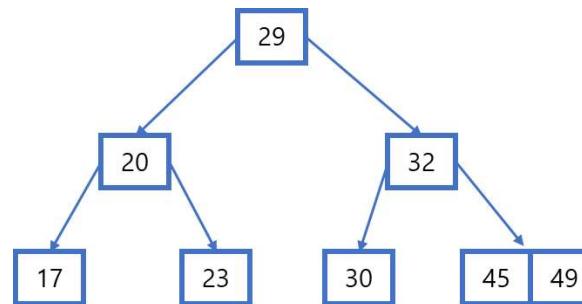
*Thao tác nhường khóa (underflow)

- Nếu nút sau khi xóa bớt khóa, có số khóa nhỏ hơn số khóa tối thiểu trong 1 nút, thì ta sẽ xét các nút kề của nút này.
- Nếu có nút kề có số khóa lớn hơn số khóa tối thiểu trong 1 nút, ta sẽ tiến hành nhường khóa của nút kề này cho nút đang thiếu khóa.
- Khóa được nhường sẽ lên thế chỗ 1 khóa của nút cha, khóa của nút cha sẽ vào vị trí của nút bị thiếu khóa.

- **Ví dụ 7.16:** Xét cây ở Hình 7.22. Hãy tiến hành xóa nút 27 khỏi cây.



- Khi xóa nút lá 27 khỏi cây, ta thấy nút hiện hành có 0 khóa, ít hơn số khóa tối thiểu.
- Ta tiến hành xét nút kề của nút 27 là nút (17, 20). Nút này có số khóa nhiều hơn số khóa tối thiểu, nên sẽ tiến hành thao tác underflow. Khóa 23 sẽ xuống thay thế cho khóa 27. Khóa 20 sẽ lên thay thế khóa 23.



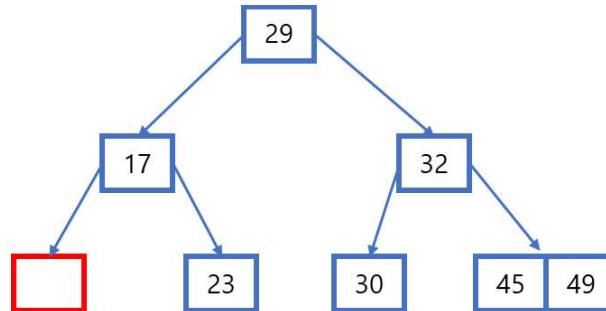
Hình 7.23. Cây B-Tree sau khi xóa khóa 27

*Thao tác hợp (catenate)

- Trong trường hợp không có nút kề nào có số khóa lớn hơn số khóa tối thiểu, ta sẽ tiến hành thao tác **catenate**.
- Thao tác **catenate** là thao tác, gom các khóa của nút đang bị thiếu khóa, với nút cha và nút anh chị của nó (nút có chung khóa cha) trở thành 1 nút. Khi này, nút cha của nút bị thiếu khóa sẽ mất đi một nút, và các khóa và nút được gom sẽ trở thành nút con mới của nút cha này.
- Ở nút cha có 1 khóa bị mất, 2 cây con bị mất, và 1 cây con mới được tạo. Vậy nút cha mất đi 1 khóa và 1 cây con. Nghĩa là số cây con vẫn đảm bảo hơn số khóa 1. Cấu trúc cây vẫn được đảm bảo.
- Trong trường hợp, sau khi **catenate**, nút cha của nút vừa bị xóa khóa không đảm bảo số khóa tối thiểu => Xem xét để thực hiện thao tác **underflow**, nếu không thực hiện được thao tác **underflow** thì ta sẽ thực hiện thao tác **catenate** cho nút này. Cứ thế đến khi nào không có nút nào bị thiếu số khóa tối thiểu, hoặc nút vừa gom trở thành nút gốc thì dừng việc **catenate**.

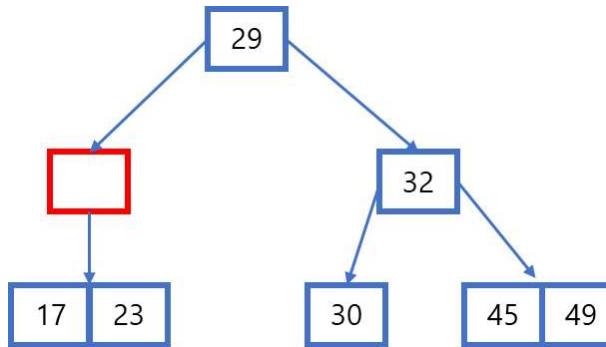
- **Ví dụ 7.17:** Xét cây ở **Hình 7.23** Hãy xóa khóa 20 khỏi cây.

Khi xóa khóa 20, là nút trung gian. Ta chọn 17 là khóa thế mạng của nó. Việc xóa khóa 20 quy về xóa khóa của nút lá vị trí nút khóa 17.



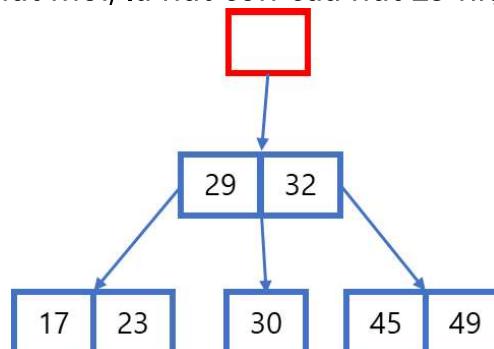
Ở vị trí nút lá vừa bị xóa khóa để làm khóa thế mạng, số khóa của nút hiện tại là 0, nhỏ hơn số khóa tối thiểu. Tuy nhiên, nút kề duy nhất của nút này là nút chứa khóa 23, chỉ có 1 khóa. Nên không thể thực hiện **underflow**. => Ta thực hiện thao tác **catenate**.

Ta tiến hành gom tất cả khóa nút kề của nút đang thiếu khóa và nút cha thành một nút mới. Tức là ta gom 17, 23 thành 1 nút mới. Và làm con của nút cha hiện tại là nút chứa khóa 17. Nút 17 mất đi 1 khóa.

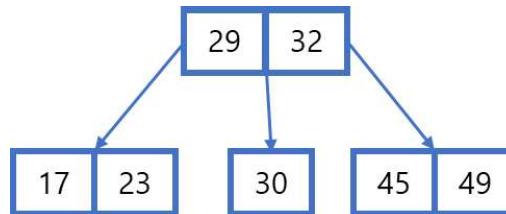


Tuy nhiên nút cha này, tức là nút chứa khóa 17 trước đó, lại không có khóa nào, và nút kề của nó là 32 cũng chỉ có 1 khóa, không nhiều hơn số khóa tối thiểu, nên không thể thực hiện thao tác **underflow**. Ta tiếp tục tiến hành **catenate** cho nút này.

Gom 29, 32 tạo thành nút mới, là nút con của nút 29 hiện tại.



Tuy nhiên nút cha lúc này không có khóa nào, tuy nhiên vì đây là nút gốc, nên chúng ta không cần thực hiện thao tác **underflow** hay **catenate**, mà chỉ cần cho nút (29, 32) trở thành nút gốc mới. Cây sau khi hoàn tất xóa nút 20:



7.4.2.2. Xác định số cây con, số khóa tối đa, tối thiểu:

Các bước tạo cây B-tree với bậc m:

- Số cây con tối đa: m
- Số cây con tối thiểu (cho nút trung gian): $m/2$ nếu m chẵn, $(m+1)/2$ nếu m lẻ
- Số khóa tối đa: $m-1$
- Số khóa tối thiểu: số cây con tối thiểu – 1.

- **Ví dụ 7.15:** Xác định số cây con tối đa, tối thiểu (cho nút trung gian), số khóa tối đa, số khóa tối thiểu (trừ nút gốc) của cây B-tree bậc 5.

- Số cây con tối đa: 5
- Số cây con tối thiểu (cho nút trung gian): 3
- Số khóa tối đa: 4
- Số khóa tối thiểu (trừ nút gốc): 2

7.4.2.3. Thêm nút vào cây B-tree:

Thuật toán thêm khóa:

- Xác định vị trí của khóa cần thêm
- Thêm khóa vào nút
- Nếu số khóa vượt quá số khóa tối đa, thực hiện thao tác split (tách)
- Nếu khóa được đưa lên làm khóa cha, nằm ở nút có số khóa vượt quá số khóa tối đa khi thêm nút vừa tách, ta tiếp tục thực hiện thao tác split (tách) cho nút này.

Ví dụ 7.16: Hãy vẽ cây B-tree bậc 5, thêm vào cây các khóa theo thứ tự sau: **3, 7, 9, 23, 45, 1, 5, 14, 25, 24, 13, 11, 8, 19, 4, 31, 35, 56**

Xác định số cây con, số khóa tối đa, tối thiểu:

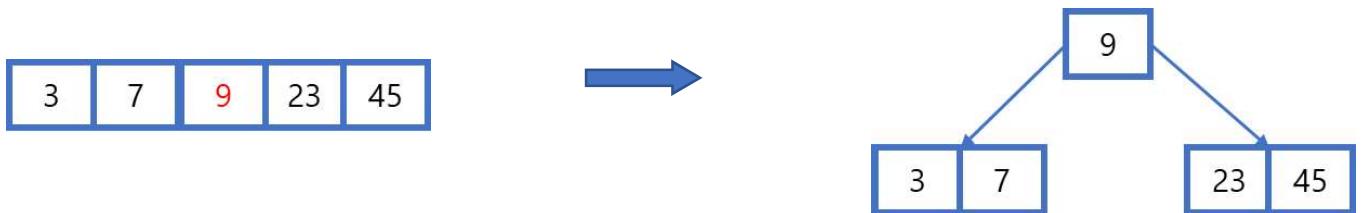
- Số cây con tối đa: 5
- Số cây con tối thiểu: $(5+1)/2 = 3$
- Số khóa tối đa: $5 - 1 = 4$
- Số khóa tối thiểu: $3 - 1 = 2$

Thêm các khóa 3, 7, 9, 23:

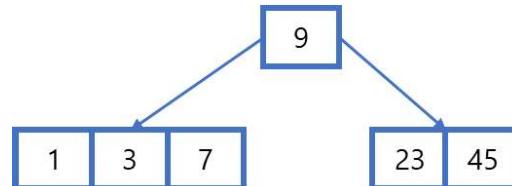
- Vì số khóa tối đa của 1 nút là 4, nên 4 khóa đầu tiên đều nằm ở nút gốc. Cây B-tree sau khi thêm 4 khóa 3, 7, 9, 23



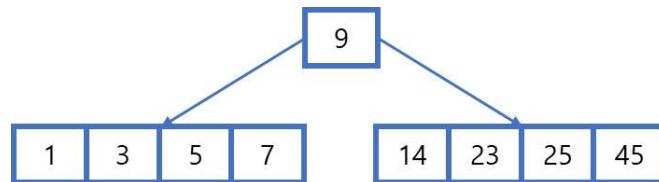
Thêm khóa 45, khi thêm khóa 45 vào nút hiện hành thì số nút sẽ vượt quá số khóa tối đa của 1 nút, vậy nên thao tác Split (tách) sẽ được thực hiện:



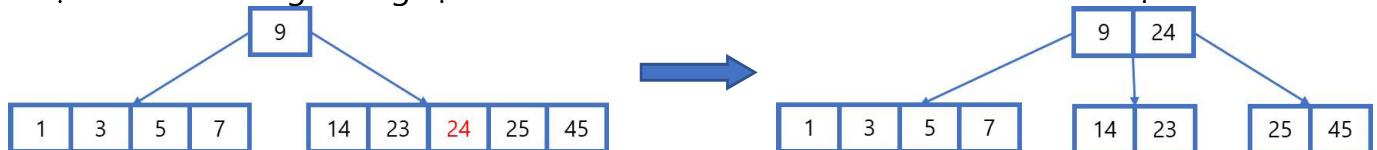
Thêm khóa 1, vì số khóa hiện tại của nút chứa 3, 7 < số khóa tối đa. Nên có thể thêm khóa 1 vào nút đó.



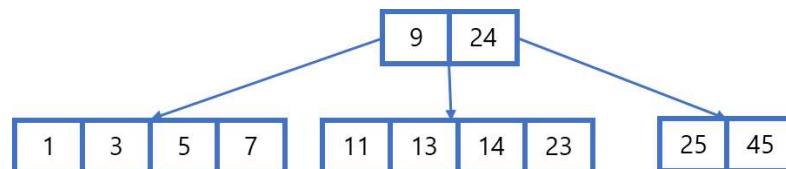
Tương tự với khóa 5, 14, 25:



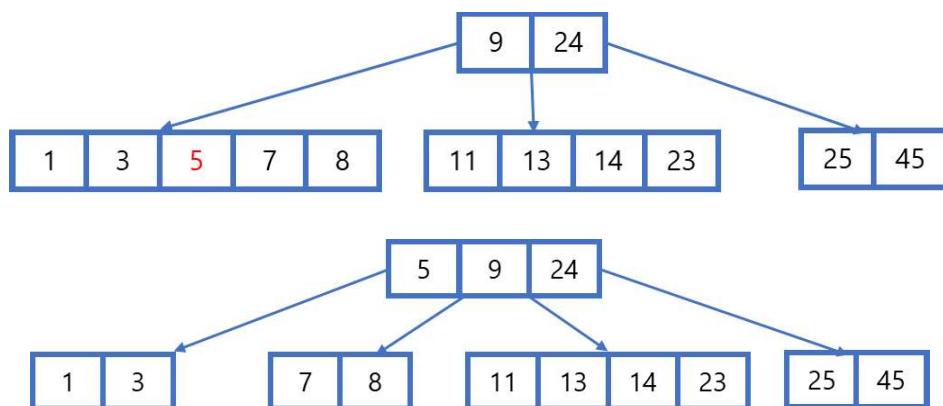
Thêm khóa 24: tương tự như khóa 45, số khóa ở nút 14, 23, 25, 45 đã đạt tối đa nên thao tác split sẽ được thực hiện. Lúc này (14, 23) và (25, 45) sẽ được tách thành nút mới, khóa 24 sẽ được tách lên đứng chung vị trí với nút 9 để làm nút cha của 2 nút vừa được tách.



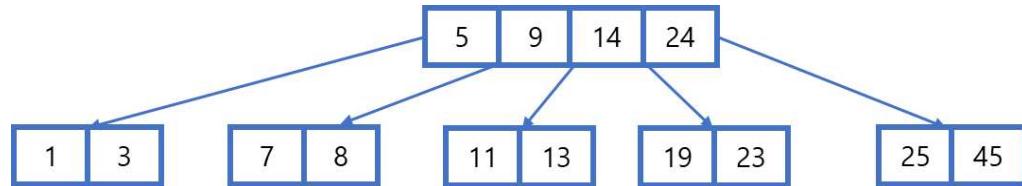
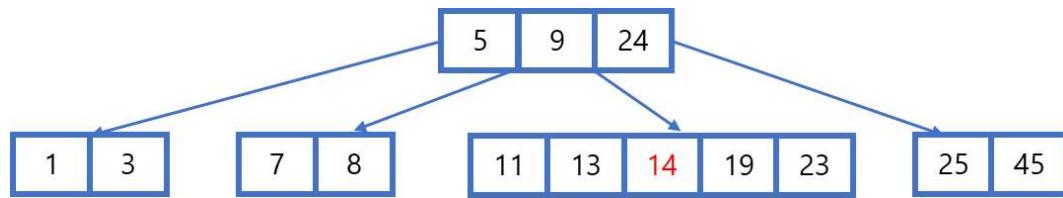
Thêm khóa 13, 11:



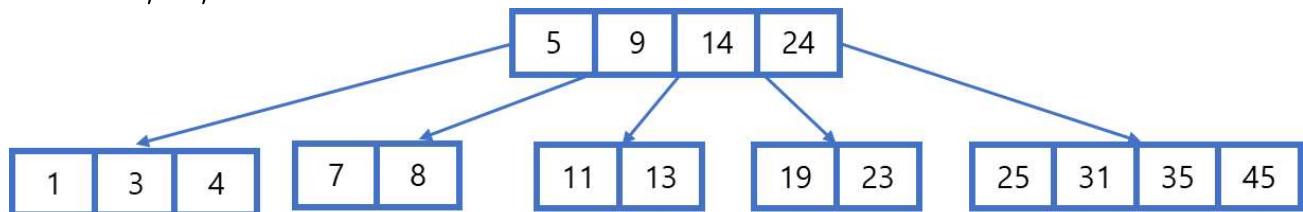
Thêm khóa 8:



Thêm khóa 19:

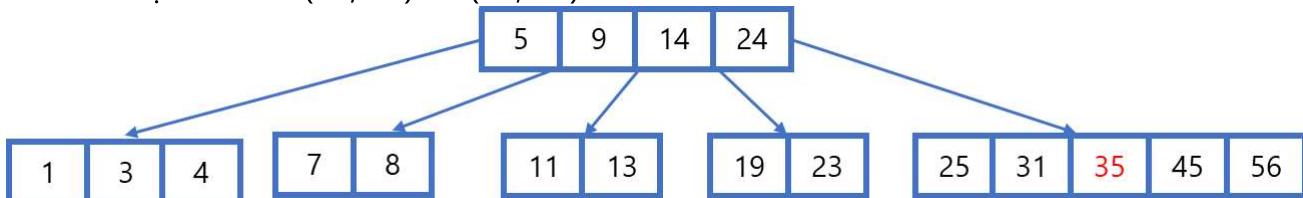


Thêm khóa 4, 31, 35:

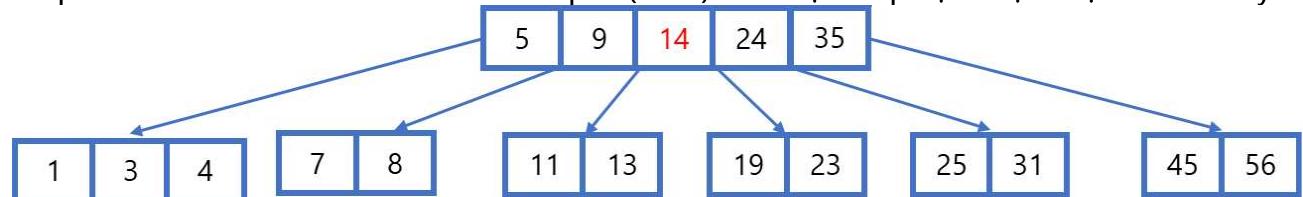


Thêm khóa 56:

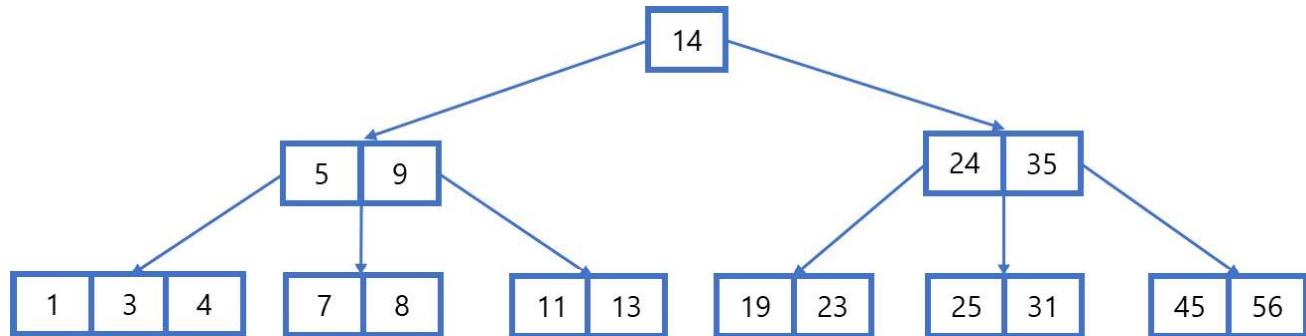
- Khóa 56 sau khi thêm vào nút (25, 31, 35, 45) thì tổng số khóa của nút này sẽ vượt quá số khóa tối đa trên 1 nút. Vậy nên nút chính giữa là 35 sẽ được tách lên làm cha của 2 nút được tách là (25, 31) và (45, 56)



- Khi khóa 35 được tách lên nằm trong nút 5, 9, 14, 24 thì số khóa của nút này lại vượt quá số khóa tối đa. Nên thao tác split (tách) sẽ được tiếp tục thực hiện ở nút này.



- Khóa chính giữa là 14 sẽ được tách lên làm cha của 2 nút được tách ra là (5, 19) và (24, 35)



Hình 7.22. Cấu trúc cây B-tree sau khi thêm các nút

7.4.2.4. Xóa nút trên cây B-Tree

*Thuật toán xóa khóa:

- Xóa khóa tại mọi nút được quy đổi về xóa khóa trong một nút lá. Vì khi xóa 1 nút trung gian, ta sẽ tìm nút thế mạng tương tự như khi ta xóa nút trung gian trong cây nhị phân tìm kiếm (Sẽ là nút trái nhất các cây con bên phải, hoặc nút phải nhất trong các cây con trái). Lúc này thì số lượng các khóa tại nút trung gian không bị thay đổi, và nút lá thì bị mất một khóa, nên ta có thể xem như việc xóa 1 khóa của nút lá.
- Đặt $k = (m-1)/2$ nếu m lẻ, và $k = m/2 - 1$ nếu m chẵn (Chương trình UIT chỉ xét cây B-tree bậc lẻ).
- Nếu nút lá chứa khóa cần xóa có nhiều hơn k khóa, thực hiện xóa khóa đó mà không sợ ảnh hưởng đến cấu trúc cây.
- Nếu nút lá chỉ chứa k khóa, sau khi xóa chỉ còn $k-1$ khóa, ta thực hiện thao tác **underflow**:

i) Nếu một nút kề nút vừa mất khóa có nhiều hơn k khóa, chuyển khóa đó sang cho nút vừa xóa đi 1 khóa.

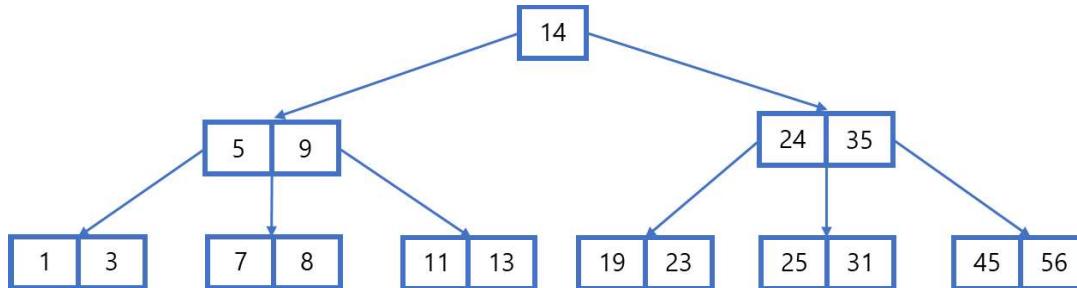
ii) Ngược lại, thực hiện **catenate** với một nút kề.

Lặp lại thao tác **underflow** như trên cho nút trung gian nếu nút trung gian có ít hơn k khóa.

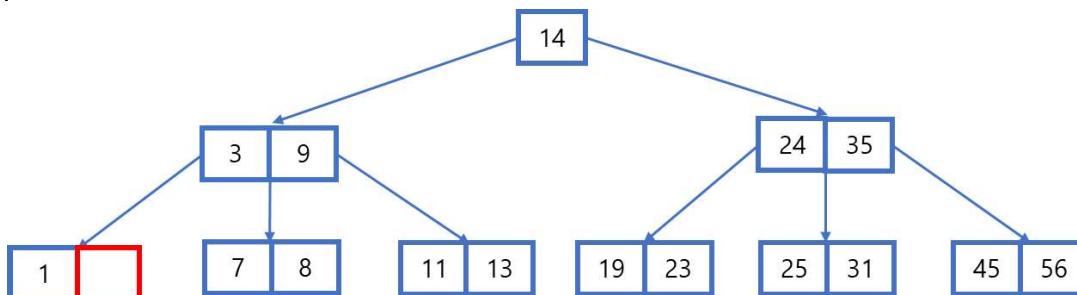
- Nếu chúng ta thực hiện **catenate** đến nút gốc và nút gốc chỉ còn lại 1 nút con, thì cho nút con làm nút gốc mới.

- **Ví dụ 7.17:** Hãy xóa lần lượt các khóa: 4, 5, 7, 3, 14 khỏi cây B-tree trong Hình 7.22.

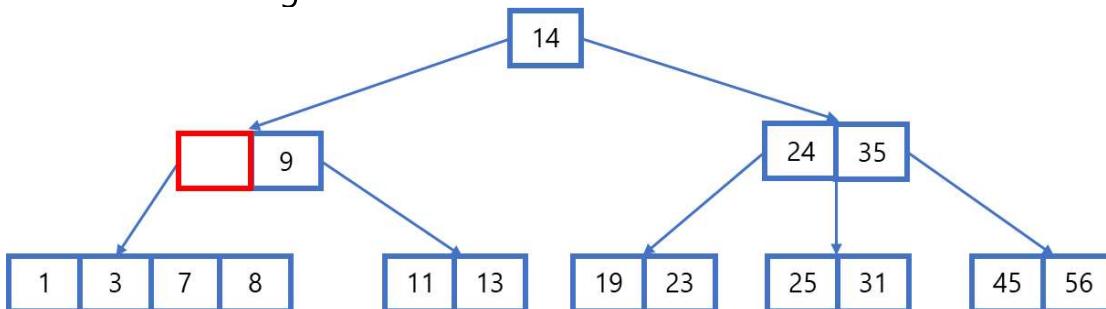
- Đầu tiên, ta xác định $k = (5-1)/2 = 2$
- Nút hiện tại chứa khóa 4 là nút lá, có 3 khóa $> k$. Nên ta chỉ cần xóa khóa 4 mà không sợ ảnh hưởng đến cấu trúc cây. Cây sau khi xóa khóa 4:



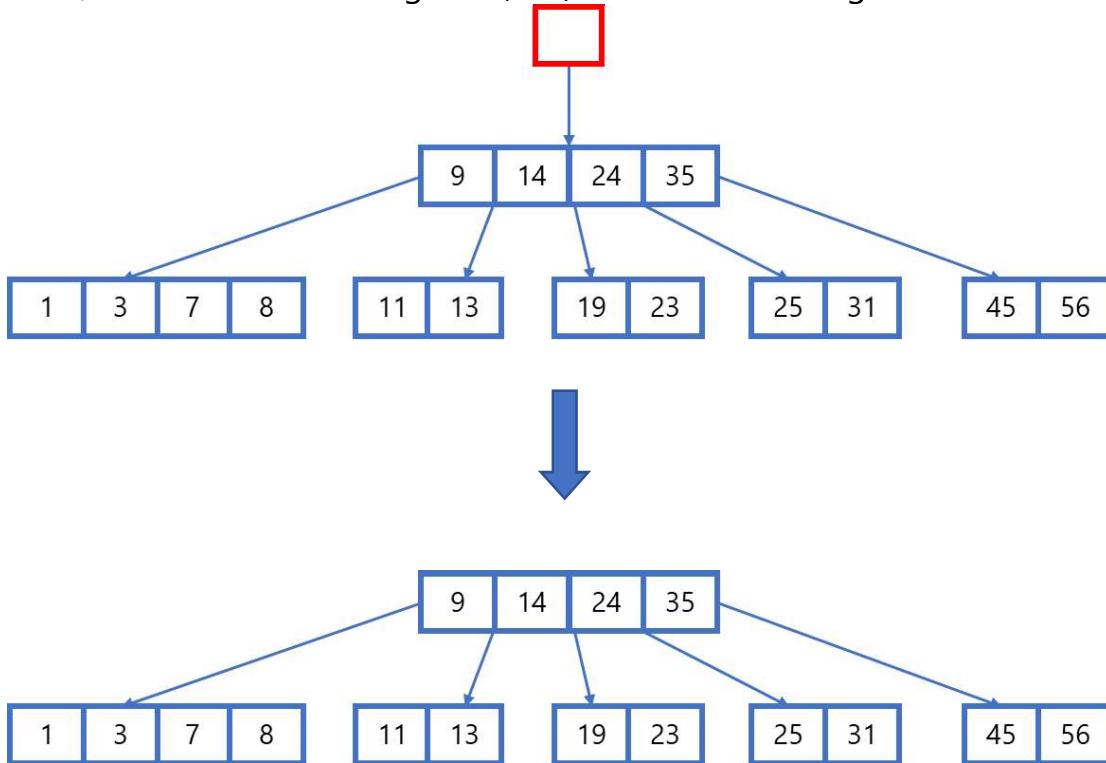
- Xóa khóa 5: Vì khóa 5 nằm ở nút trung gian, nên ta tìm khóa thế mạng, có thể chọn khóa 3 (phải nhất cây trái) hoặc 7 (trái nhất cây phải). Ở đây, giả sử ta chọn 3 là khóa thế mạng.
- Sau khi đẩy khóa 3 lên thay thế, ta quy việc xóa khóa trung gian về xóa 1 khóa ở nút lá (1,3).



- Và nút lá (1, 3) sau khi đem khóa 3 lên thế mạng thì số khóa không đảm bảo số lượng khóa tối thiểu trong 1 nút của cây bậc 5. Xét nút kề của nút 1 là nút (7, 8) chỉ có 2 khóa (không nhiều hơn k khóa) nên không thể thực hiện **underflow**.
- Vì vậy, ta phải thực hiện **catenate**.
- Ta tiến hành gom các nút anh chị của nút 1 (có chung cha) và khóa cha của nó thành 1 nút mới (ta quy ước khóa 3 là khóa cha của nút 1 và nút (7, 8)), và các nút được gom lại thành 1 nút sẽ làm con của nút chứa khóa cha. (Trong ví dụ đang xét, nút 1 chỉ có 1 khóa cha duy nhất là 3, và 1 nút anh chị duy nhất là (7, 8), nên chỉ có 1 cách gom duy nhất là gom 1, 3, 7, 8 thành 1 nút. Nếu gặp trường hợp có nhiều hơn 1 lựa chọn, tức nút 1 có 2 khóa cha và 2 nút anh chị, thì tùy theo yêu cầu xóa của đề bài để chọn gom bên nhánh anh chị nào (nếu không nói gì thêm thì cứ chọn 1 bên trái hoặc phải để gom). Như trong ví dụ trên, nút 1 chỉ có nhánh anh chị bên phải nên ta sẽ gom nó thành 1 nút mới). Và sau khi gom:



- Sau khi gom, nút 9 chỉ có 1 khóa, không đủ số khóa tối thiểu. Xét các nút kề của nút 9 thì không có nút nào có nhiều hơn 2 khóa (2 khóa) nên không thể thực hiện **underflow**. Vậy nên ta sẽ tiến hành **catenate**, gom 9, 14, 24, 35 thành 1 nút. Lúc này, khóa của nút gốc đã bị gom lại thành nút mới (9, 14, 24, 35) nên số khóa của nút gốc chỉ còn là 0, nhưng vì nút gốc không còn khóa nào nên không cần phải thực hiện underflow hay catenate, mà nút con của nút gốc hiện tại sẽ trở thành nút gốc.



7.5. Bài tập

BT 7.5.1 Cho dãy số sau: **11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 31**

Hãy thực hiện các yêu cầu sau:

- Xây dựng cây nhị phân tìm kiếm từ dãy số đã cho vào cây theo thứ tự thêm các số từ trái sang phải của dãy số.
- Duyệt cây trong câu a theo **NLR, RLN**.
- Xóa khỏi cây lần lượt các nút 8, 11, 43, 6 (vẽ hình từng trường hợp) sao cho cây vẫn là cây nhị phân tìm kiếm sau khi xoá nút.
- Viết hàm in ra màn hình các nút trên cây có duy nhất một nút con.
- Viết hàm đếm số lượng nút lá có trên cây.

BT 7.5.2 Cho dãy ký tự như sau: **F, D, B, A, C, E, H, G, I**

Hãy thực hiện các yêu cầu sau:

- Vẽ cây nhị phân tìm kiếm bằng cách thêm lần lượt từng ký tự vào cây theo thứ tự từ trái qua phải của dãy ký tự trên, biết rằng giá trị của từng ký tự tương ứng



theo thứ tự xuất hiện của ký tự trong từ điển.

- b. Cho biết kết quả duyệt cây theo **RNL**, **NRL**.
- c. Huỷ lần lượt từng nút **D**, **E**, **F**, **H** trên cây, mỗi lần huỷ 1 nút vẽ lại cây nối tiếp theo như thứ tự huỷ.
- d. Viết hàm đếm số lượng nút có một nút con trên cây, nếu cây rỗng thì in ra giá trị -1.

BT 7.5.3 Giả sử cho thông tin một sinh viên bao gồm các thông tin:

- Mã sinh viên: dạng số nguyên dương, mã sinh viên là giá trị duy nhất để phân biệt
- Họ sinh viên: dạng chuỗi
- Tên sinh viên: dạng chuỗi
- Điểm trung bình học tập: dạng số thực, có miền giá trị từ 0.0 đến 10.0

Hãy thực hiện các yêu cầu sau:

- a. Khai báo cấu trúc cây nhị phân tìm kiếm để lưu thông tin sinh viên theo mô tả ở trên.
- b. Viết hàm thêm các sinh viên vào cây.
- c. Viết hàm xóa các sinh viên có điểm trung bình < 5.0 ra khỏi cây.
- d. Viết hàm hiển thị danh sách sinh viên khi duyệt cây theo thứ tự trước.

BT 7.5.4 Hãy tạo cây B-Tree bậc 3:

- a. Lần lượt thêm các khóa **A**, **D**, **Z**, **B**, **F**, **G**, **H**, **O**, **N**, **P**, **X**, **C** vào cây. Và cho biết ở thao tác nào thì có thao tác **split node**.
- b. Lần lượt xóa các khóa **P**, **D**, **F**, **C** khỏi cây. Xóa khóa nào thì chỉ cần thực hiện thao tác **underflow**, khóa nào thì phải thực hiện **catenate**.

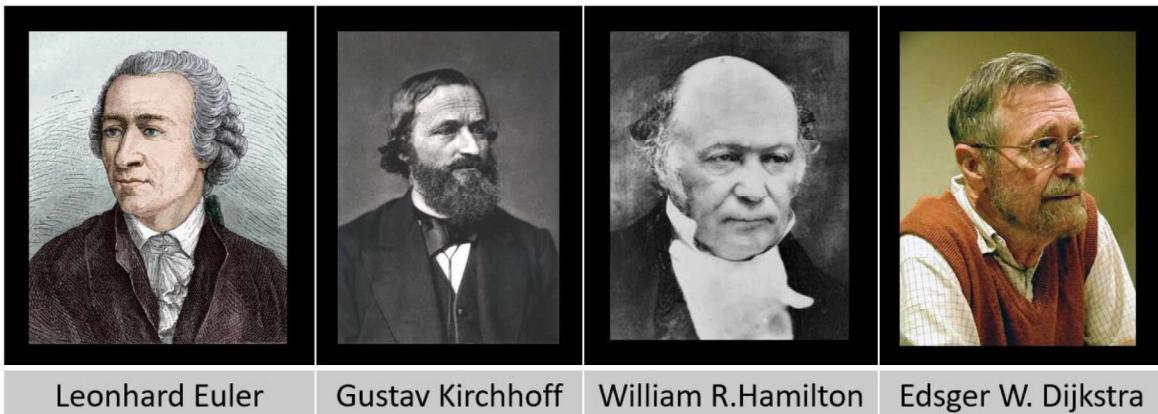
BT 7.5.5 Hãy tạo cây B-Tree bậc 5:

- a. Lần lượt thêm các khóa **C**, **N**, **G**, **A**, **H**, **E**, **K**, **Q**, **M**, **F**, **W**, **L**, **T**, **Z**, **D**, **P**, **R**, **X**, **Y**, **S** vào cây. Và cho biết ở thao tác nào thì có thao tác **split node**.
- b. Lần lượt xóa các khóa **H**, **T**, **R**, **E**, **C** khỏi cây. Xóa khóa nào thì chỉ cần thực hiện thao tác **underflow**, khóa nào thì phải thực hiện **catenate**.

Chương 8. Đồ thị (Graph)

8.1. Sơ lược về lịch sử hình thành của đồ thị

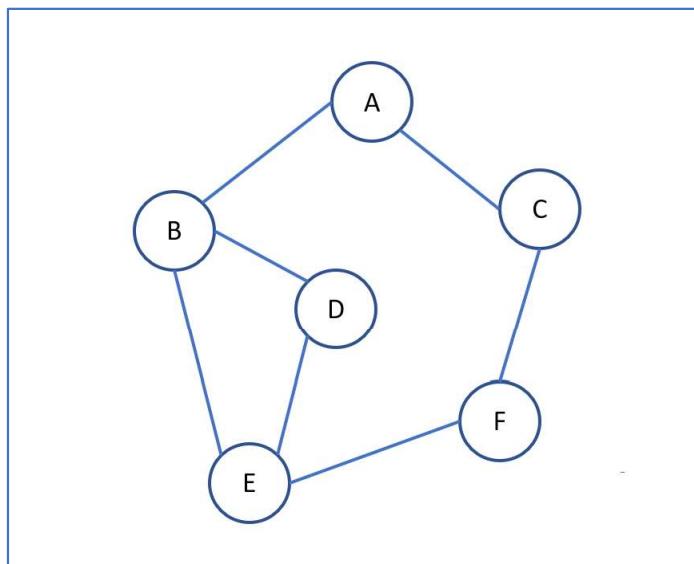
- Năm 1736: Leonhard Euler, nhà toán học người Thụy Sĩ, được coi là cha đẻ của lý thuyết đồ thị. Trong bài toán Cầu Königsberg, Euler đã giới thiệu khái niệm về đồ thị và sử dụng nó để giải quyết bài toán này. Euler cũng đưa ra định lý Euler, nền tảng cho nhiều phát triển sau này trong lĩnh vực lý thuyết đồ thị.
- Trong thế kỷ 19, các nhà toán học như Gustav Kirchhoff và William Rowan Hamilton tiếp tục nghiên cứu lý thuyết đồ thị và áp dụng nó vào lĩnh vực vật lý và toán học khác. Kirchhoff đã phát triển lý thuyết mạng điện và áp dụng nó vào giải quyết các vấn đề trong hệ thống mạng. Hamilton đã đưa ra khái niệm về đồ thị Hamilton và bài toán du lịch điểm qua tất cả các đỉnh một lần.
- Trong thế kỷ 20, lý thuyết đồ thị tiếp tục phát triển mạnh mẽ. Đặc biệt, các nhà toán học như Paul Erdős và Edgar Gilbert đã nghiên cứu các tính chất và cấu trúc của các đồ thị ngẫu nhiên. Các phân nhóm đồ thị và các khái niệm khác như đồ thị siêu cơ bản (fundamental graph) và đồ thị cấu trúc (structural graph) cũng được đề xuất.
- Trong thế kỷ 21, lý thuyết đồ thị tiếp tục phát triển và được ứng dụng rộng rãi trong nhiều lĩnh vực. Đồ thị xã hội đã trở thành một lĩnh vực quan trọng, nghiên cứu các mạng xã hội và các tương tác xã hội. Ngoài ra, các thuật toán đồ thị, như thuật toán Dijkstra và thuật toán tìm kiếm đồ thị rộng trước (BFS), được sử dụng trong nhiều ứng dụng thực tế như hệ thống định tuyến và tìm kiếm trên web.



8.2. Định nghĩa và các khái niệm liên quan

8.2.1. Đồ thị (graph) là gì?

- Đồ thị (graph) trong lĩnh vực cấu trúc dữ liệu là một cấu trúc dữ liệu phi tuyến tính (non-linear) bao gồm tập các đỉnh (vertex) và tập các cạnh (edge). Ký hiệu: $G(V, E)$
- Đỉnh (vertex ~ node, point):** là đơn vị cơ bản của đồ thị, mỗi node có thể được dán nhãn hoặc không.
- Cạnh (Edge ~ link, line):** cạnh trong đồ thị thể hiện sự kết nối giữa 2 đỉnh (node), mỗi cạnh có thể được dán nhãn hoặc không.



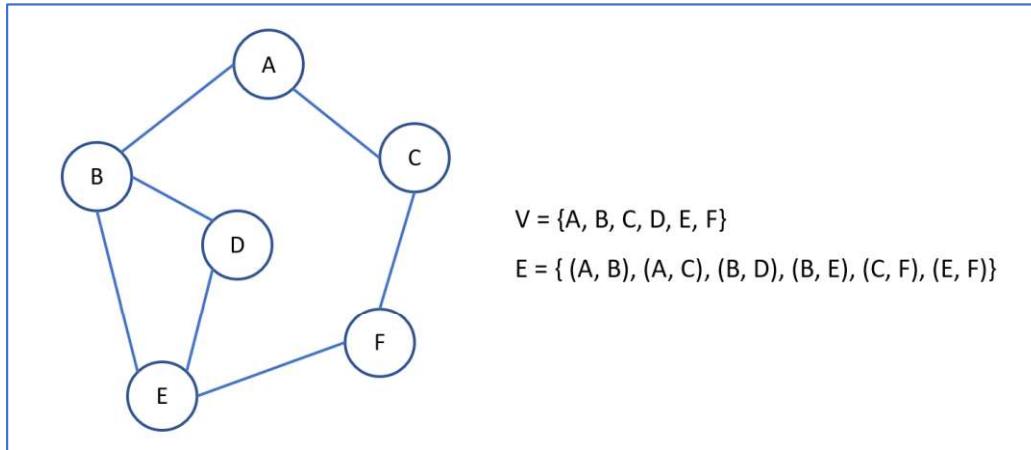
- Trong thực tế, đồ thị có rất nhiều ứng dụng:
 - Mạng xã hội: Đồ thị được sử dụng để mô hình hóa và phân tích các mạng xã hội như Facebook, Twitter, LinkedIn và các nền tảng mạng xã hội khác. Nó giúp phân tích mối quan hệ giữa các người dùng, tìm kiếm cộng đồng, phân tích thông tin lan truyền và gợi ý bạn bè.
 - Hệ thống giao thông: Đồ thị được sử dụng trong quản lý giao thông và định tuyến trong thành phố. Nó giúp mô phỏng và tối ưu hóa mạng lưới giao thông, xác định các tuyến đường tối ưu và dự đoán lưu lượng giao thông.
 - Tìm kiếm trên web: Đồ thị được sử dụng trong các công cụ tìm kiếm trên web để xác định tương quan và mức độ liên quan giữa các trang web và tìm kiếm kết quả tối ưu.
 - ...

8.2.2. Đồ thị vô hướng (undirected graph)

Đồ thị $G = (V, E)$ là một đồ thị vô hướng:

- Tập đỉnh V
- Tập cạnh E

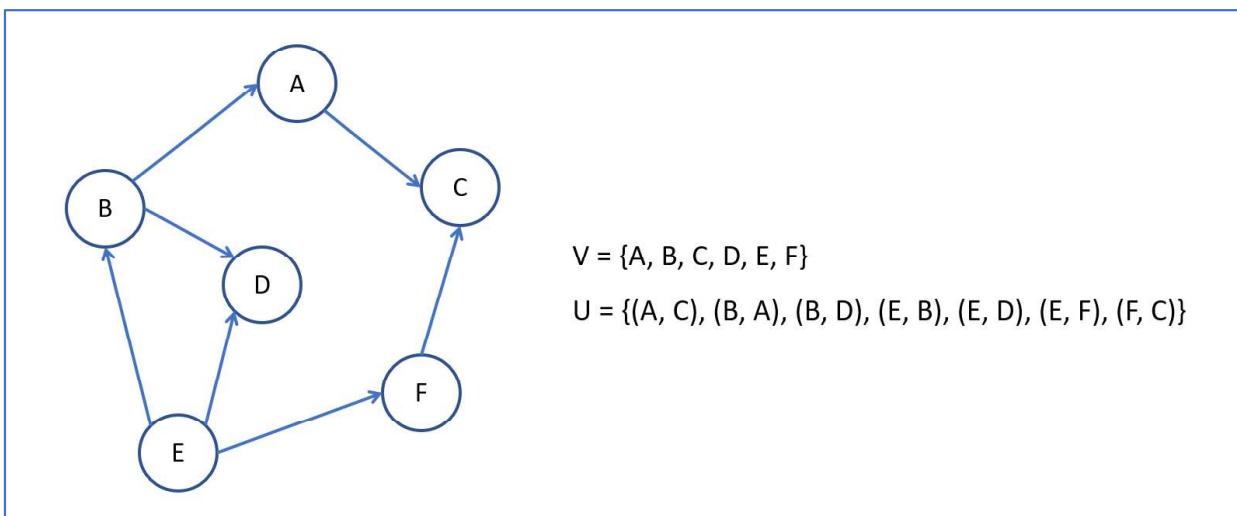
- Mỗi cạnh $e \in E$ được liên kết với một cặp đỉnh $\{i, j\} \in V^2$ không có sự phân biệt giữa đỉnh đầu và đỉnh cuối của mỗi cạnh.



8.2.3. Đồ thị có hướng (directed graph hoặc digraph)

Đồ thị $G = (V, U)$ là một đồ thị có hướng:

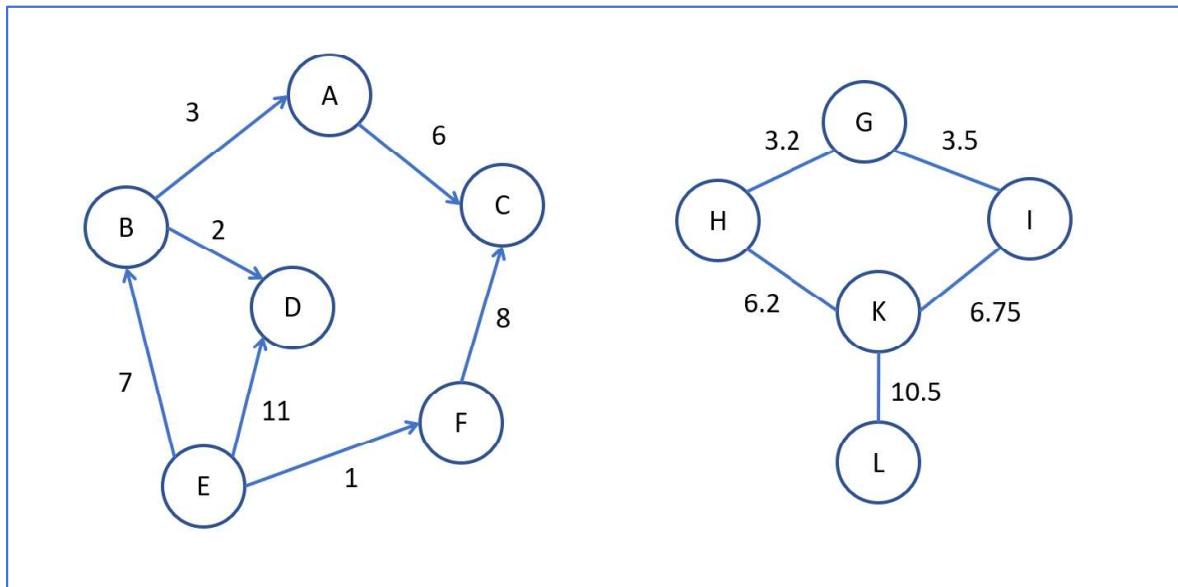
- Tập đỉnh V
- Tập cạnh U
- Mỗi cạnh $u \in U$ được liên kết với một cặp đỉnh $\{i, j\} \in V^2$ có sự phân biệt giữa đỉnh đầu và đỉnh cuối của mỗi cạnh. Mỗi cạnh chỉ cho phép di chuyển từ đỉnh đầu đến đỉnh cuối của cạnh



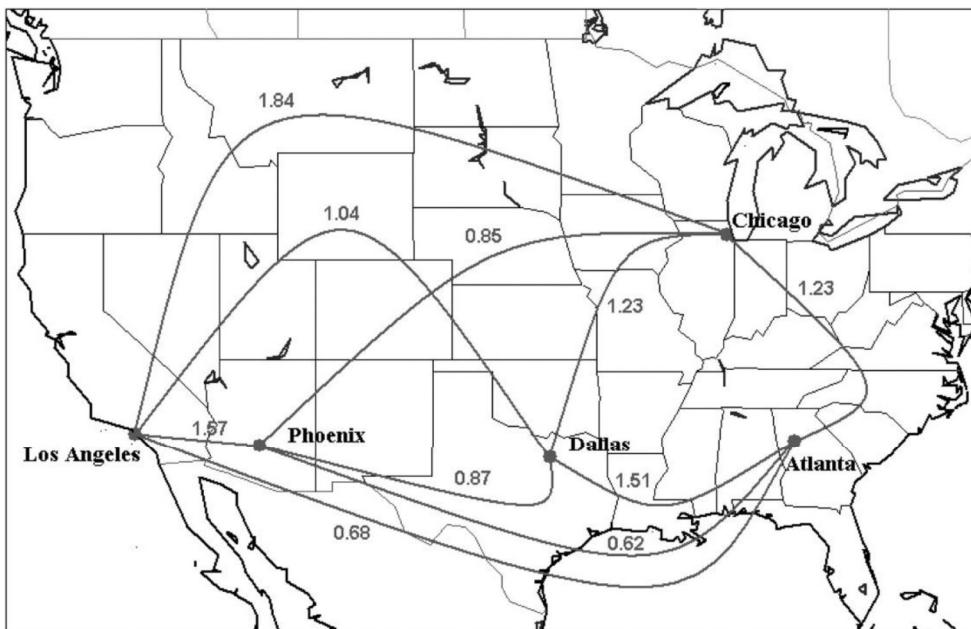
8.2.4. Đồ thị có trọng số (weighted graph)

- Đồ thị có trọng số (Weighted graph) là một loại đồ thị trong đó mỗi cạnh được gán một giá trị số gọi là trọng số.

- Trọng số đại diện cho một thuộc tính, độ quan trọng hoặc khoảng cách giữa hai đỉnh trong đồ thị. Trọng số của mỗi cạnh có thể được biểu diễn bằng một số thực dương hoặc số nguyên
- Đồ thị có trọng số cho phép mô hình hóa và phân tích các vấn đề phức tạp hơn trong thực tế, như tìm kiếm đường đi ngắn nhất dựa trên trọng số, tối ưu hóa đường đi dựa trên chi phí hoặc tìm kiếm cấu trúc mạng phù hợp dựa trên sự tương tự,...

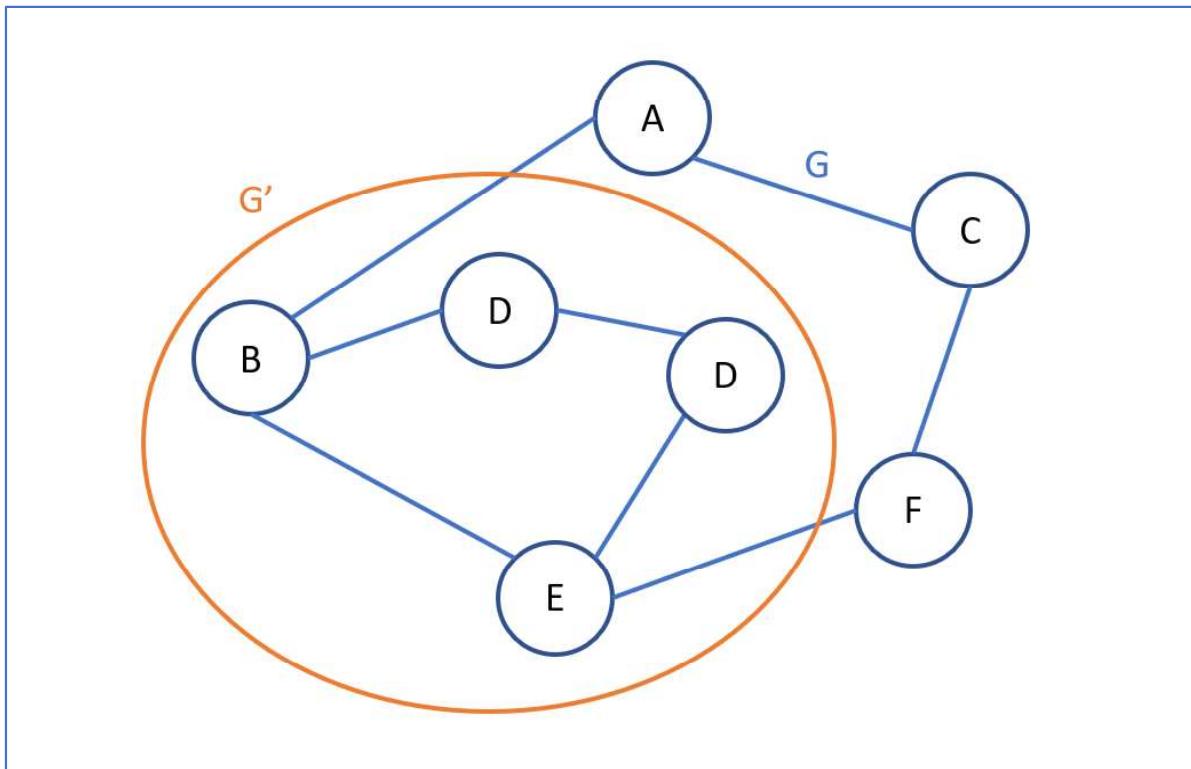


- Mô hình hóa các hướng di chuyển trong giao thông hàng không



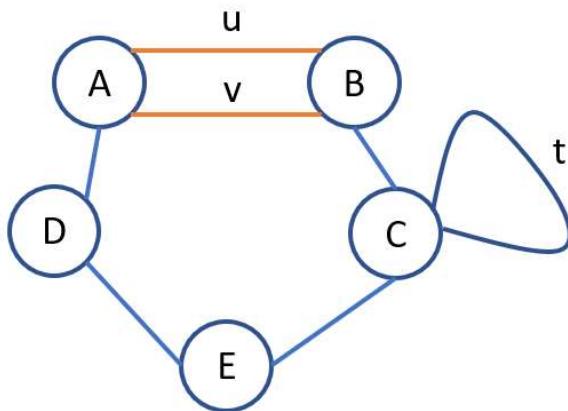
8.2.5. Đồ thị con (subgraph)

- Đồ thị con (Subgraph) là một đồ thị nhỏ hơn được tạo thành bằng cách chọn một tập hợp con của các đỉnh và cạnh từ một đồ thị gốc. Đồ thị con giữ lại một phần của đồ thị gốc bằng cách loại bỏ một số đỉnh và/hoặc cạnh.
- Xét 2 đồ thị $G = (X, U)$, $G' = (X', U')$. G' là con của G . Ký hiệu: $G' \in G$ thỏa mãn:
 - $X' \in X, U' \in U$
 - $u = (i, j) \in U$ của G , nếu $u \in U'$ thì $i, j \in X'$



8.2.6. Các dạng đồ thị

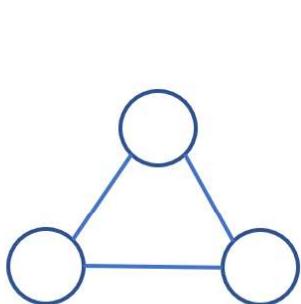
- Đồ thị rỗng (Empty graph ~ null graph):** là một đồ thị không chứa bất kỳ đỉnh nào. Nó không có đỉnh và không có cạnh. Trong đồ thị rỗng, không có mối quan hệ hoặc liên kết nào giữa các đỉnh, vì không có đỉnh nào tồn tại để tạo ra các mối quan hệ đó.
- Đơn đồ thị (Simple graph):** là đồ thị không có các cạnh song song (đa cạnh) giữa hai đỉnh và không có cạnh nối một đỉnh với chính nó (khuyên).
- Đa đồ thị (Multigraph):** Là đồ thị có các cạnh song song và tùy theo định nghĩa mà nó có khuyên hay không.



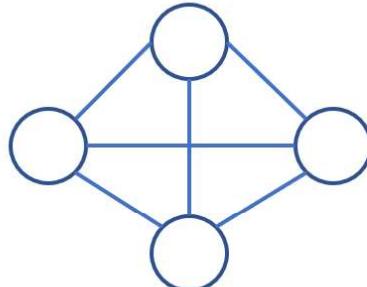
Giữa 2 đỉnh A và B tồn tại cặp **cạnh** (u,v) **song song**
t là **khuyên**

- **Đồ thị đầy đủ (Complete graph):** là đồ thị trong đó mỗi cặp đỉnh bất kỳ đều được nối bằng đúng một cạnh. Ký hiệu K_n với n là số đỉnh của đồ thị

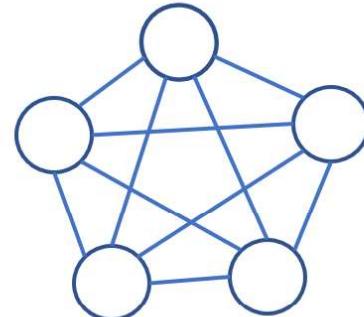
$$\text{Số cạnh} = \frac{n(n - 1)}{2}$$



Đồ thị K_3

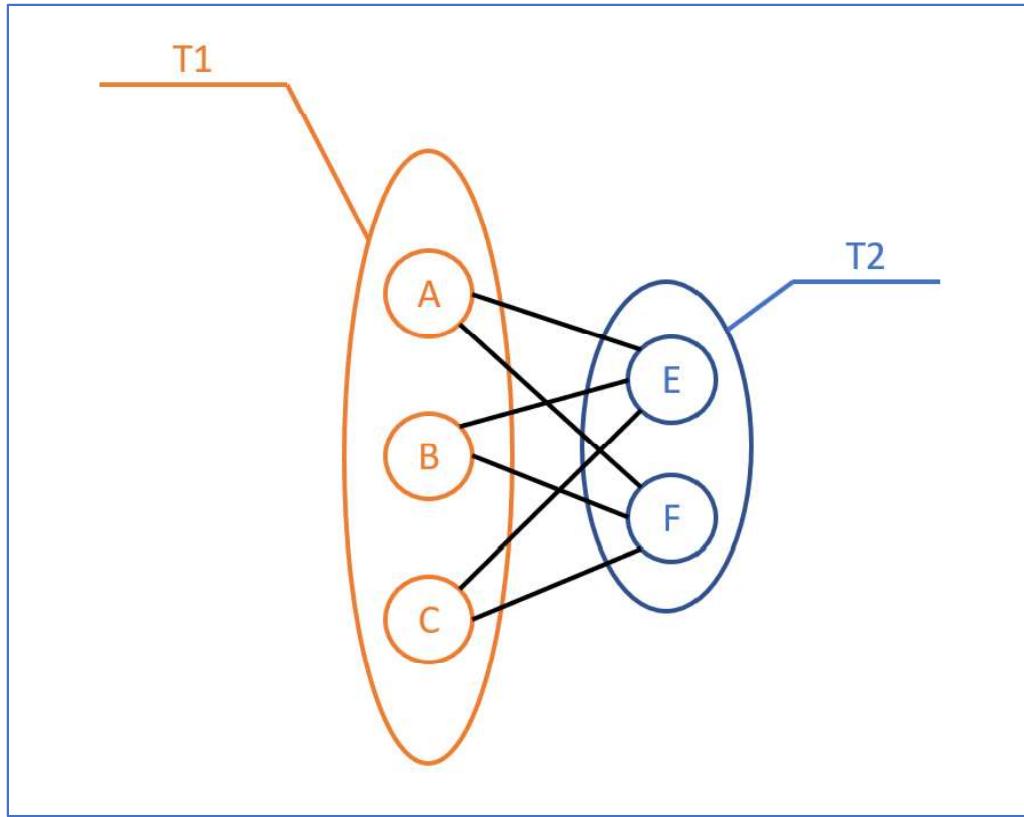


Đồ thị K_4



Đồ thị K_5

- **Đồ thị lưỡng phân (Bipartite graph):** là đồ thị mà tập đỉnh của nó có thể được chia thành hai tập **disjoint** (không có phần tử chung) sao cho mọi cạnh chỉ nối giữa các đỉnh thuộc hai tập khác nhau, không có cạnh nối giữa hai đỉnh cùng tập.
- **Hình minh họa:** Tập $T_1 = \{A, B, C\}; T_2 = \{E, F\}$.



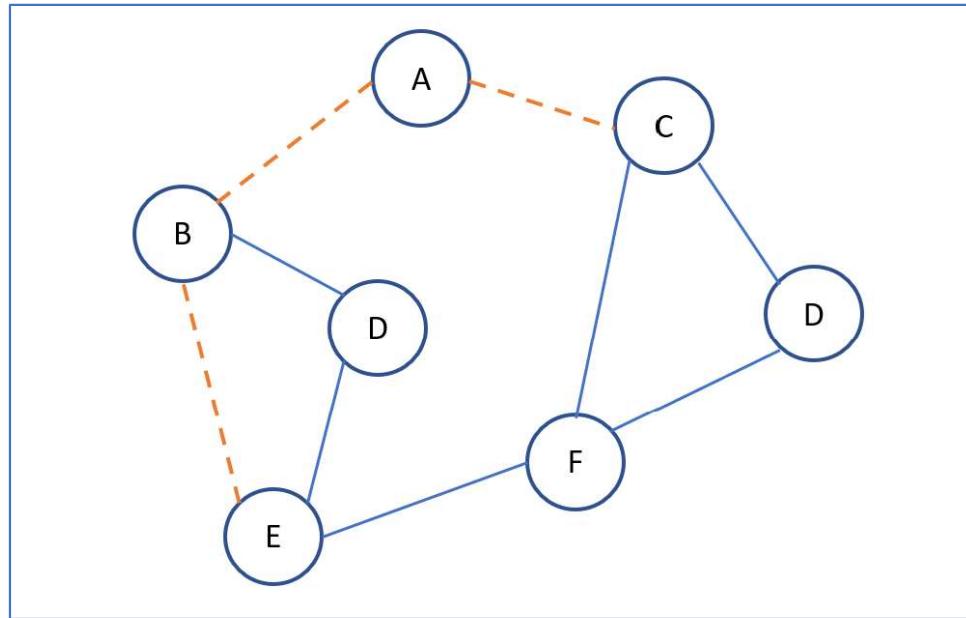
8.2.7. Đường đi, chu trình

- Đường đi:

(*) Cho đồ thị $G=(X, U)$:

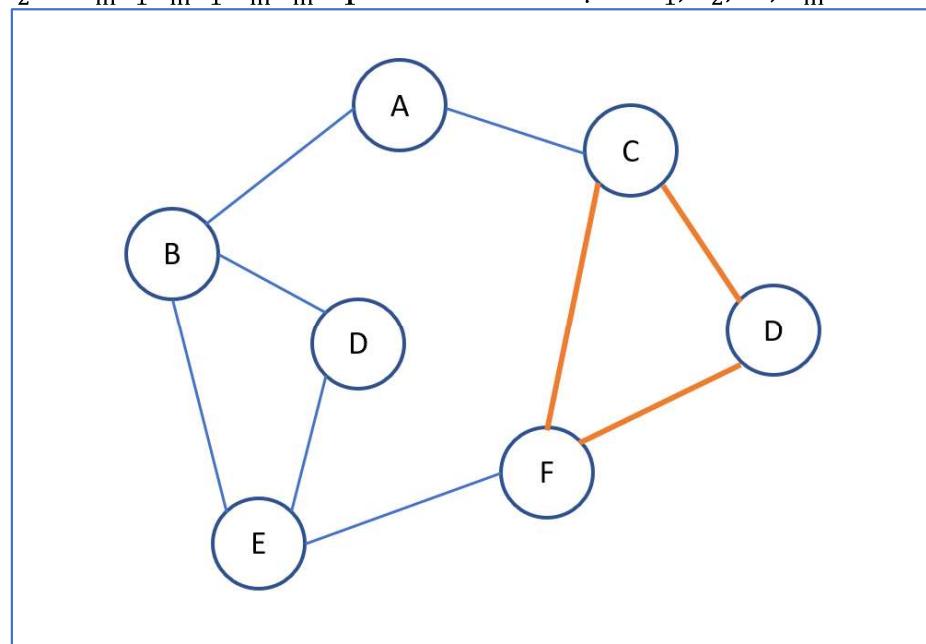
- Một đường đi trong G là một dãy luân phiên các đỉnh và cạnh:
 $x_1 u_1, x_2 u_2 \dots x_{m-1} u_{m-1} x_m$ (x_i là đỉnh và u_i là cạnh)
- Trong đồ thị thỏa mãn điều kiện u_i liên kết với cặp đỉnh (x_i, x_{i+1}) , nghĩa là:
 - $u_i = (x_i, x_{i+1}) \neq (x_{i+1}, x_i)$ nếu đồ thị **có hướng**.
 - $u_i = \{x_i, x_{i+1}\}$ nếu đồ thị **vô hướng**.

Khi đó ta gọi x_1 là đỉnh đầu và x_m là đỉnh cuối của đường đi.



(*) E B A C là một đường đi

- **Chu trình:** Một chu trình trong đồ thị là một đường đi có đỉnh đầu trùng đỉnh cuối có dạng: $x_1 u_1 x_2 u_2 \dots x_{m-1} u_{m-1} x_m u_m x_1$ sao cho các cạnh u_1, u_2, \dots, u_m đôi một khác nhau.



(*) C D F C là một chu trình

- **Tính chất đơn và sơ cấp:**
 - Tính chất "**đơn**" của chu trình hay đường đi: không có cạnh nào xuất hiện hai lần trong chu trình hay đường đi đó.
 - Tính chất "**sơ cấp**" (hay cũng gọi là "thứ cấp"): không có đỉnh nào xuất hiện hai lần.
- **Chu trình đơn:** Là chu trình không đi qua cạnh nào quá một lần.

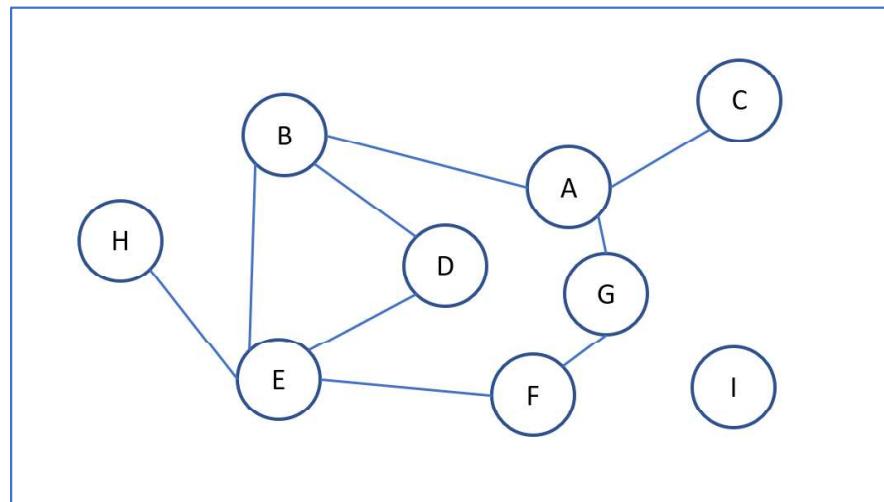
- **Chu trình sơ cấp:** là chu trình không chứa một đỉnh quá một lần (trừ đỉnh đầu và đỉnh cuối).

8.2.8. Đỉnh

- **Đỉnh kề (Adjacency vertex):** Hai đỉnh được coi là kề nhau nếu chúng được nối với nhau bởi một cạnh.
 - **Khuyên (Loop):** là một cạnh nối từ một đỉnh về chính nó.
 - **Đỉnh treo (hanging vertex):** là một đỉnh trong đồ thị chỉ có một cạnh kết nối nó với các đỉnh khác.
 - **Đỉnh cô lập (isolated vertex):** là một đỉnh trong đồ thị không có cạnh kết nối nó với bất kỳ đỉnh khác.
 - **Bậc của đỉnh (degree):**
 - **Đồ thị vô hướng:** bậc của một đỉnh trong đồ thị vô hướng là số lượng cạnh kết nối với đỉnh đó. Ký hiệu bằng "deg(v)" hoặc "d(v)".
 - Đỉnh treo có bậc là 1
 - Đỉnh cô lập có bậc là 0
 - **Đồ thị có hướng:** bậc của một đỉnh trong đồ thị có hướng là tổng số cạnh đi ra từ đỉnh đó và số cạnh đi vào đỉnh đó.
 - Bậc ra (outdegree): là số lượng cạnh đi ra từ một đỉnh. Ký hiệu: $\text{deg}^+(v)$
 - Bậc vào (indegree): là số lượng cạnh đi vào một đỉnh. Ký hiệu $\text{deg}^-(v)$
- Ta có: Bậc của một đỉnh v: $\text{deg}(v) = \text{deg}^+(v) + \text{deg}^-(v)$

- **Mối liên hệ giữa bậc – số cạnh:** Trong mọi đồ thị $G = (V, E)$, tổng số bậc của các đỉnh của G bằng 2 lần số cạnh của nó

$$|E| = \frac{1}{2} \sum_{v \in V} d(v)$$



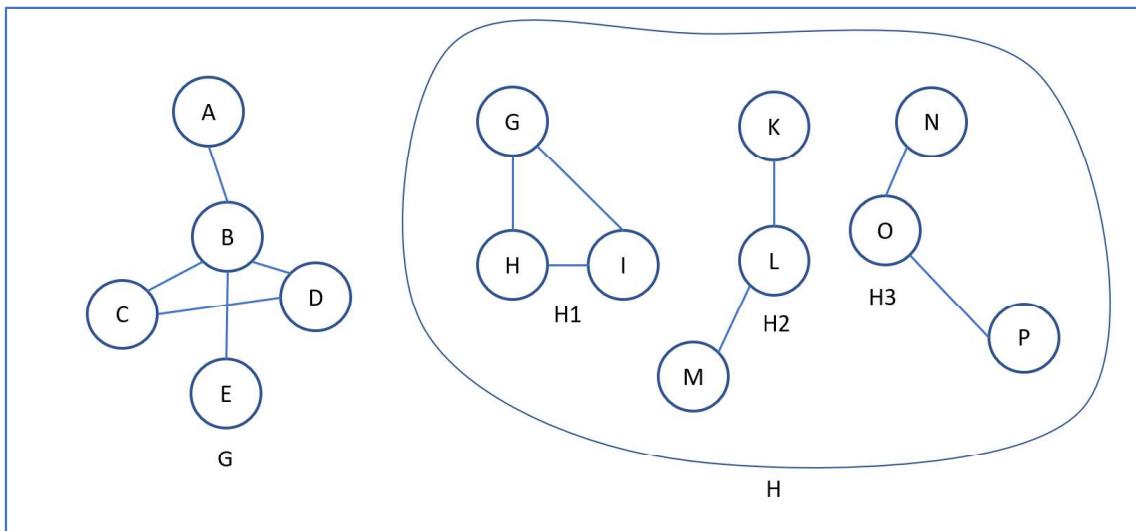
Trong đó:

- H, C là đỉnh treo ($\deg(H) = \deg(C) = 1$)
- I là đỉnh cô lập ($\deg(I) = 0$)

8.2.9. Liên thông (Connection)

8.2.9.1. Đồ thị vô hướng

- **Đồ thị liên thông:** Đồ thị vô hướng $G = (V, E)$ được gọi là liên thông nếu luôn tìm được đường đi giữa hai đỉnh bất kỳ của nó



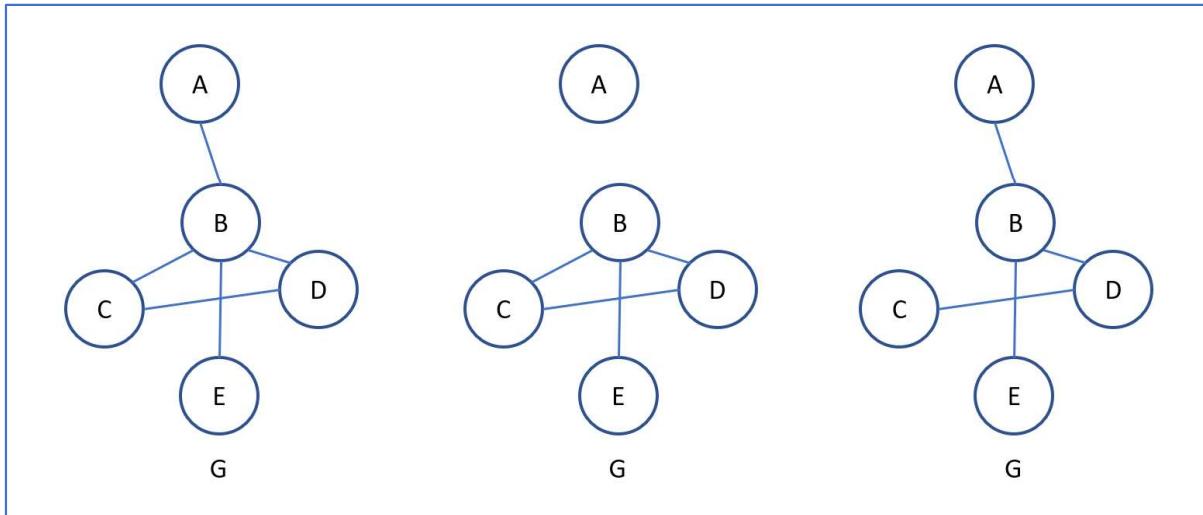
Nhận xét: G là đồ thị liên thông, H không phải là đồ thị liên thông.

- **Thành phần liên thông:** Trong trường hợp đồ thị là không liên thông, nó sẽ rã ra thành một số đồ thị con liên thông không có đỉnh chung. Những đồ thị con liên thông như vậy ta sẽ gọi là các thành phần liên thông của đồ thị.

Nhận xét: Đồ thị H gồm 3 thành phần liên thông là H1, H2, H3.

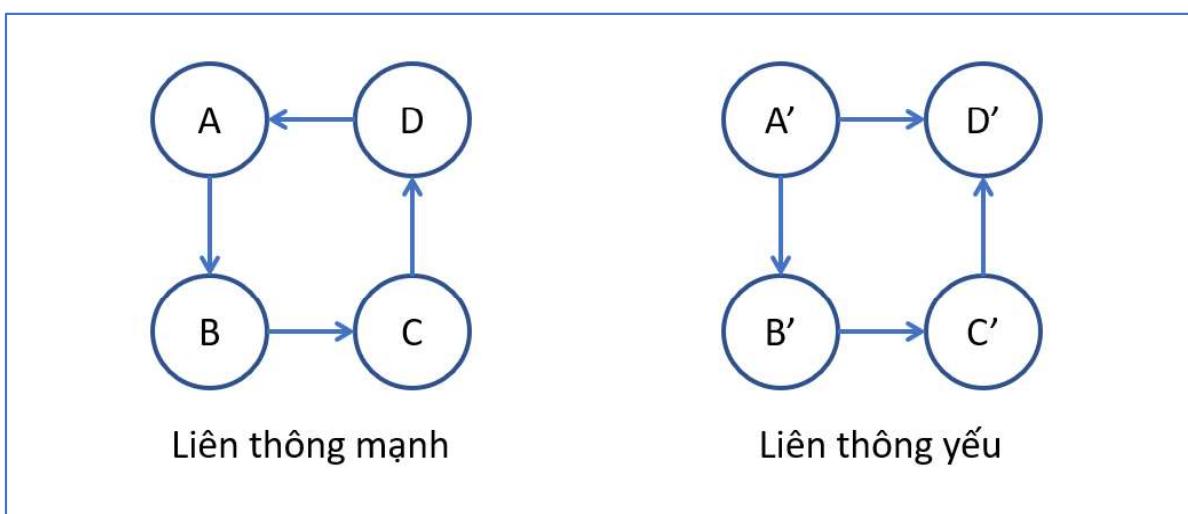
- **Cầu:** Cạnh e được gọi là cầu nếu việc loại bỏ nó khỏi đồ thị làm tăng số thành phần liên thông của đồ thị.

Ví dụ: Cạnh AB là cầu vì sau khi bỏ AB ta được 2 thành phần liên thông. Cạnh BC không phải là cầu vì sau khi bỏ BC đồ thị vẫn liên thông.



8.2.9.2. Đồ thị có hướng

- **Liên thông mạnh:** Đồ thị có hướng $G = (V, A)$ được gọi là liên thông mạnh nếu luôn tìm được đường đi giữa hai đỉnh bất kỳ của nó.
 - **Liên thông yếu:** Đồ thị có hướng $G = (V, A)$ được gọi là liên thông yếu nếu đồ thị vô hướng tương ứng với nó là vô hướng liên thông.
- (!)**Note:** Đồ thị liên thông mạnh thì cũng liên thông yếu.



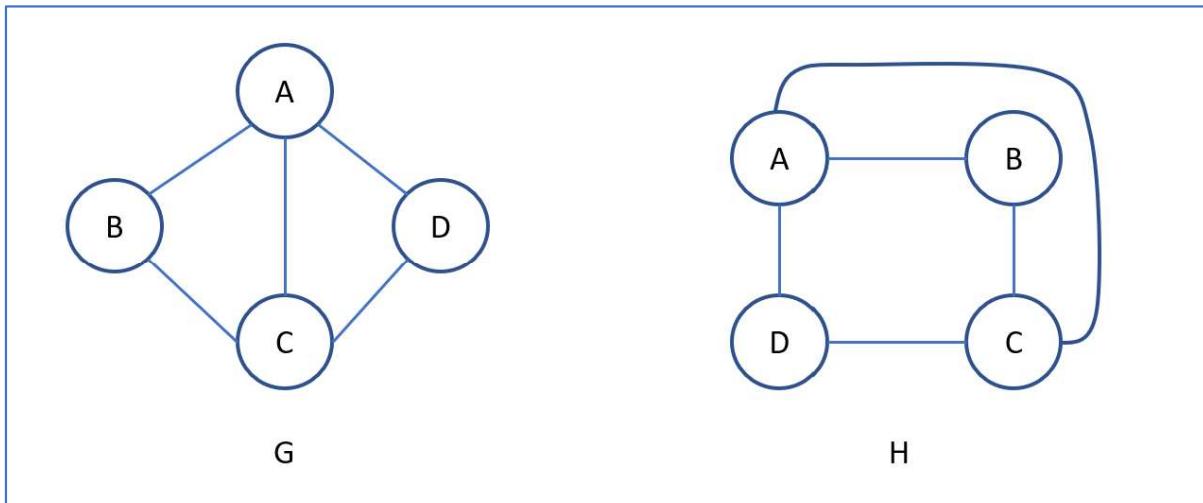
1.1.1. Đẳng cấu đồ thị (Graph isomorphism):

Khái niệm: Hai đồ thị G_1 và G_2 được gọi là đẳng cấu với nhau, ký hiệu là $G_1 \approx G_2$, nếu có thể vẽ lại (bằng cách dời đỉnh, dời cạnh...) sao cho hai đồ thị này có hình vẽ y hệt nhau.

Đặc điểm

- Cùng số đỉnh.
- Cùng số đỉnh bậc k, mọi k nguyên dương ≥ 0 .

- Cùng số cạnh.
- Cùng số thành phần.



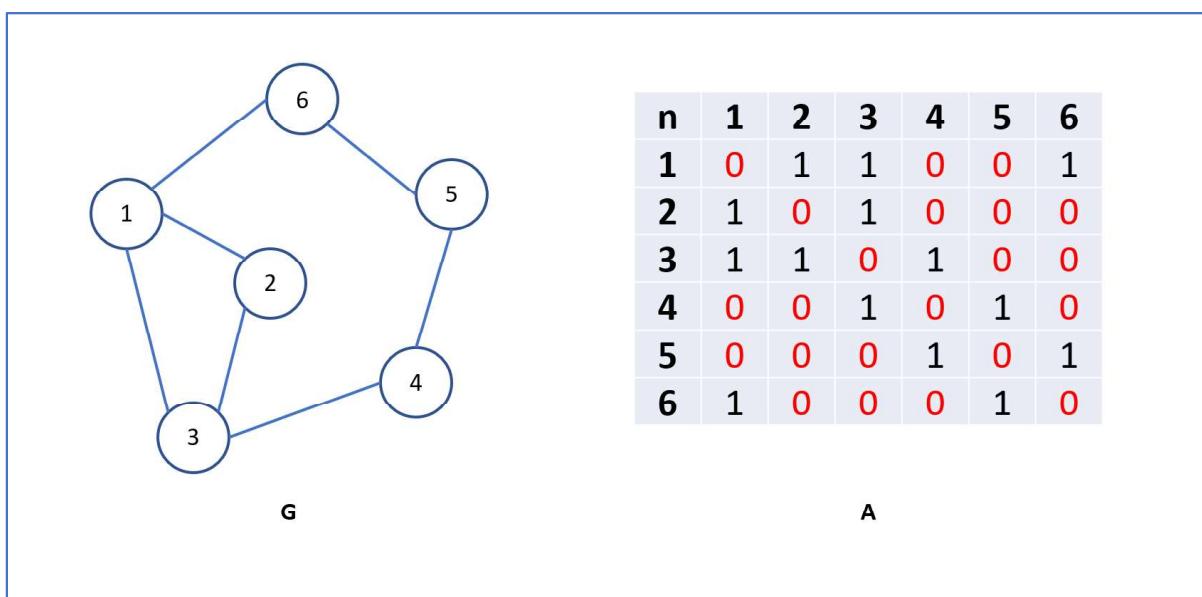
Nhận xét: Hai đồ thị G và H được gọi là đẳng cấu.

8.3. Biểu diễn đồ thị trên máy tính

8.3.1. Ma trận kề (Adjacency matrix)

Khái niệm: Ma trận kề là một cách biểu diễn đồ thị trên máy tính bằng ma trận vuông cấp n, trong đó n là số lượng đỉnh trong đồ thị (các đỉnh được đánh số từ 1 đến n). Các phần tử của ma trận có giá trị 0 hoặc 1 thể hiện mối liên kết giữa các đỉnh.

Đồ thị vô hướng: Xét ma trận kề A là biểu diễn của đồ thị G vô hướng



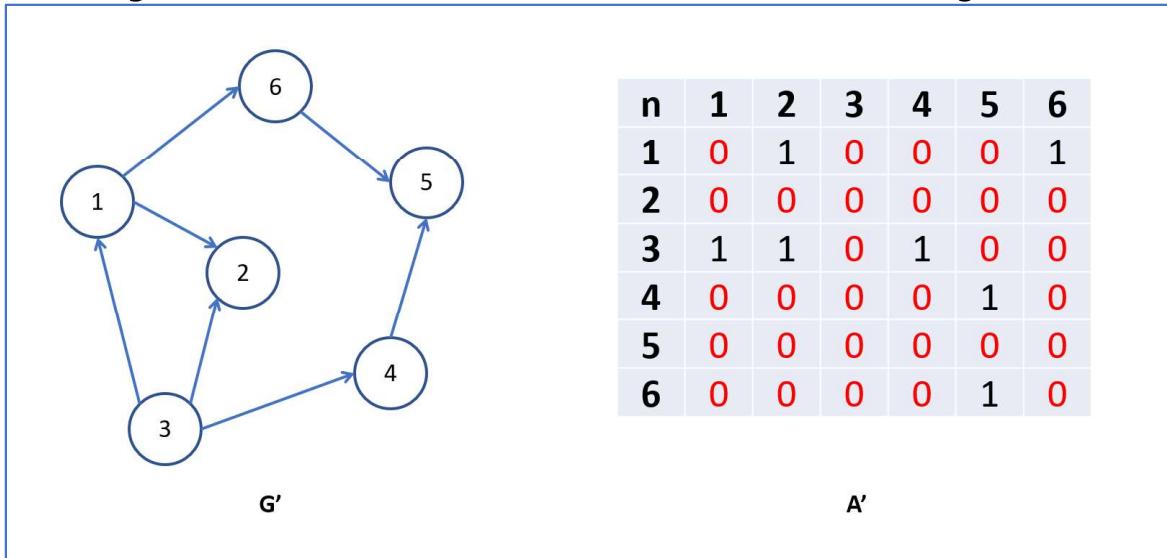
Nhận xét:

- $A[i][j] = 1 \rightarrow$ Có liên kết giữa 2 đỉnh
- $A[i][j] = 0 \rightarrow$ Không có liên kết giữa 2 đỉnh

Tính chất:

- Ma trận kề là ma trận đối xứng
- Tổng các phần tử trên ma trận bằng 2 lần số cạnh
- Tổng các phần tử trên hàng thứ k hoặc cột thứ k là bậc của đỉnh k

Đồ thị có hướng: Xét ma trận kề A' là biểu diễn của đồ thị G' có hướng



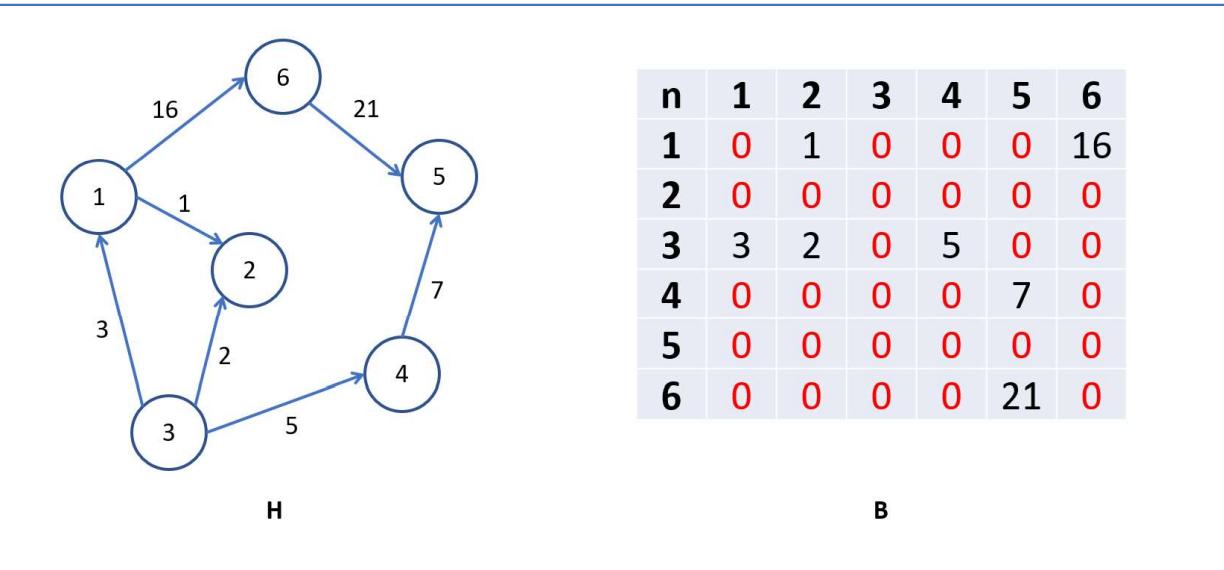
Nhận xét:

- $A[i][j] = 1 \rightarrow$ Có liên kết từ đỉnh i sang j.
- $A[i][j] = 0 \rightarrow$ Không có liên kết từ đỉnh i sang j.

Tính chất:

- Ma trận kề có thể không đối xứng
- Tổng các phần tử trên ma trận bằng số cạnh
- Tổng các phần tử trên hàng thứ k là tổng số bậc ra của đỉnh k
- Tổng các phần tử trên cột thứ k là tổng số bậc vào của đỉnh k

Đồ thị có trọng số: Xét ma trận kề B là biểu diễn của đồ thị trọng số H



Nhận xét:

- $A[i][j] = x \rightarrow$ Có liên kết từ đỉnh i sang j và trọng số là x.
- $A[i][j] = 0 \rightarrow$ Không có liên kết từ đỉnh i sang j và trọng số bằng 0.

Ưu điểm:

- Dễ dàng thể hiện quan hệ kề, nếu 2 đỉnh kề nhau thì giá trị tương ứng trong mảng khác 0
- Thể hiện thông tin đầy đủ: Ma trận kề cho phép lưu trữ toàn bộ thông tin về cấu trúc kề của đồ thị và mỗi quan hệ kề được biểu diễn rõ ràng.

Nhược điểm:

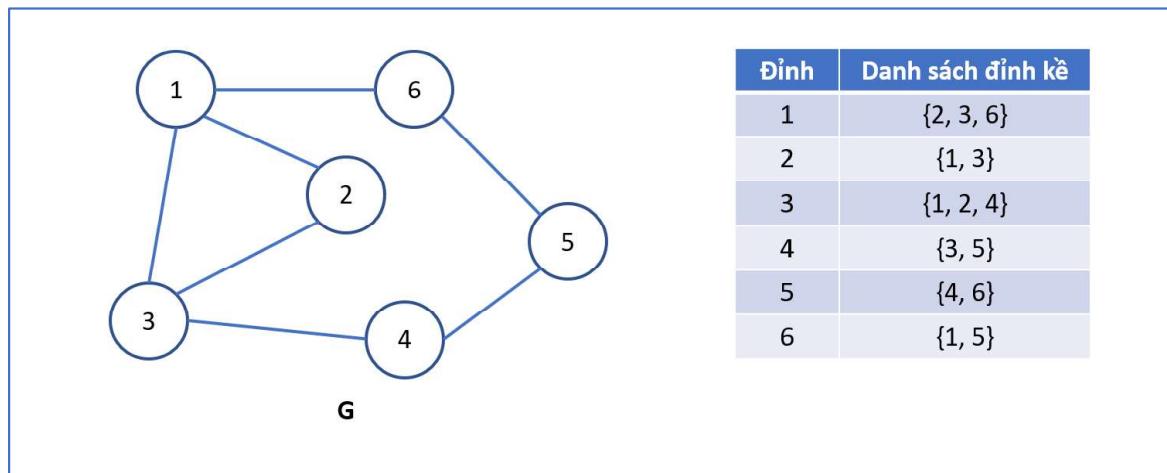
- Độ phức tạp lưu trữ: Với ma trận vuông $n \times n$ thì độ phức tạp lưu trữ là $O(n^2) \rightarrow$ không hiệu quả khi xử lý các đồ thị lớn với số lượng đỉnh lớn.
- Độ phức tạp tính toán: việc truy xuất hoặc cập nhật các quan hệ kề giữa các đỉnh có thể đòi hỏi thời gian và tài nguyên tính toán đáng kể, đặc biệt là đối với các đồ thị lớn.
- Không phù hợp cho đồ thị thưa: Nếu một **đồ thị là thưa** (số cạnh nhỏ hơn 6 lần số đỉnh) thì ma trận kề sẽ chứa rất nhiều phần tử 0 \Rightarrow gây lãng phí không gian lưu trữ.
- Khó mô hình hóa các thuộc tính đỉnh: Ma trận kề chỉ mô tả quan hệ kề giữa các đỉnh mà không thể dễ dàng thể hiện các thuộc tính riêng của mỗi đỉnh (màu sắc,...).

8.3.2. Danh sách kề (Adjacency list)

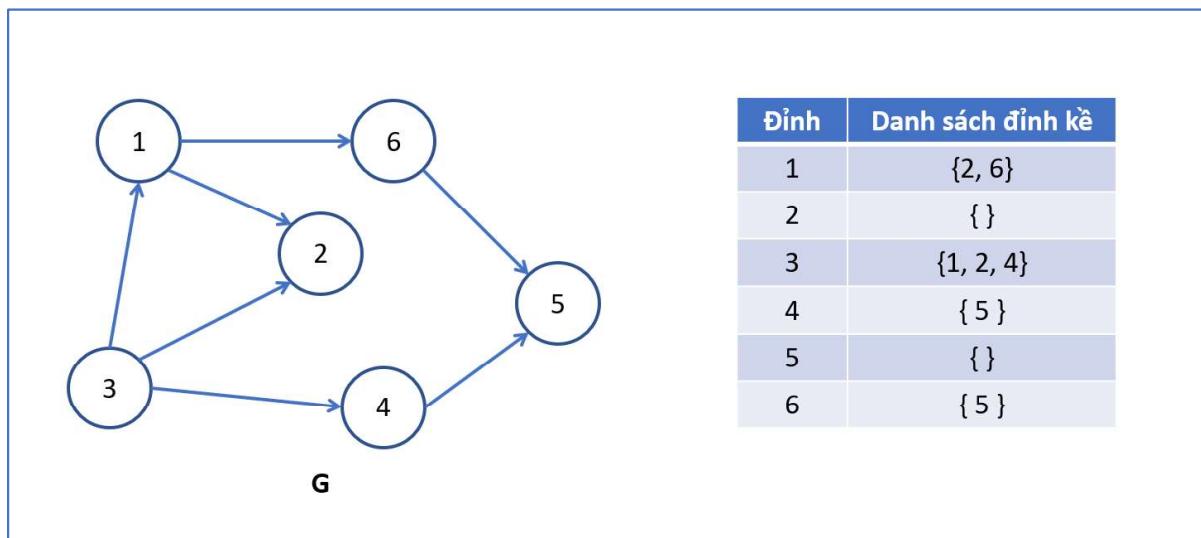
Khái niệm: là danh sách các mảng hoặc danh sách liên kết, trong đó mỗi phần tử tương ứng với một đỉnh trong đồ thị. Mỗi phần tử trong danh sách này chứa thông tin về các

đỉnh kề của đỉnh tương ứng đó. Trong C++ ta sử dụng một mảng vector để biểu diễn danh sách kề.

Đồ thị vô hướng:



Đồ thị có hướng:



Ưu điểm:

- Tiết kiệm không gian lưu trữ: chỉ lưu các cạnh và đỉnh kề của mỗi đỉnh
- Truy cập nhanh đến đỉnh kề: dễ dàng truy cập đến các đỉnh kề của một đỉnh cụ thể, hữu ích trong việc thực hiện các thuật toán đồ thị như duyệt theo chiều rộng (BFS) hoặc duyệt theo chiều sâu (DFS).
- Tối ưu về phương pháp biểu diễn

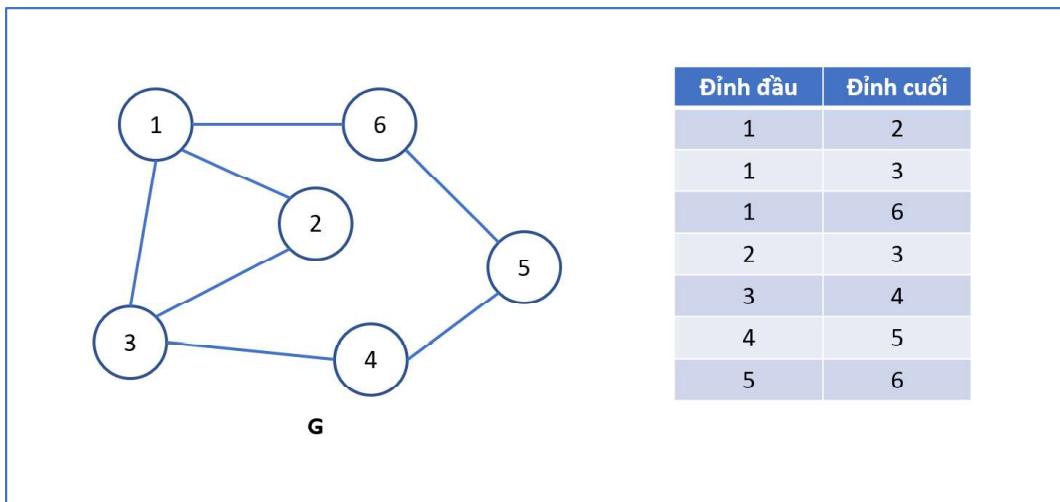
Nhược điểm:

- Không thích hợp cho đồ thị khác đồ thị thưa: danh sách kề có thể tốn nhiều không gian lưu trữ hơn so với ma trận kề.

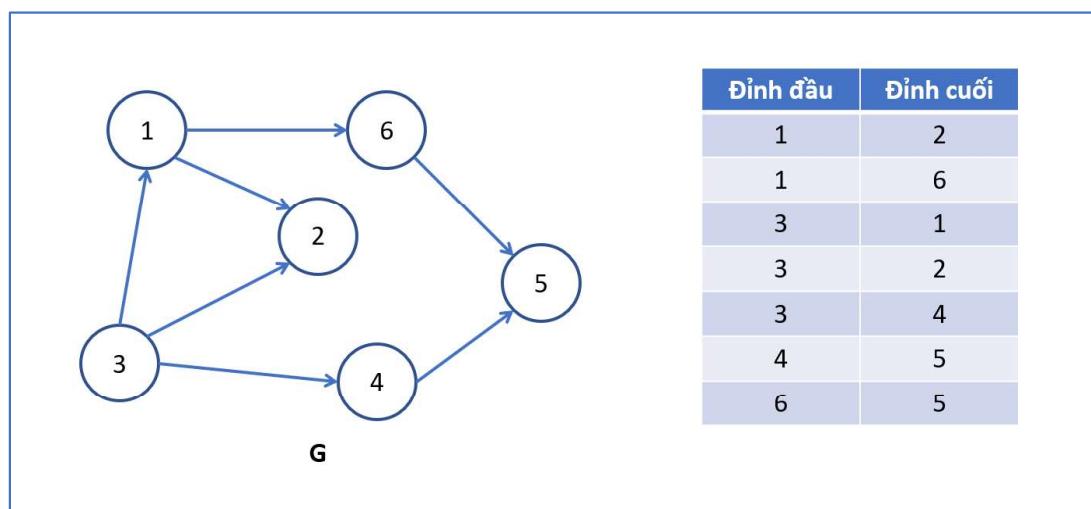
8.3.3. Danh sách cạnh (Edge list)

Khái niệm: Danh sách cạnh gồm một danh sách các cặp đỉnh, mỗi cặp đỉnh tương ứng với một cạnh trong đồ thị. Mỗi cạnh được biểu diễn bằng hai đỉnh mà nó kết nối. Nếu giả thuyết đầu vào là đồ thị có n đỉnh, m cạnh ta thường biểu diễn đồ thị dưới dạng danh sách cạnh (thông qua mảng hay danh sách liên kết).

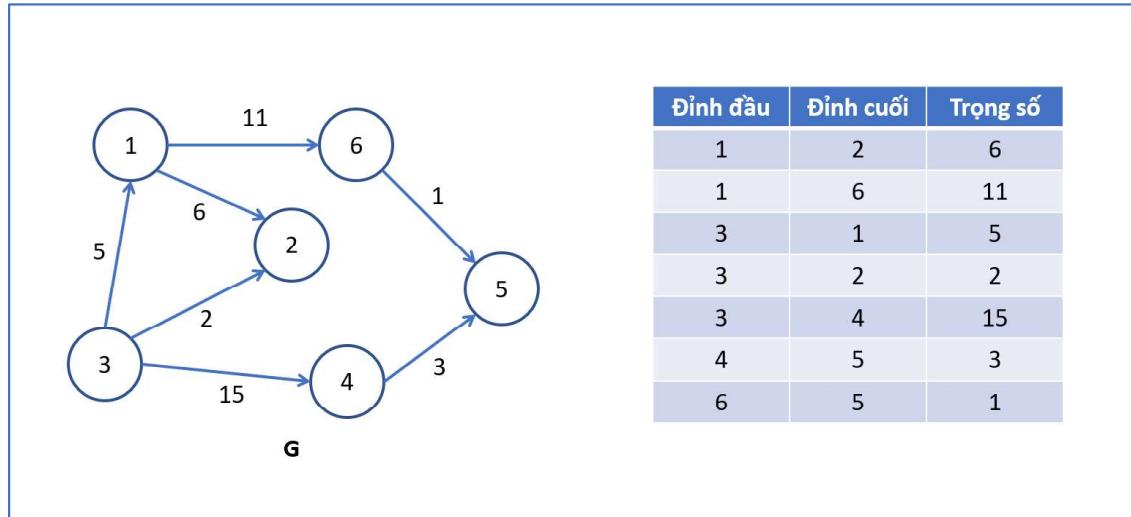
Đồ thị vô hướng: Nếu tồn tại cạnh giữa hai điểm i và j ta chỉ cần liệt kê cạnh (i, j) (khi $i < j$) hoặc (j, i) (khi $j < i$) theo thứ tự tăng dần của đỉnh đầu các cạnh.



Đồ thị có hướng: Nếu tồn tại cung giữa hai điểm i, j ta liệt kê theo hướng đỉnh đầu đến đỉnh cuối của cung (i, j)



Đồ thị có trọng số: Khi có trọng số, biểu diễn trên máy tính ta có thể tạo struct lưu 3 thành phần của danh sách là: Dinh_dau, Dinh_cuoi, Trong_so.



Ưu điểm:

- Tiết kiệm không gian lưu trữ: chỉ lưu trữ các cạnh thực sự tồn tại trong đồ thị => phù hợp cho đồ thị thưa (cạnh < 6 lần đỉnh)
- Linh hoạt: dễ dàng thêm, xóa và truy cập các cạnh trong đồ thị.
- Thích hợp cho các thuật toán xử lý đồ thị: tăng hiệu quả tính toán cho một số thuật toán đồ thị

Nhược điểm:

- Tốn thời gian tìm kiếm cạnh: Độ phức tạp là O(n) khi tìm cạnh trong danh sách có n cạnh
- Khó khăn truy xuất quan hệ kề: So với ma trận kề hoặc danh sách kề, việc truy xuất các đỉnh kề của một đỉnh trong danh sách cạnh có thể tốn kém hơn.

8.3.4. Cách chuyển đổi giữa các dạng biểu diễn của đồ thị

- Ma trận kề (**A**)
- Danh sách cạnh (**B**)
- Danh sách kề (**C**)

Chuyển từ B -> A:

```
void NhapBA(int a[][100], int& n, int& m)
{
    cout << "Nhập số đỉnh: ";
    cin >> n;
    cout << "Nhập số cạnh: ";
    cin >> m;
    cout << "Nhập danh sách cạnh: ";
    for (int i = 0; i < m; i++)
    {
        cout << "Cạnh " << i + 1 << ": ";
        int u, v, w;
        cin >> u >> v >> w;
        a[u][v] = w;
    }
}
```

```

cin >> n;
cout << "Nhập số đỉnh: ";
cin >> m;
for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++)
        a[i][j] = 0;
for (int i = 0; i < m; i++)
{
    int x, y;
    cout << "Nhập cạnh: ";
    cin >> x >> y;
    a[x][y] = 1;
    a[y][x] = 1;
}
}

void XuatBA(int a[][100], int n)
{
    cout << "Ma tran ke tuong ung voi danh sach canh: " << endl;
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= n; j++)
            cout << a[i][j] << " ";
        cout << endl;
    }
}

```

Chuyển từ B -> C:

```

void NhapBC(vector<int> a[100], int& n, int& m)
{
    cout << "Nhập số đỉnh DSC: ";
    cin >> n;
    cout << "Nhập số cạnh DSC: ";
    cin >> m;
    for (int i = 1; i <= m; i++)
    {
        int x, y;
        cin >> x >> y;
        a[x].push_back(y);
        a[y].push_back(x);
    }
}

```

```

        }
    }

void XuatBC(vector<int> a[100], int& n, int& m)
{
    cout << "Danh sach ke tuong ung voi Danh sach canh: " << endl;
    for (int i = 1; i <= n; i++)
    {
        cout << "Dinh " << i << ":" ;
        for (int x : a[i])
            cout << x << " ";
        cout << endl;
    }
}

```

Chuyển từ A -> B:

```

void NhapAB(int a[101][101], int& n)
{
    cout << "Nhập số đỉnh: ";
    cin >> n;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            cin >> a[i][j];
}

void XuatAB(int a[101][101], int n)
{
    cout << "Danh sách cạnh tương ứng với ma trận kề: " << endl;
    for (int i = 1; i <= n; i++)
    {
        for (int j = i + 1; j <= n; j++)
            if (a[i][j] == 1)
                cout << i << " " << j << endl;
    }
}

```

NOTE: Có thể sử dụng các cấu trúc dữ liệu như: vector, pair, set, map,... để tối ưu.

(*)Luyện tập: Sử dụng các cấu trúc dữ liệu khác và chuyển các cách biểu diễn:

- + A -> C
- + C -> A

+ C → B

8.4. Các thao tác duyệt đồ thị DFS, BFS

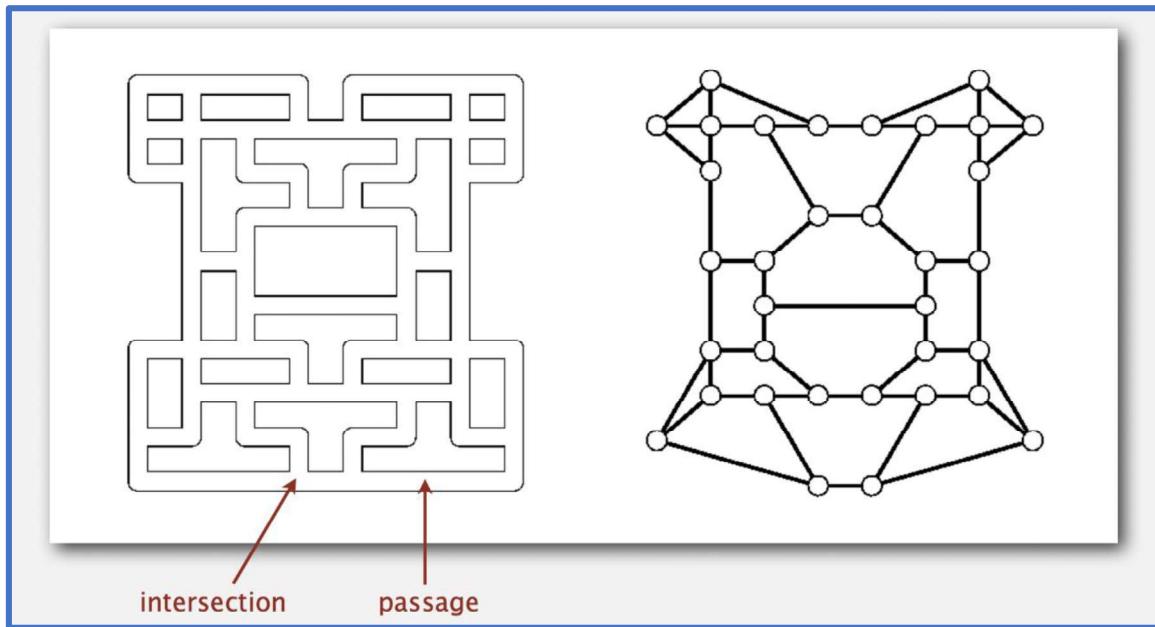
- Các thao tác trên đồ thị, như duyệt và tìm kiếm, là những kỹ thuật quan trọng trong lý thuyết đồ thị và được áp dụng rộng rãi trong nhiều bài toán thực tế.
- Hai thuật toán phổ biến nhất để duyệt và tìm kiếm trên đồ thị là Duyệt theo chiều rộng (Breadth-First Search - BFS) và Duyệt theo chiều sâu (Depth-First Search - DFS).

8.4.1. Duyệt theo chiều sâu DFS (Depth-first search)

8.4.1.1. Giới thiệu về mê cung (dẫn nhập)

Khám phá mê cung:

- Đồ thị mê cung.
- Đỉnh = ngã tư.
- Cạnh = lối đi.

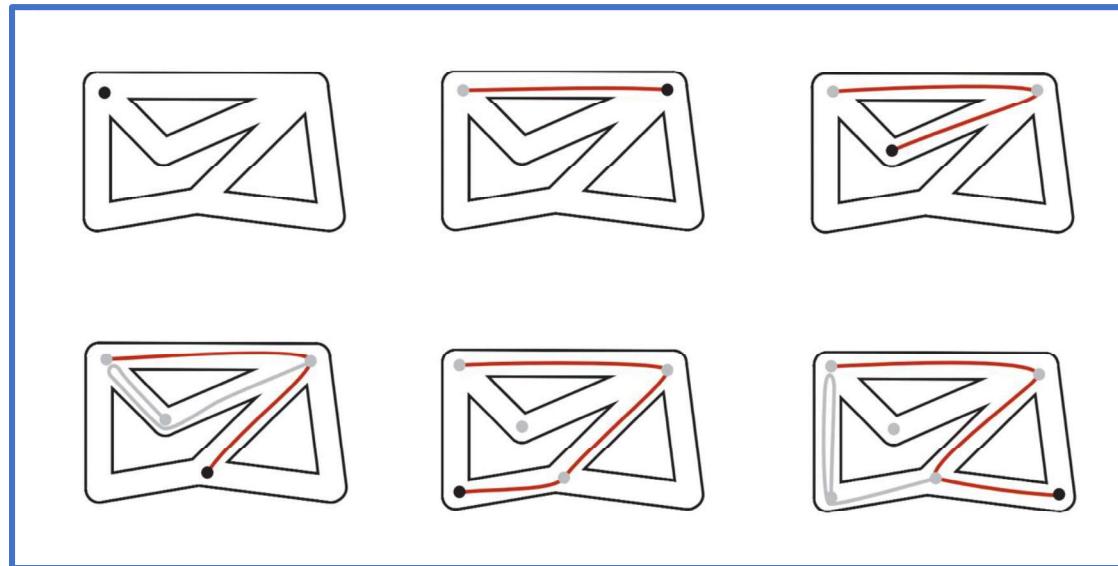


Mục tiêu: Khám phá mỗi ngã rẽ trong mê cung

8.4.1.2. Phương pháp khám phá mê cung Trémaux

Thuật toán

- Giải tháo một cuộn dây và thả nó phía sau khi bạn di chuyển qua các đoạn đường trong mê cung.
- Đánh dấu mỗi ngã rẽ và mỗi đoạn đường đã đi qua.
- Quay trở lại các bước đã đi khi không còn lựa chọn nào chưa được khám phá.

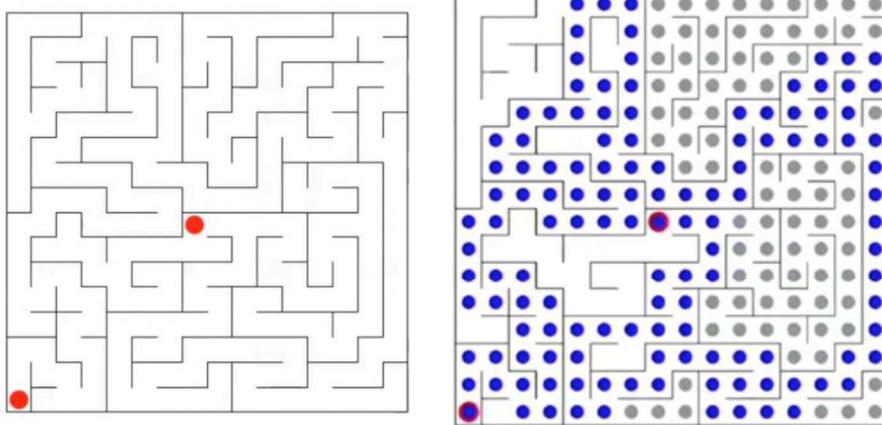


Lần sử dụng đầu tiên? Theseus đã nhập vào mê cung để tiêu diệt Minotaur quái dị. Ariadne chỉ dẫn Theseus sử dụng một cuộn dây để tìm đường ra về.



Claude Shannon (with Theseus mouse)

Khám phá mê cung



8.4.1.3. Phương pháp DFS (Depth-first search)

Mục tiêu : Tìm kiếm một cách có hệ thống thông qua một đồ thị.

Ý tưởng: Bắt chước việc khám phá mê cung.

Các ứng dụng điển hình.

- Tìm tất cả các đỉnh kết nối với một đỉnh nguồn đã cho.
- Tìm một đường đi giữa hai đỉnh.

Thách thức thiết kế. Làm thế nào để thực hiện?

Mẫu thiết kế cho xử lý đồ thị:

Mẫu thiết kế. Tách biệt kiểu dữ liệu đồ thị khỏi xử lý đồ thị.

- Tạo một đối tượng Đồ thị (Graph).
- Truyền đối tượng Đồ thị đến một quy trình xử lý đồ thị.
- Truy vấn quy trình xử lý đồ thị để lấy thông tin.

```
class Graph
{
private:
    int V; // số đỉnh
    vector<int>* adj; // danh sách kề
public:
    Graph(int V)
    {
        this->V = V;
        adj = new vector<int>[V];
    }
    void addEdge(int v, int w)
    {
        adj[v].push_back(w);
    }
}
```

```

int getV()
{
    return V;
}
vector<int> getAdj(int v)
{
    return adj[v];
}
};
```

```

class GraphProcessor
{
public:
    void process(Graph& G)
    {
        // xử lý đồ thị và truy vấn thông tin thông qua các phương thức của
        đối tượng Graph
        int V = G.getV();
        for (int i = 0; i < V; i++)
        {
            vector<int> adj = G.getAdj(i);
            cout << "Adjacency list of vertex " << i << ":" ;
            for (auto j = adj.begin(); j != adj.end(); ++j)
                cout << *j << " ";
            cout << endl;
        }
    }
};
```

8.4.1.4. Depth-first search demo

Để truy cập một đỉnh v:

- Đánh dấu đỉnh v là đã truy cập.
- Độ quy truy cập tất cả các đỉnh chưa được đánh dấu kề với v.

Mục tiêu. Tìm tất cả các đỉnh kết nối với đỉnh nguồn s (và một đường đi tương ứng).

Ý tưởng. Mô phỏng việc khám phá mê cung.

Thuật toán.

- Sử dụng đệ quy (quả bóng dây).
- Đánh dấu mỗi đỉnh đã được truy cập (và giữ thông tin về cạnh đã đi để truy cập đỉnh đó).
- Trả về (đi ngược lại các bước đã đi) khi không có tùy chọn chưa được truy cập.



Cấu trúc dữ liệu.

- boolean[] marked để đánh dấu các đỉnh đã được truy cập.
- int[] edgeTo để giữ cây các đường đi.

(edgeTo[w] == v) có nghĩa là cạnh v-w đã được đi qua để truy cập w lần đầu tiên.

```
class DFS
{
private:
    bool* marked;
    int* edgeTo;
    int s;
    void dfs(Graph G, int v)
    {
        marked[v] = true;
        vector<int> adj = G.getAdj(v);
        for (int w : adj) {
            if (!marked[w]) {
                edgeTo[w] = v;
                dfs(G, w);
            }
        }
    }
public:
    DFS(Graph G, int s)
    {
        marked = new bool[G.V];
        edgeTo = new int[G.V];
        this->s = s;
        dfs(G, s);
    }
    bool hasPathTo(int v)
    {
        return marked[v];
    }
    vector<int> pathTo(int v)
    {
        vector<int> path;
        if (!hasPathTo(v))
        {
            return path;
        }
        for (int x = v; x != s; x = edgeTo[x])
```

```
{  
    path.push_back(x);  
}  
path.push_back(s);  
return path;  
};  
};
```

//Hình minh họa

8.4.1.5. Ứng dụng (**ĐANG CẬP NHẬP...**)

- Ứng dụng để chuẩn bị cho một buổi hẹn hò

<https://xkcd.com/761/>

- Ứng dụng flood fill.

Flood fill là một kỹ thuật được sử dụng để tô màu các vùng được liên thông trong một ảnh số hoặc một bức tranh.

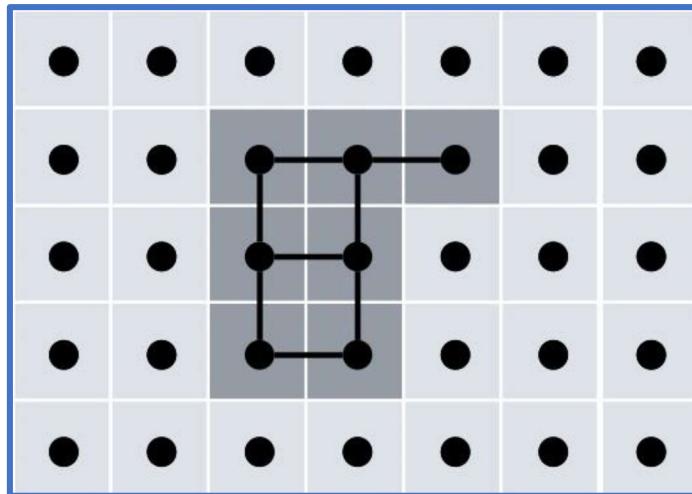
Thách thức : Flood fill (Photoshop magic wand)

Giả định : Hình ảnh có hàng triệu đến hàng tỷ điểm ảnh.



Giải pháp. Xây dựng đồ thị lưới.

- Đỉnh: điểm ảnh.
- Cạnh: giữa hai điểm ảnh xám kề nhau.
- Vùng: tất cả các điểm ảnh kết nối với điểm ảnh đã cho.



8.4.2. Duyệt theo chiều rộng BFS (Breadth-first search)

Lặp lại cho đến khi hàng đợi trống:

- Loại bỏ đỉnh v ra khỏi hàng đợi.
- Thêm vào hàng đợi tất cả các đỉnh kề chưa được đánh dấu với v và đánh dấu chúng.

Tìm kiếm theo chiều sâu : Đưa các đỉnh chưa được thăm vào **ngăn xếp**.

Tìm kiếm theo chiều rộng: Đưa các đỉnh chưa được thăm vào **hàng đợi**.

Đường đi ngắn nhất: Tìm đường đi từ s đến t sử dụng **ít cạnh nhất**.

BFS (from source vertex s):

Đưa s vào hàng đợi FIFO và đánh dấu s là đã thăm.

Lặp lại cho đến khi hàng đợi trống:

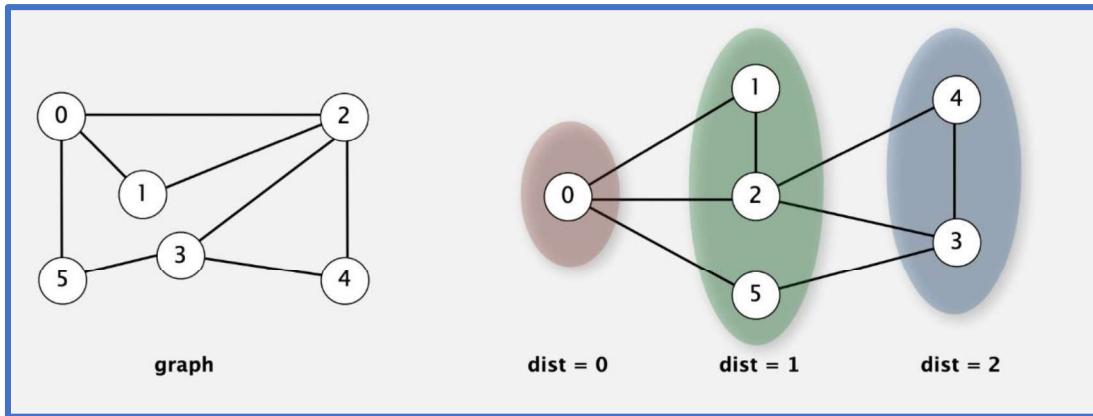
- Loại bỏ đỉnh v được thêm gần đây nhất
- Thêm mỗi neighbors chưa được thăm của v vào hàng đợi và đánh dấu chúng là đã thăm.

Tính Chất BFS:

Định lý. BFS tính toán đường đi ngắn nhất (ít cạnh nhất) từ s đến tất cả các đỉnh khác trong đồ thị với thời gian tỉ lệ với $E + V$.

Chứng minh. [đúng đắn] Hàng đợi luôn chứa không hoặc nhiều đỉnh ở khoảng cách k từ s, theo sau là không hoặc nhiều đỉnh ở khoảng cách k + 1.

Chứng minh. [thời gian chạy] Mỗi đỉnh kết nối với s được duyệt một lần.



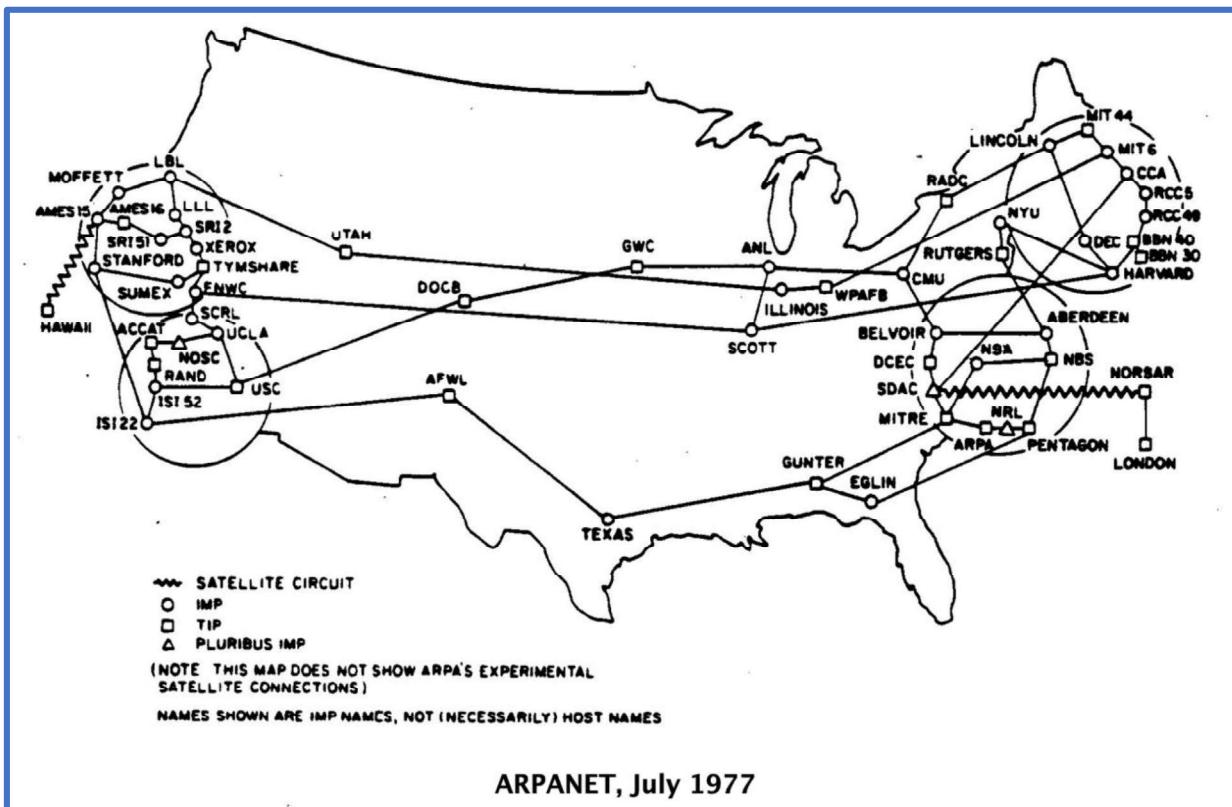
BFS C++

```
class BFS
{
private:
    bool* marked;
    int* edgeTo;
    int s;

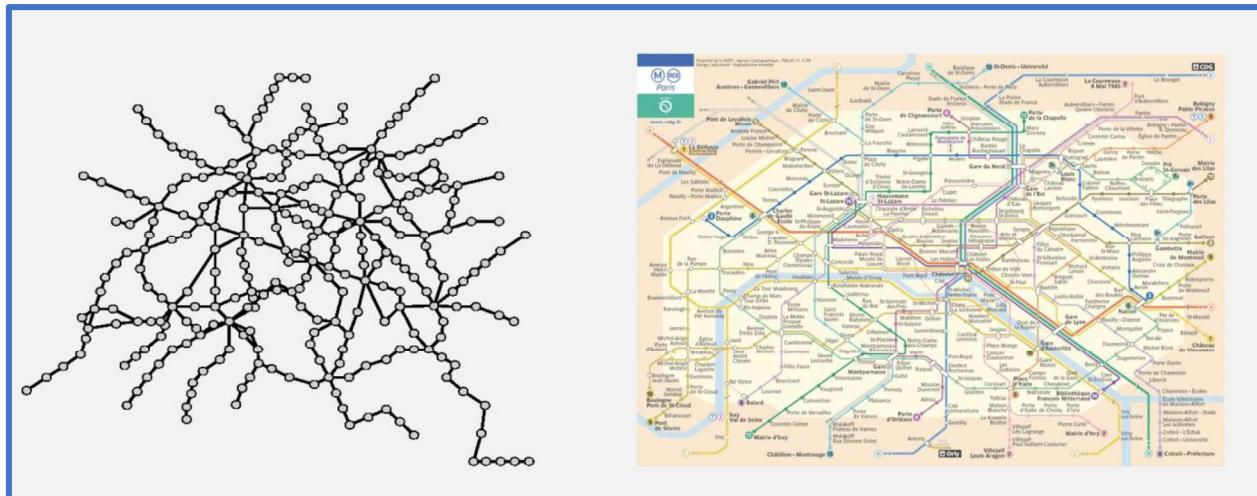
public:
    BFS(Graph G, int s)
    {
        marked = new bool[G.V];
        edgeTo = new int[G.V];
        this->s = s;
        queue<int> q;
        q.push(s);
        marked[s] = true;
        while (!q.empty())
        {
            int v = q.front();
            q.pop();
            vector<int> adj = G.getAdj(v);
            for (int w : adj)
            {
                if (!marked[w])
                {
                    q.push(w);
                    marked[w] = true;
                    edgeTo[w] = v;
                }
            }
        }
    }
}
```

```
    }  
};  
};  
};  
};
```

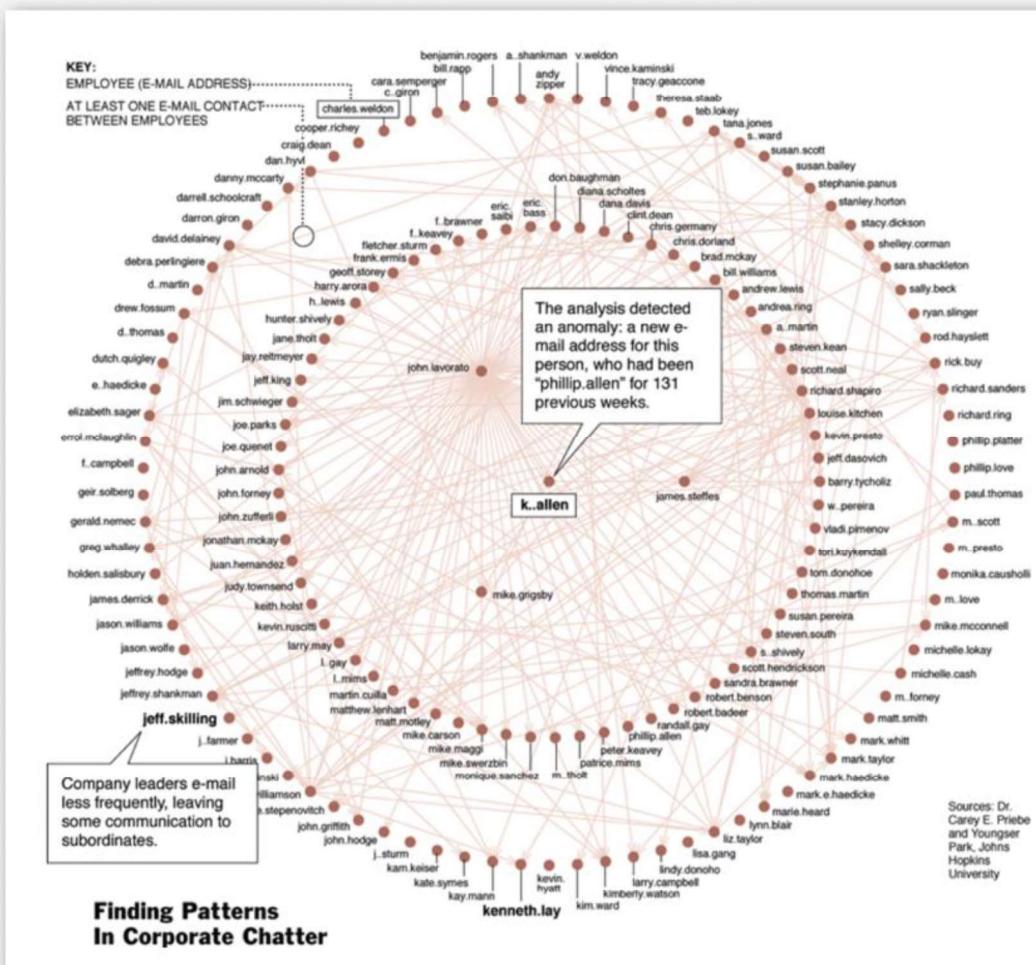
3.1.5 Ứng dụng phổ biến của BFS là trong các hệ thống định tuyến (routing) mạng.



Một số hình ảnh về đồ thị



One week of Enron emails



8.5. Ứng dụng đồ thị (Bài toán tô màu,...)

(Trích đề thi thử **Cấu trúc dữ liệu và giải thuật – UIT – HK2** năm học **2021-2022**, và phát triển thêm)

Cho bài toán “Tô màu bản đồ” được đặt ra như sau: Có một bản đồ các quốc gia trên thế giới, ta muốn tô màu các quốc gia này sao cho hai nước có cùng ranh giới được tô khác màu nhau. Yêu cầu tìm cách tô sao cho số màu sử dụng là ít nhất. Bài toán có thể được mô hình hóa thành một bài toán trên đồ thị, khi đó mỗi nước trên bản đồ là một đỉnh của đồ thị, hai nước láng giềng tương ứng với hai đỉnh kề nhau được nối với nhau bằng một cạnh, bài toán trở thành: tô màu các đỉnh của đồ thị sao cho mỗi đỉnh chỉ được tô một màu, hai đỉnh kề nhau có màu khác nhau và số màu sử dụng là ít nhất. Giả sử cho thông tin đầu vào của bài toán được nhập vào chương trình như sau:

Ví dụ Input	Giải thích
15 Viet_Nam Lao Viet_Nam Trung_Quoc Thai_Lan Lao ... Campuchia Thai_Lan	<ul style="list-style-type: none"> - Dòng đầu tiên chứa một số e là số cạnh của đồ thị - e dòng tiếp theo, mỗi dòng chứa 2 chuỗi u và l, thể hiện thông tin có một cạnh nối từ đỉnh u sang đỉnh l trong đồ thị <p>Lưu ý: Không biết trước số đỉnh và danh sách các đỉnh</p>

Hãy thực hiện các yêu cầu sau:

- Xây dựng cấu trúc dữ liệu thích hợp để biểu diễn đồ thị nhằm lưu trữ các thông tin cần thiết trên bản đồ.
- Viết hàm nhập đồ thị (bằng cách nhập số cạnh và danh sách các cạnh như ví dụ ở trên) và lưu trữ thông tin của đồ thị vào cấu trúc dữ liệu đã đề xuất ở câu a.
- Xuất ra tên tất cả các nước có trong danh sách cạnh đã nhập, kèm với các nước láng giềng của nó.

Lời giải:

- Vì ta chưa biết trước số cạnh của đồ thị, có thể sử dụng cấu trúc vector để lưu trữ. Ta có thể định nghĩa cấu trúc 1 đỉnh của đồ thị - tương ứng với một quốc gia như sau:

```

struct Country {
    string name;
    vector<string> neighbors;
};

vector<Country> listCountry;

```

Tuy nhiên, việc quản lý danh sách các đỉnh – quốc gia bằng vector kém hiệu quả. Bởi lúc ta thêm 1 cạnh vào, thì có khả năng đỉnh của cạnh vừa thêm đã tồn tại trong danh sách. Và việc nhập/ xuất các đỉnh kề với đỉnh hiện tại cũng kém hiệu quả và cồng kềnh.



TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN – ĐHQG-HCM BẢN HỌC TẬP CÔNG NGHỆ PHẦN MỀM

Ta sẽ nghĩ đến việc sử dụng một cấu trúc tối ưu hơn struct, một cấu trúc có thể dùng string là key để truy xuất đến danh sách các đỉnh kề. Và value chính là danh sách các đỉnh kề. Cấu trúc đó chính là map.

Chúng ta sẽ dùng cấu trúc map, chỉ với một dòng đơn giản như sau:

```
map<string, set<string>> listCountry;
```

Giá trị đầu tiên của map listCountry là tương ứng với 1 đỉnh – 1 quốc gia và dữ liệu đại diện cho quốc gia đó là tên với kiểu dữ liệu string.

Giá trị thứ hai (second) của map listCountry đó chính là danh sách các đỉnh kề - các quốc gia láng giềng và dữ liệu đại diện cũng là tên với kiểu dữ liệu string. Ở đây, chúng ta sẽ dùng set<string> vì set được xây dựng dựa trên cây nhị phân tìm kiếm, nên lúc ta thêm đỉnh kề vào tập đỉnh kề sẽ không cần kiểm tra có lặp hay không, vì set chỉ lưu trữ các giá trị khác nhau, mỗi giá trị chỉ xuất hiện trong set đúng 1 lần, đồng thời tốc độ truy xuất cũng sẽ tốt hơn.

b.

Khi ta đã xây dựng được cấu trúc dữ liệu như trên, thì việc nhập dữ liệu hoàn toàn đơn giản như sau:

```
for(int j=0; j<e; j++){
    string u, i;
    cin>>u>>i;
    listCountry[u].insert(i);
    listCountry[i].insert(u);
}
```

c.

Để xuất tên tất cả các quốc gia trong tập cạnh nhập ở đề bài, ta chỉ cần duyệt qua map listCountry. Với mỗi listCountry thứ i, thì listCountry[i].first sẽ là giá trị của đỉnh – quốc gia ta hiện tại, và listCountry[i].second sẽ là set các đỉnh kề của đỉnh hiện tại – quốc gia láng giềng.

Và ta tiếp tục tiến hành duyệt toàn bộ set listCountry[i].second để lấy ra các quốc gia láng giềng.

Để gọn nhẹ, ta sẽ sử dụng con trỏ auto để truy xuất map và set.

```
for(auto x: listCountry){
    cout<<x.first<<"'s neighbors: ";
    for(auto y: x.second){
        cout<<y<<" ";
    }
    cout<<endl;
}
```

Chương 9. Bảng băm (Hash table)

9.1. Giới thiệu cơ bản về bảng băm

9.1.1. Sơ lược về lịch sử hình thành

Vào những năm 50 của thế kỷ 20, thuật ngữ bảng băm bắt đầu xuất hiện và phát triển một cách nhanh chóng. Vào năm 1973, cuốn sách "*The Art of Computer Programming*" do nhà khoa học nổi tiếng Donald Knuth viết giới thiệu về các khái niệm và thuật toán liên quan đến bảng băm, đã đánh dấu một trong những bước phát triển mạnh mẽ của bảng băm. Từ khi xuất hiện đến nay, bảng băm đã trở thành một công cụ quan trọng và phổ biến trong việc lưu trữ và tìm kiếm dữ liệu nhanh chóng và hiệu quả.

9.1.2. Các khái niệm

- Cơ sở dữ liệu:** Bảng băm được sử dụng trong các hệ quản trị cơ sở dữ liệu để tìm kiếm nhanh dữ liệu.
- Tìm kiếm và tra cứu:** Bảng băm được sử dụng để tìm kiếm nhanh các phần tử trong tập dữ liệu lớn. Ví dụ, trong các công cụ tìm kiếm web, bảng băm được sử dụng để lưu trữ và tìm kiếm các từ khóa.
- Xác thực và bảo mật:** Mật khẩu người dùng thường được lưu trữ dưới dạng giá trị băm trong cơ sở dữ liệu để đảm bảo tính an toàn và riêng tư.
- Mã hóa:** Bảng băm được sử dụng trong các thuật toán mã hóa để tạo chữ ký số và băm các thông tin nhạy cảm. Ví dụ, thuật toán SHA-256 sử dụng bảng băm để tạo giá trị băm 256-bit cho dữ liệu đầu vào.
- Tối ưu hóa và tìm kiếm trong đồ thị:** Các thuật toán như Dijkstra và A* sử dụng bảng băm để lưu trữ và truy xuất các đỉnh và cạnh trong đồ thị.
- Một số thuật ngữ thường dùng
- Key (khóa):** Là giá trị duy nhất được sử dụng để xác định vị trí lưu trữ và tìm kiếm trong bảng băm.
- Hash table (bảng băm):** Thường có dạng mảng có kích thước cố định, trong đó mỗi phần tử của mảng là một bucket (ngăn) hoặc một danh sách liên kết đơn, ký hiệu HT, có N vị trí được đánh chỉ mục từ 0 đến N-1, N là kích thước của bảng băm.
- Hash function (hàm băm):** Là một hàm được sử dụng để chuyển đổi khóa thành một chỉ số trong mảng lưu trữ của bảng băm. Hàm băm cần phải có tính chất phân tán tốt để tránh xung đột khóa và giúp phân tán dữ liệu đều trong bảng.
- Collision (đụng độ):** Xảy ra khi hai khóa được ánh xạ đến cùng một vị trí trong mảng. Điều này có thể xảy ra do hàm băm không phân tán tốt hoặc do kích thước mảng không đủ lớn để chứa tất cả các khóa.
- Collision resolution (giải quyết đụng độ):** Là quá trình giải quyết xung đột khi hai khóa ánh xạ đến cùng một vị trí. Có nhiều phương pháp để giải quyết xung đột, bao

gồm sử dụng danh sách liên kết (chaining), băm kép (double hashing) và băm mở (open addressing).

- **Load (tải):** Là tỷ lệ giữa số lượng phần tử hiện có trong bảng băm và kích thước của mảng. Tải cao có thể dẫn đến hiện tượng xung đột tăng và làm giảm hiệu suất của bảng băm.
- **Dynamic hashing (Băm động):** Là kỹ thuật điều chỉnh kích thước của mảng dựa trên tải của bảng băm, để đảm bảo rằng không gian lưu trữ được sử dụng hiệu quả và tránh tình trạng quá tải hoặc lãng phí.

9.2. Hàm băm

9.2.1. Định nghĩa

- Hàm băm là một ánh xạ biến đổi từ các khóa thành các địa chỉ trong bảng.
- Trong điều kiện lý tưởng, một hàm băm tốt là một hàm băm có thể phân bố đều trên miền giá trị của địa chỉ, tuy nhiên điều này là rất khó xảy ra.
- Hàm băm hoàn hảo: Với một tập hợp các phần tử, hàm băm ánh xạ từng khóa vào 1 địa chỉ duy nhất trong bảng được gọi là hàm băm hoàn hảo.
- Một số cách chọn hàm băm tốt:
 - Hàm băm phải đơn giản để tính toán.
 - Số lần xử lý va chạm phải ít hơn khi đặt bản ghi vào bảng băm. Tốt nhất là không nên để đúng độ (Xử lý va chạm xem ở mục 1.3)
 - Hàm băm tạo ra các khóa nên được phân bố đều trên các địa chỉ trong bảng.

9.2.2. Một số phương pháp sử dụng hàm băm

Giả sử gọi N là các phần tử được chứa trong bộ nhớ (thường N là số nguyên tố), hàm băm sẽ biến đổi các khóa (thường là số nguyên hoặc là các chuỗi ký tự ngắn) thành các đoạn số nguyên trong khoảng từ 0 đến $N - 1$.

Giả sử các khóa là số nguyên, hàm băm $H(k)$ là:

$H(k) = k \bmod N$

với k là khóa, $H(k)$ là hàm lấy dư của k chia cho N

- **Ví dụ 1:**

$k = 256, N = 17$ (số nguyên tố)

$H(256) = 256 \bmod 17 = 1$

- **Ví dụ 2:**

$k = 26, N = 17$

$H(26) = 26 \bmod 17 = 9$

Tuy nhiên với $k = 60, N = 17$ thì:

$H(60) = 60 \bmod 17 = 9$

Ở Ví dụ 2, ta thấy 2 khóa cùng 1 địa chỉ, điều này xảy ra và chạm giữa các khóa với nhau.

Vậy làm cách nào để xử lý chúng?

9.3. Giải quyết – xử lý va chạm (đụng độ)

9.3.1. Khái niệm đụng độ

- Đụng độ là hiện tượng các khóa khác nhau nhưng khi bấm cho ra cùng địa chỉ nhau.
- Khi $\text{key1} \# \text{key2}$ mà $f(\text{key1}) = f(\text{key2})$, chúng ta nói nút có khóa key1 đụng độ với nút có khóa key2.
- Thực tế, người ta giải quyết đụng độ theo 2 phương pháp:
 - Phương pháp nối kết
 - Phương pháp băm lại

9.3.2. Phương pháp nối kết

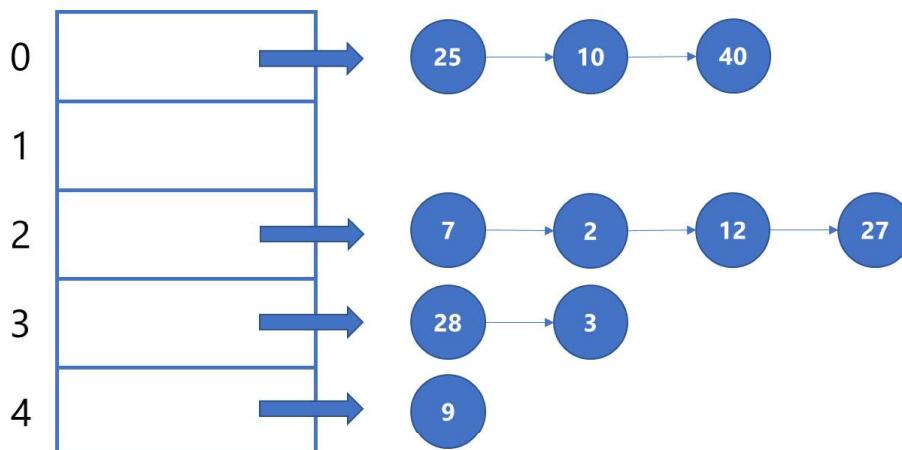
9.3.2.1. Phương pháp nối kết trực tiếp

***Hướng giải quyết:** Các nút bị băm cùng địa chỉ (đụng độ/ xung đột) sẽ được gom thành một danh sách liên kết

Ví dụ 1: Cho bảng băm có $M = 5$ là số vị trí trống chứa khóa như hình bên dưới.

Hàm băm: $h(\text{key}) = (\text{key \% } M)$, xử lý đụng độ bằng phương pháp nối kết trực tiếp.

Lần lượt thêm các khóa 25, 7, 2, 10, 40, 12, 27, 9, 28, 3 vào bảng. Ta sẽ được kết quả như sau:



9.3.2.2. Phương pháp nối kết hợp nhất

- Bảng băm trong trường hợp này được cài đặt bằng danh sách liên kết dùng mảng, có M nút. Các nút bị xung đột địa chỉ được kết nối với nhau qua một danh sách liên kết.

- Mỗi nút của bảng băm là một mảnh tin có 2 trường:
 - Trường key: chứa các khóa của nút
 - Trường next: con trỏ trỏ đến nút kế tiếp nếu có xung đột.
- Khi khởi tạo bảng băm thì tất cả trường key được gán bằng NULL, tất cả trường next được gán bằng -1.
- Khi thêm một nút có khóa key vào bảng băm, hàm băm $f(key)$ sẽ xác định địa chỉ i trong khoảng từ 0 đến $M-1$, nếu chưa bị xung đột thì thêm nút mới vào địa chỉ này. Nếu xung đột thì nút mới được cấp phát là nút trống phía cuối mảng. Và địa chỉ của nút trống cuối mảng này sẽ được lưu vào trường next của nút vừa bị xung đột.
- Khi tìm một nút có khóa key trong bảng băm, hàm băm $f(key)$ sẽ xác định địa chỉ i trong khoảng từ 0 đến $M-1$, tìm nút khóa key trong danh sách liên kết xuất phát từ địa chỉ i .

Ví dụ 2: Ví dụ cho bảng băm có $M = 6$ như hình bên dưới.

- Lần lượt thêm các khóa 12, 18, 24, 1, 49 vào bảng.
- Sử dụng hàm băm: $h(key) = (\text{key \% } M)$
- Giải quyết đụng độ bằng phương pháp nối kết hợp nhất.

Ta sẽ được kết quả:

	Key	Next
0	NULL	-1
1	NULL	-1
2	NULL	-1
3	NULL	-1
4	NULL	-1
5	NULL	-1



	Key	Next
0		
1		
2		
3		
4		

	Key	Next
0	12	5
1	1	3
2	NULL	-1
3	49	-1
4	24	-1
5	18	4

9.3.3. Phương pháp băm lại.

9.3.3.1. Phương pháp dò tuyến tính

- Bảng băm trong trường hợp này được cài đặt bằng một danh sách kề có M nút, mỗi nút của bảng băm là một mẫu tin có trường key để chứa khóa.
- Khi khởi tạo bảng băm thì tất cả trường key được gán bằng NULL.
- Khi thêm nút có khóa key vào bảng, hàm băm f(key) sẽ xác định địa chỉ i trong khoảng từ 0 đến M – 1.
- Nếu chưa bị đụng độ thì thêm nút mới vào địa chỉ này.
- Nếu xảy ra đụng độ thì tiến hành duyệt tuyến tính, hàm băm lại lần 1 – f1 sẽ xét địa chỉ kế tiếp, nếu lại bị xung đột thì hàm băm lại lần 2 – f2 sẽ xét địa chỉ kế tiếp, quá trình duyệt cứ tiếp tục cho đến khi nào tìm được địa chỉ trống và thêm nút vào địa chỉ này.
- Khi tìm một nút có khoá key vào bảng băm, hàm băm f(key) sẽ xác định địa chỉ i trong khoảng từ 0 đến M-1, tìm nút khoá key trong khối đặt chứa các nút xuất phát từ địa chỉ i.
- Hàm băm lại của phương pháp dò tuyến tính là truy xuất địa chỉ kế tiếp. Hàm băm lại lần I được biểu diễn bằng công thức sau:

$$f(key) = (f(key) + i) \% M$$

***Lưu ý:** địa chỉ dò tìm kế tiếp là địa chỉ 0 nếu đã dò đến cuối bảng.

***Nhận xét:**

- Bảng băm này chỉ tối ưu khi băm đều, ít xảy ra đụng độ liên tục, tốc độ truy xuất lúc này có bậc O(1).
- Trường hợp xấu nhất là bảng băm không đều, nếu liên tục xảy ra đụng độ thì các khóa sẽ được phân bố nối tiếp nhau theo thứ tự từ trên xuống, khi tiến hành băm để tìm vị trí thì số lần băm tương ứng với việc phải duyệt toàn bộ bảng, dẫn đến tốc độ truy xuất lúc này có bậc O(n).

- **Ví dụ 3:** Cho bảng băm ở trạng thái vừa khởi tạo có **M = 5** nút, lần lượt thêm các khóa **27, 32, 46, 14** vào bảng.

Hàm băm: $f(key) = (\text{key \% } M)$

Xử lý đụng độ bằng phương pháp dò tuyến tính.

Khởi tạo:

Key
NULL

Thêm khóa 27:

Key		
0	NULL	
1	NULL	
2	27	
3	NULL	
4	NULL	

Thêm khóa 32:

- $f(32) = (32\%5) = 2 \Rightarrow$ Xảy ra đụng độ. Ta tiến hành băm lại lần 1:
- $f1(32) = (32+1)\%5 = 3 \Rightarrow$ Vị trí trống, điền khóa 32 vào vị trí này.

Key		
0	NULL	
1	NULL	
2	27	
3	32	
4	NULL	

Thêm khóa 46:

- $f(46) = (46\%5) = 1 \Rightarrow$ Vị trí trống, điền khóa 46 vào vị trí này.

Key		
0	NULL	
1	46	
2	27	
3	32	
4	NULL	

Thêm khóa 51:

- $f(51) = (51 \% 5) = 1 \Rightarrow$ Xảy ra đụng độ, ta tiến hành băm lại lần 1:
- $f1(51) = (51+1) \% 5 = 2 \Rightarrow$ Xảy ra đụng độ, ta tiến hành băm lại lần 2:
- $f2(51) = (51+2) \% 5 = 3 \Rightarrow$ Xảy ra đụng độ, ta tiến hành băm lại lần 3.

- $f_3(51) = (51+3) \% 5 = 4 \Rightarrow$ Vị trí trống. Ta điền khóa 51 vào vị trí này.

Key	
0	NULL
1	46
2	27
3	32
4	51

Thêm khóa 14:

- $f(14)=14\%5 = 4 \Rightarrow$ Xảy ra đụng độ, ta tiến hành băm lại lần 1:
- $f_1(14) = (14+1) \% 5 = 0 \Rightarrow$ Vị trí trống, ta tiến hành điền khóa vào vị trí này.

Key	
0	14
1	46
2	27
3	32
4	51

9.3.3.2. Phương pháp dò bậc hai

- Hạn chế của bảng băm dùng phương pháp dò tuyến tính bị hạn chế do rải các nút không đều, nếu liên tục xảy ra đụng độ thì các khóa sẽ được phân bố nối tiếp nhau theo thứ tự từ trên xuống. Dẫn đến việc truy xuất chậm.
- Phương pháp dò bậc hai sẽ rải các nút đều hơn.
- Bảng băm sử dụng phương pháp dò bậc hai có cấu trúc tương tự với bảng băm sử dụng phương pháp dò tuyến tính, tuy nhiên hàm băm lại khi xảy ra đụng độ lần thứ i sẽ là:
 - $f_i(key) = (f(key) + i^2) \% M$ với $f(key)$ là hàm băm chính của bảng băm.
 - Khi sử dụng phương pháp băm bậc 2 nên chọn số địa chỉ M là số nguyên tố
 - Sẽ có trường hợp bảng băm còn trống vị trí để chứa khóa, nhưng khi băm sẽ không tìm ra được vị trí để điền khóa vào bảng băm. Vì phương pháp dò bậc hai khi áp dụng với một số số nguyên M sẽ không duyệt được hết các vị trí của bảng băm.

- **Ví dụ 4:** Cho bảng băm ở trạng thái vừa khởi tạo có **M = 11** nút, lần lượt thêm các khóa **31, 19, 2, 13, 25, 24, 21, 9** vào bảng.

Hàm băm: $f(key) = (\text{key} \% M)$

Xử lý đụng độ bằng phương pháp dò bậc hai. Hàm băm lại khi xảy ra đụng độ lần thứ i sẽ là: $f_i(\text{key}) = (f(\text{key}) + i^2) \% M$ với $f(\text{key})$ là hàm băm chính của bảng băm.

Khởi tạo:

Key	
0	NULL
1	NULL
2	NULL
3	NULL
4	NULL
5	NULL
6	NULL
7	NULL
8	NULL
9	NULL
10	NULL

Thêm khóa 31:

- $f(31) = 31 \% 11 = 9 \Rightarrow$ Vị trí trống, ta tiến hành điền khóa 31 vào vị trí này.

Key	
0	NULL
1	NULL
2	NULL
3	NULL
4	NULL
5	NULL
6	NULL
7	NULL
8	NULL
9	31
10	NULL

Thêm khóa 19:

- $f(19) = 19 \% 11 = 8 \Rightarrow$ Vị trí trống, ta tiến hành điền khóa 19 vào vị trí này.

Key	
0	NULL
1	NULL
2	NULL
3	NULL
4	NULL
5	NULL
6	NULL
7	NULL
8	19
9	31
10	NULL

Thêm khóa 2:

- $f(2) = 2 \% 11 = 2 \Rightarrow$ Vị trí trống, ta tiến hành điền khóa 2 vào vị trí này.

Key	
0	NULL
1	NULL
2	2
3	NULL
4	NULL
5	NULL
6	NULL
7	NULL
8	19
9	31
10	NULL

Thêm khóa 13:

- $f(13) = 13 \% 11 = 2 \Rightarrow$ Xảy ra đụng độ, ta tiến hành băm lại lần 1:
- $f_1(13) = (f(13) + 1^2) \% 11 = 3 \Rightarrow$ Vị trí trống, ta tiến hành điền khóa 13 vào vị trí này.

	Key
0	NULL
1	NULL
2	2
3	13
4	NULL
5	NULL
6	NULL
7	NULL
8	19
9	31
10	NULL

Thêm khóa 25:

- $f(25) = 25 \% 11 = 3 \Rightarrow$ Xảy ra đụng độ, ta tiến hành băm lại lần 1:
- $f_1(25) = (f(25) + 1^2) \% 11 = 4 \Rightarrow$ Vị trí trống, ta tiến hành điền khóa 25 vào vị trí này.

	Key
0	NULL
1	NULL
2	2
3	13
4	25
5	NULL
6	NULL
7	NULL
8	19
9	31
10	NULL

Thêm khóa 24:

- $f(24) = 24 \% 11 = 2 \Rightarrow$ Xảy ra đụng độ, ta tiến hành băm lại lần 1:
- $f_1(24) = (f(24) + 1^2) \% 11 = 3 \Rightarrow$ Xảy ra đụng độ, ta tiến hành băm lại lần 2:
- $f_2(24) = (f(24) + 2^2) \% 11 = 6 \Rightarrow$ Vị trí trống, ta tiến hành điền khóa 24 vào vị trí này.

	Key
0	NULL
1	NULL
2	2
3	13
4	25
5	NULL
6	24
7	NULL
8	19
9	31
10	NULL

Thêm khóa 21:

- $f(21) = 21 \% 11 = 10 \Rightarrow$ Vị trí trống, ta tiến hành điền khóa 21 vào vị trí này.

	Key
0	NULL
1	NULL
2	2
3	13
4	25
5	NULL
6	24
7	NULL
8	19
9	31
10	21

Thêm khóa 9:

- $f(9) = 9 \% 11 = 9 \Rightarrow$ Xảy ra đụng độ, ta tiến hành băm lại lần 1:
- $f_1(9) = (f(9) + 1^2) \% 11 = 10 \Rightarrow$ Xảy ra đụng độ, ta tiến hành băm lại lần 2:
- $f_2(9) = (f(9) + 2^2) \% 11 = 2 \Rightarrow$ Xảy ra đụng độ, ta tiến hành băm lại lần 3:
- $f_3(9) = (f(9) + 3^2) \% 11 = 7 \Rightarrow$ Vị trí trống, ta tiến hành điền khóa 7 vào vị trí này.

	Key
0	NULL
1	NULL
2	2
3	13
4	25
5	NULL
6	24
7	7
8	19
9	31
10	21

9.3.3.3. Phương pháp băm kép

Bảng băm này dùng hai hàm băm khác nhau với mục đích rải đều các phần tử trên bảng băm. Chúng ta có thể sử dụng hai hàm băm bất kỳ, ví dụ chọn hai hàm băm như sau:

$$h1(key) = key \% M$$

$$h2(key) = (M-2) - key \% (M-2)$$

- Băm lần thứ i: $h(key, i) = (h1(key) + i * h2(key)) \% M$
- Bảng băm trong trường hợp này được cài đặt bằng danh sách kề có M phần tử tương tự như ở phương pháp dò tuyến tính và dò bậc hai.
- Khi khởi tạo bảng băm, tất cả trường key được gán bằng NULL
- Khi thêm phần tử có khóa key vào bảng băm, đặt $i = f1(key)$ và $j = f2(key)$ (i và j trong khoảng từ 0 đến $M - 1$):
 - Nếu chưa bị xung đột thì thêm phần tử mới tại địa chỉ i này.
 - Nếu bị xung đột thì hàm băm lại lần 1, $f1$ sẽ xét địa chỉ mới $i+j$, nếu lại bị xung đột thì hàm băm lại lần 2, $f2$ sẽ xét địa chỉ $i+2j$,... quá trình cứ thế cho đến khi tìm được địa chỉ trống và thêm phần tử và địa chỉ này.

- **Ví dụ 5:** Cho bảng băm có $M = 5$ là số vị trí trống chứa khóa như hình bên dưới.

Hàm băm: $h1(key) = (key \% M)$, $h2(key) = (M-2) - key \% (M-2)$ xử lý đụng độ bằng phương pháp băm kép.

Lần lượt thêm các khóa **27, 32, 46, 14** vào bảng.

Khởi tạo:

Key		
0	NULL	
1	NULL	
2	NULL	
3	NULL	
4	NULL	

Thêm khóa 27:

Key		
0	NULL	
1	NULL	
2	27	
3	NULL	
4	NULL	

Thêm khóa 32:

- $f_1(32) = (32\%5) = 2 \Rightarrow$ Xảy ra đụng độ. Ta tiến hành băm lại lần 1:
- $f_2(32) = (5-2) - 32 \% (5-2) = 1$
- $(f_1(32) + f_2(32)) \% 5 = (1 + 2) \% 5 = 3 \Rightarrow$ Vị trí trống, điền khóa 32 vào vị trí này.

Key		
0	NULL	
1	NULL	
2	27	
3	32	
4	NULL	

Thêm khóa 46:

- $f_1(46) = (46\%5) = 1 \Rightarrow$ Vị trí trống, điền khóa 46 vào vị trí này.

Key		
0	NULL	
1	46	
2	27	
3	32	
4	NULL	

Thêm khóa 51:

- $f1(51) = (51 \% 5) = 1 \Rightarrow$ Xảy ra đụng độ, ta tiến hành băm lại lần 1:
- $f2(51) = (5 - 2) - 51 \% (5 - 2) = 3$
- $(f1(51) + f2(51)) \% 5 = 4 \% 5 = 4 \Rightarrow$ Vị trí trống. Ta điền khóa 51 vào vị trí này.

Key		
0	NULL	
1	46	
2	27	
3	32	
4	51	

Thêm khóa 14:

- $f1(14)=14\%5 = 4 \Rightarrow$ Xảy ra đụng độ, ta tiến hành băm lại lần 1:
- $f2(14) = (5 - 2) - 14\%(5 - 2) = 1$
- $(f1(14) + f2(14)) \% 5 = (4 + 1) \% 5 = 0$

\Rightarrow Vị trí trống, ta tiến hành điền khóa vào vị trí này.

Key		
0	14	
1	46	
2	27	
3	32	
4	51	

Bài tập



Câu 1: Đề thi CTDLGT 2018 – 2019)

Cho bảng băm A kích thước 13 phần tử và tập khóa $K = \{10, 26, 52, 76, 13, 8, 3, 33, 60, 42\}$, ta cần nạp các giá trị khóa K vào bảng băm A sử dụng hàm băm $H(K) = K \% 13$. Hãy vẽ bảng băm khi **thêm từng khóa K vào bảng A**, trong trường hợp xảy ra đụng độ, sử dụng phương pháp dò tuyến tính để giải quyết đụng độ

Câu 2: Cho một cơ sở dữ liệu lưu trữ tài khoản và mật khẩu của các người dùng.

Hãy viết chương trình nhập vào tài khoản và mật khẩu của từng user và xuất ra mật khẩu theo yêu cầu. (**sử dụng bảng băm**)

Input:

Dòng đầu tiên là m (là số lượng người dùng có trong cơ sở dữ liệu) và n (là số lượng tài khoản nhập vào)

m dòng tiếp theo: mỗi dòng có 2 chuỗi là Tai_Khoan và Mat_Khau

n dòng cuối: mỗi dòng là một chuỗi tên tài khoản muốn lấy mật khẩu

Out put:

Gồm n dòng: Mỗi dòng là mật khẩu tương ứng tên tài khoản đã nhập

NOTE:

+ Trong cơ sở dữ liệu nếu nhập 2 tài khoản giống nhau, sẽ lấy mật khẩu của tài khoản sau làm mật khẩu.

+ Nếu tài khoản tồn tại thì xuất ra mật khẩu tương ứng, tài khoản không tồn tại thì xuất ra câu thông báo “Tai khoan khong ton tai!”.

Lời giải tham khảo:

Câu 1:

Hàm băm: $h(key) = key \% 13$

Ta có hàm băm lại của phương pháp dò tuyến tính: $f(key, i) = (h(key) + i) \% 13$

	Khởi tạo	Thêm 10	Thêm 26	Thêm 52	Thêm 76	Thêm 13	Thêm 8	Thêm 3	Thêm 33	Thêm 60	Thêm 42
0	NULl		26	26	26	26	26	26	26	26	26
1	NULl			52	52	52	52	52	52	52	52
2	NULl					13	13	13	13	13	13
3	NULl							3	3	3	3
4	NULl										42
5	NULl										
6	NULl										
7	NULl								33	33	33
8	NULl						8	8	8	8	8
9	NULl									60	60
10	NULl	10	10	10	10	10	10	10	10	10	10
11	NULl				76	76	76	76	76	76	76
12	NULl										

Chi tiết từng bước thêm:

- Thêm khóa 10: $h(10) = 10 \% 13 = 10$. Vị trí trống => thêm khóa 10 vào vị trí này.
- Thêm khóa 26: $h(26) = 26 \% 13 = 0$. Vị trí trống => thêm khóa 26 vào vị trí này.
- Thêm khóa 52: $h(52) = 52 \% 13 = 10$. Xảy ra đụng độ => tiến hành băm lại lần 1.
 $i = 1, f(52,1) = (f(52) + 1) \% 13 = 1$. Vị trí trống => Thêm khóa 52 vào vị trí này.
- Thêm khóa 76: $h(76) = 76 \% 13 = 11$. Vị trí trống => thêm khóa 76 vào vị trí này.
- Thêm khóa 13: $h(13) = 13 \% 13 = 0$. Xảy ra đụng độ => tiến hành băm lại lần 1.
 $i = 1, f(13,1) = (f(13) + 1) \% 13 = 1$. Xảy ra đụng độ => tiến hành băm lại lần 2.
 $i = 2, f(13,2) = (f(13) + 2) \% 13 = 2$. Vị trí trống => thêm khóa 13 vào vị trí này
- Thêm khóa 8: $h(8) = 8 \% 13 = 8$. Vị trí trống => thêm khóa 8 vào vị trí này.
- Thêm khóa 3: $h(3) = 3 \% 13 = 3$. Vị trí trống => thêm khóa 3 vào vị trí này.
- Thêm khóa 33: $h(33) = 33 \% 13 = 7$. Vị trí trống => thêm khóa 33 vào vị trí này.
- Thêm khóa 60: $h(60) = 60 \% 13 = 8$. Xảy ra đụng độ => tiến hành băm lại lần 1.
 $i = 1, f(60,1) = (f(60) + 1) \% 13 = 9$. Vị trí trống => Thêm khóa 60 vào vị trí này.
- Thêm khóa 42: $h(42) = 42 \% 13 = 3$. Xảy ra đụng độ => tiến hành băm lại lần 1.
 $i = 1, f(42,1) = (f(42) + 1) \% 13 = 4$. Vị trí trống => Thêm khóa 42 vào vị trí này.

Câu 2:

Code tham khảo:



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

#define MAXTABLESIZE 10000
const int TableSize = 10000;

struct NODE {
    string username;
    string password;
};

typedef NODE* HASHTABLE[MAXTABLESIZE];

unsigned int HF(const string& key) {
    unsigned int hashVal = 0;

    for (char ch : key)
        hashVal = 37 * hashVal + ch;

    return hashVal % TableSize;
}

int HF_LinearProbing(string key, int i) {
    return (HF(key) + i) % TableSize;
}

int isFull(HASHTABLE H, int CurrentSize) {
    if (CurrentSize == TableSize - 1)
        return 1;
    return 0;
}

void CreateHash(HASHTABLE& H, int& CurrentSize, int n) {
    CurrentSize = 0;
    for (int i = 0; i < TableSize; i++)
        H[i] = NULL;
    for (int i = 0; i < n; i++) {
        if (isFull(H, CurrentSize) == 1)
            return;
        NODE* p = new NODE;
        cin >> p->username;
        cin >> p->password;
        int b = HF(p->username);
```



```
    int j = 0;
    while (b < TableSize && H[b] != NULL && H[b]->username != p->username)
        b = HF_LinearProbing(p->username, ++j);
    if (H[b] == NULL) {
        H[b] = p;
        CurrentSize++;
        continue;
    }
    H[b]->password = p->password;
}
}

int Search(HASHTABLE H, int CurrentSize, string key) {
    int b = HF(key);
    int i = 0;
    if (H[b] == NULL)
        return -1;
    while (H[b]->username != key && H[b] != NULL)
        b = HF_LinearProbing(key, ++i);
    if (H[b]->username == key)
        return b;
}

void PrintPass(HASHTABLE H, int CurrentSize, int m) {
    int a[TableSize];
    int n = 0;
    for (int i = 0; i < m; i++) {
        string str;
        cin >> str;
        int b = Search(H, CurrentSize, str);
        if (b != -1)
            a[n++] = b;
        else
            a[n++] = -1;
    }
    for (int i = 0; i < n; i++) {
        if (a[i] == -1)
            cout << "Khong ton tai tai khoan." << endl;
        else
            cout << H[a[i]]->password << endl;
    }
}
```



```
}
```

```
int main() {
HASHTABLE H;
int CurrentSize;
int n, m;
cin >> n >> m;
CreateHash(H, CurrentSize, n);
PrintPass(H, CurrentSize, m);
return 0;
}
```



Phần 3. ĐỀ THI MẪU

Đề 01

Câu 1: (1.5 điểm)

Hãy cho biết độ phức tạp của thuật toán Heap sort theo định nghĩa Big-O (O lớn) (0.25 điểm)

Trình bày các bước giải thuật sắp xếp Heap sort (không viết chương trình) để sắp xếp mảng số nguyên N phần tử tăng dần (0.5 điểm)

Chạy từng bước thuật toán đã viết ở trên với dãy số sau: 1, 2, 5, 2, 8, 6 , 7 , 9 (0.75 điểm)

Câu 2: (3.5 điểm)

Cho dãy ký tự như sau: P, C, M, R, B, Q, U, N, H, A, Y, S, T.

Hãy thực hiện các yêu cầu sau:

- Vẽ cây nhị phân kiếm bằng cách thêm lần lượt từng ký tự vào cây theo thứ tự từ trái qua phải của dãy ký tự trên, biết rằng giá trị của từng ký tự tương ứng theo thứ tự xuất hiện của ký tự trong từ điển (1 điểm).
- Cho biết kết quả duyệt cây theo NLR, LNR. (0.5 điểm).
- Hủy lần lượt từng nút Y, S, C trên cây, mỗi lần hủy 1 nút vẽ lại cây nối tiếp theo như thứ tự hủy (1 điểm).
- Viết hàm đếm số nút lá trên cây. (1 điểm)

Câu 3: (2 điểm)

Cho biết cây B-Tree bậc 3 là một cây thỏa mãn các tính chất sau:

- Tất cả node lá nằm trên cùng một mức
- Tất cả các node, trừ node gốc và node lá, có *tối thiểu* 2 node con.
- Tất cả các node có *tối đa* 3 con
- Tất cả các node, trừ node gốc, có từ 1 cho đến 2 khóa (keys)
- Một node không phải lá và có n khóa thì phải có n+1 node con.

Hãy thực hiện các yêu cầu sau:

Cho dãy số: 30, 27, 35, 42, 19, 28, 51, 32, 39, 40, 20, 23, 26, 36, 37, 38. Hỏi khi lần lượt thêm các số trong dãy theo thứ tự từ trái qua phải vào một cây B-Tree bậc 3 rỗng thì:

- Các khóa nào khi thêm vào sẽ làm phát sinh thao tác split node? (0.5 điểm)
- Vẽ cây B-Tree trước và sau khi thêm các khóa trên. (1 điểm)
- Với cây B-Tree đã vẽ ở trên. Hãy lần lượt tiến hành xóa các khóa sau khỏi cây B-Tree: 27, 35, 40 và vẽ cây BTee sau khi xóa mỗi khóa trên. (0.5 điểm)

*Lưu ý khi xoá:



TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN – ĐHQG-HCM BAN HỌC TẬP CÔNG NGHỆ PHẦN MỀM

- Khi khóa cần xóa (gọi là x) không nằm ở node lá, chọn khóa thế mạng là khóa có giá trị lớn nhất mà nhỏ hơn x.
- Thao tác underflow sẽ được thực hiện khi hai node liền kề có tổng số khóa ≥ 2 . Khi có một node không còn đáp ứng đủ số lượng khóa tối thiểu, luôn ưu tiên thực hiện underflow thay cho catenation vì thao tác này không làm thay đổi số khóa của node cha.
- Khi có 02 lựa chọn node liền kề để thực hiện catenation, ưu tiên chọn catenate giữa node bị thiếu khóa với node liền trước.

Câu 4: (2 điểm) Cho một bảng băm theo phương pháp thăm dò bậc 2 với hàm băm $h(key)$ và hàm băm lại (hay hàm thăm dò) $prob(key, i)$ như sau:

$$h(key) = (key \% M) \quad prob(key, i) = (h(key) + i*i) \% M$$

Trong đó:

- key là giá trị khóa.
- i là một số nguyên cho biết lần thăm dò thứ i.
- M là kích thước bảng băm.

Cho $M = 7$ và trên bảng băm đã chứa các mục dữ liệu như bên dưới. Biết EMP và DEL lần lượt là ký hiệu để đánh dấu vị trí còn trống hoặc đã bị xóa trong bảng băm.

	Key
0	EMP
1	EMP
2	2
3	EMP
4	4
5	EMP
6	EMP

a. Trình bày từng bước việc thêm các khóa Key trong danh sách bên dưới vào bảng băm theo đúng thứ tự trong danh sách. (1 điểm)

STT	Key
1	6
2	16
3	10

b. Trình bày từng bước việc xóa giá trị Key = 16 trong bảng băm khi hoàn thành yêu cầu ở câu a. (0.5 điểm)



- c. Trình bày từng bước việc tìm giá trị Key = 10 trong bảng băm khi hoàn thành yêu cầu ở câu b. (0.5 điểm)

Câu 5 (Đề thi CTDL – GT kì 2 năm 2021 – 2022):

Trong ứng dụng thực tế, như các mạng lưới giao thông, người ta quan tâm đến việc tìm đường đi ngắn nhất (theo tiêu chuẩn không gian, thời gian hay chi phí). Vấn đề này được mô hình hóa thành một bài toán trên đồ thị, trong đó mỗi địa điểm được biểu diễn bởi một đỉnh, cạnh nối hai đỉnh biểu diễn cho “đường đi trực tiếp” giữa hai địa điểm (tức không đi qua địa điểm trung gian) và trọng số của cạnh là khoảng cách giữa hai địa điểm.

Bài toán có thể phát biểu dưới dạng tổng quát như sau: Cho một đơn đồ thị có hướng và có trọng số dương $G(V, E)$, trong đó V là tập đỉnh, E là tập cạnh (cung) và các cạnh đều có trọng số, hãy tìm một đường đi (không có đỉnh lặp lại) ngắn nhất xuất phát S thuộc V đến đỉnh đích F thuộc V .

Giả sử thông tin đầu vào của bài toán (input) được nhập vào chương trình như sau:

Ví dụ Input	Giải thích
7 A B 1 B E 3 E D 3 C B 4 A D 7 E C 2 C D 1 A E	- Dòng đầu tiên chứa một số e là số cạnh của đồ thị - e dòng tiếp theo, mỗi dòng chứa 02 chuỗi u, i và x , thể hiện thông tin có một cạnh nối từ đỉnh u sang đỉnh i trong đồ thị với độ dài (trọng số) là x . Dòng cuối cùng chứa hai chuỗi s và f , đây là đỉnh bắt đầu và đỉnh kết thúc của đường đi cần tìm Lưu ý: không biết trước số đỉnh và danh sách các đỉnh

Hãy thực hiện các yêu cầu sau:

- a. Xây dựng cấu trúc dữ liệu thích hợp để biểu diễn đồ thị trên máy tính theo input đã cho. (0.5 điểm)

Cấu trúc được xem là tốt nhất nếu đạt các điều kiện sau: tiết kiệm tài nguyên; hỗ trợ một số thao tác cơ bản như “Kiểm tra hai đỉnh có kề nhau không”, “Tìm danh sách các đỉnh kề với một đỉnh cho trước” với ràng buộc là không phải duyệt qua danh sách tất cả các cạnh của đồ thị.

- b. Viết hàm nhập đồ thị theo input ở đầu bài và lưu trữ thông tin của đồ thị vào cấu trúc dữ

liệu đã đề xuất ở câu a.
(0.5 điểm)

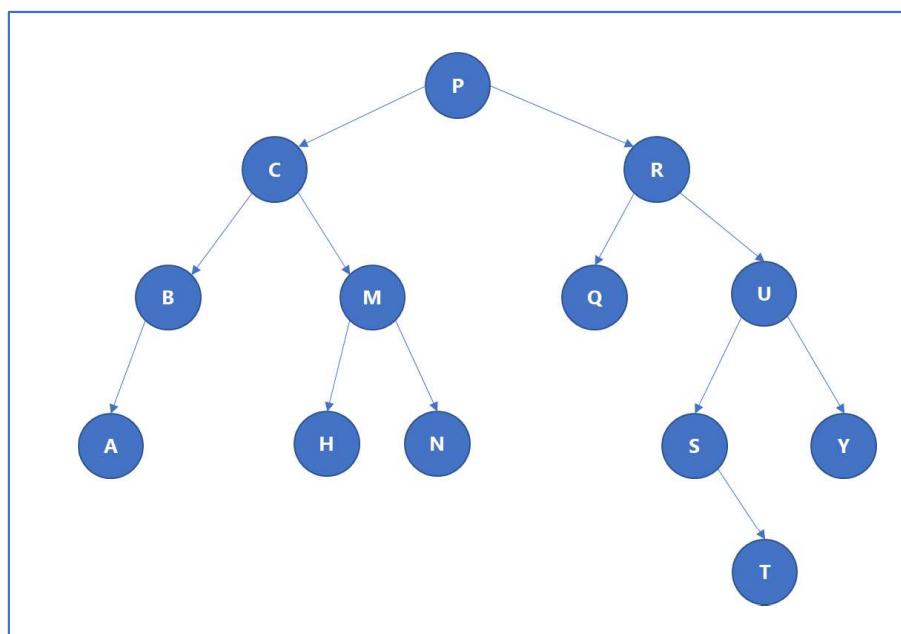
Giải đề 01

Câu 1:

Độ Phức tạp (Heap Sort): $O(n \log n)$

Gợi ý : Xây dựng hàm Heapify , sau đó gọi hàm Heapify với mọi nốt không phải là nốt lá. HS trình bày như cách đã học

Câu 2: a.

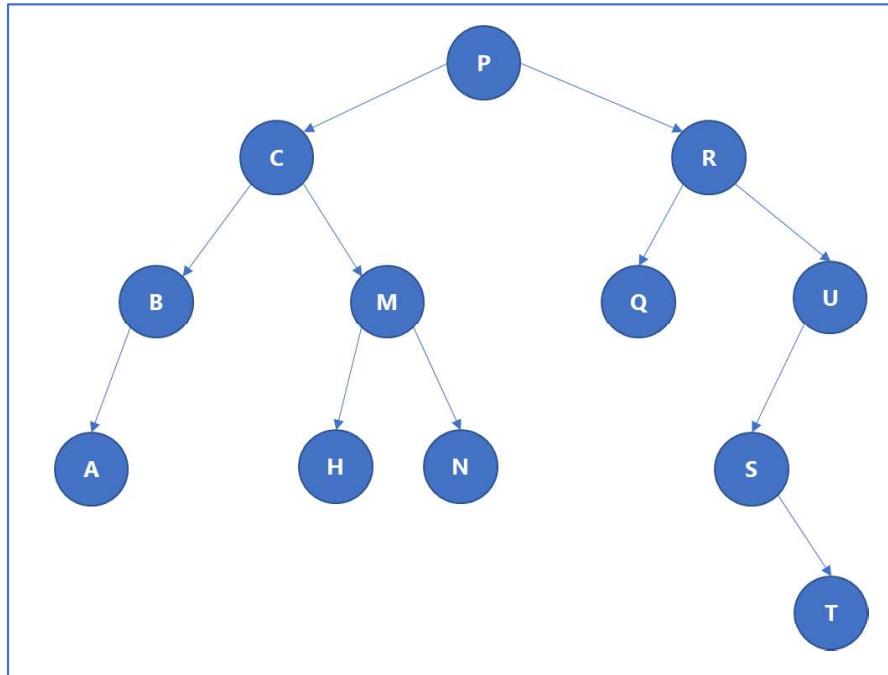


b.

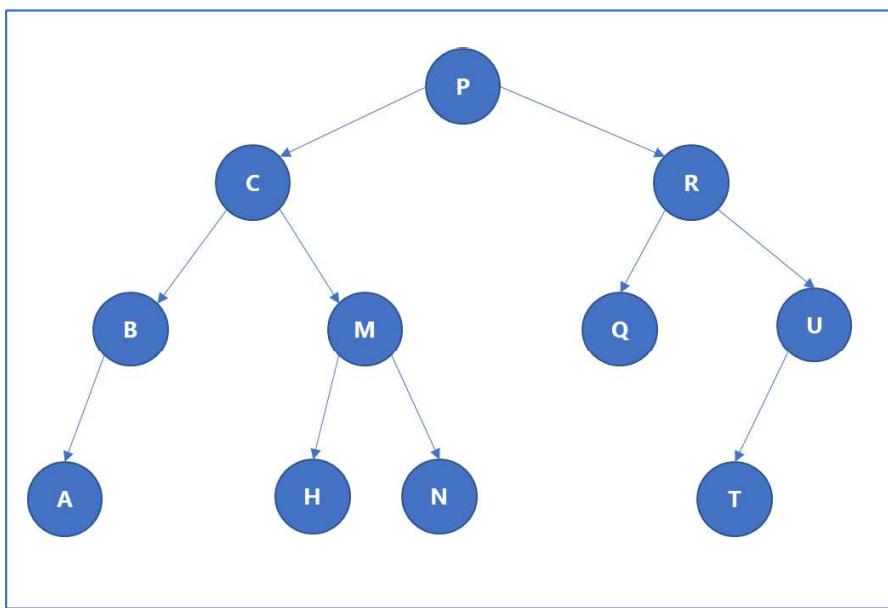
NLR: P, C, B, A, M, H, N, R, Q, U, S, T, Y

LNR: A, B, C, H, M, N, P, Q, R, S, T, U, Y

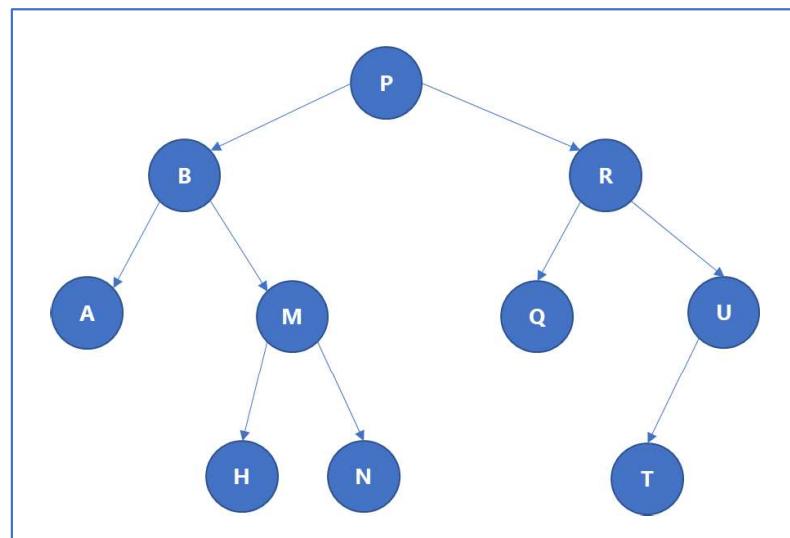
c. Hủy nút Y:



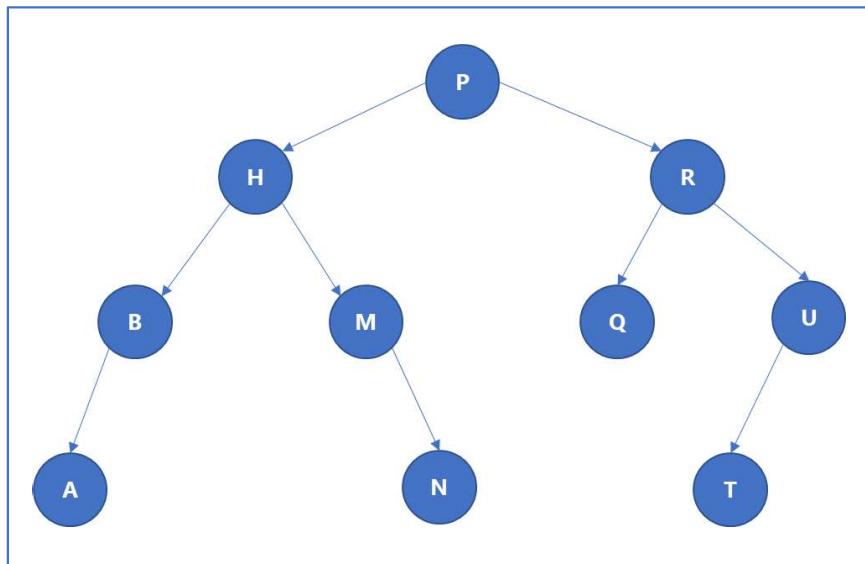
- Hủy nút S:



- Hủy nút C:



Hoặc



d.

```

int countLeaf(tree& t){
    if(t==NULL){
        return 0;
    }
    else if(t->left == NULL && t->right == NULL){
        return 1;
    }
    else{
  
```

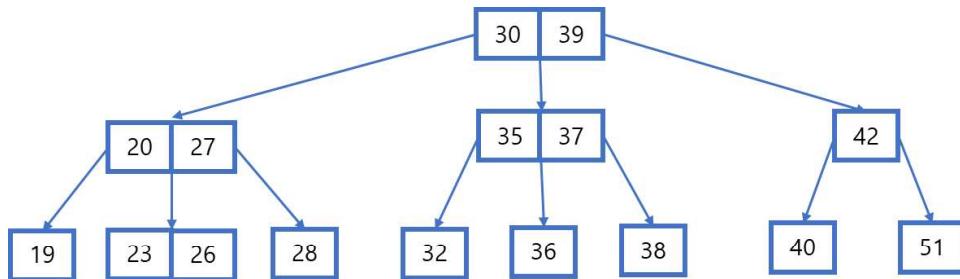
```

        return countLeaf(t->left) + countLeaf(t->right);
    }
}

```

Câu 3:

a.



b.

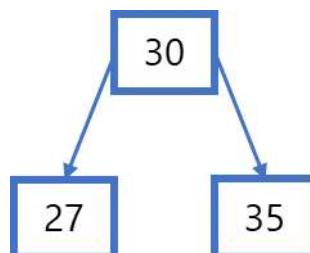
- Thao tác split: 35, 28, 51, 39, 23, 36, 38

*Thêm khóa 35

- Trước khi thêm khóa 35:

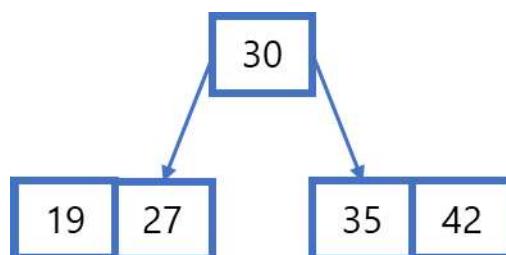


- Sau khi thêm khóa 35:

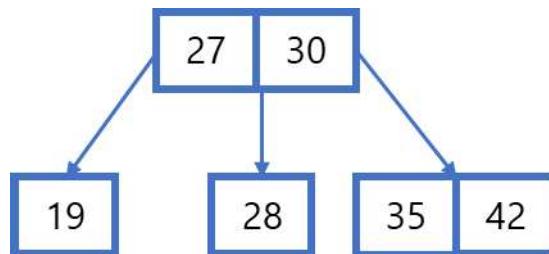


*Thêm khóa 28:

- Trước khi thêm khóa 28:

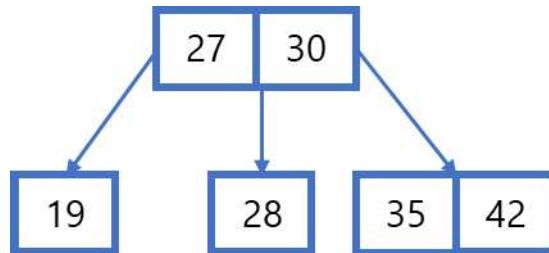


- Sau khi thêm khóa 28:

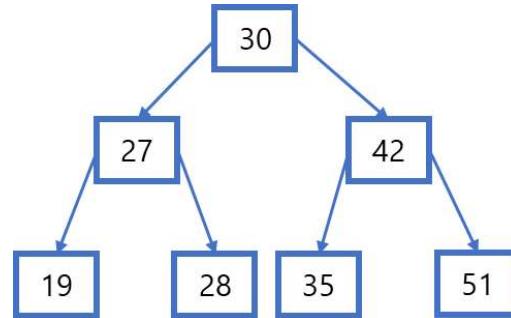


*Thêm khóa 51:

- Trước khi thêm khóa 51:

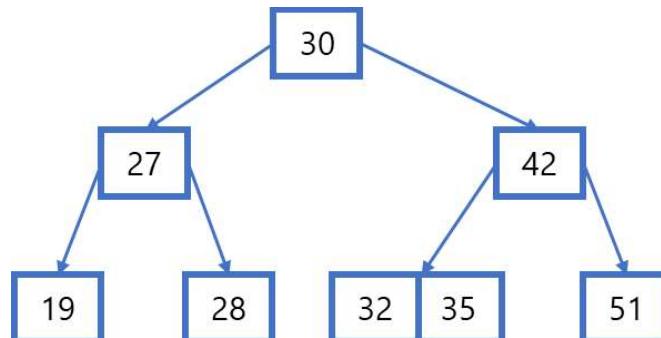


- Sau khi thêm khóa 51:

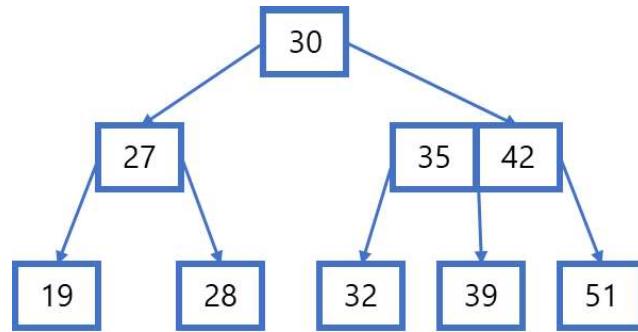


*Thêm khóa 39:

- Trước khi thêm khóa 39:

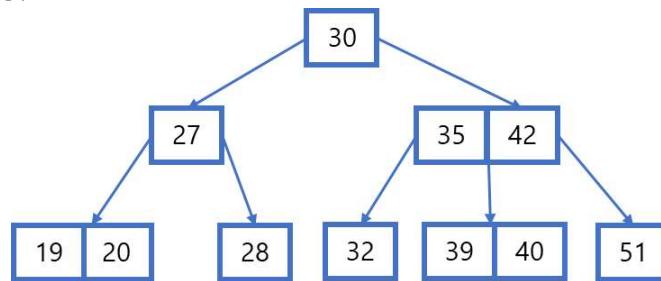


- Sau khi thêm khóa 39:

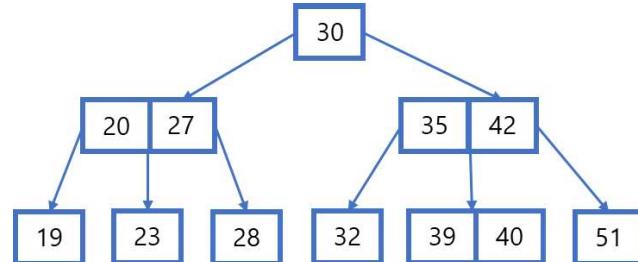


*Thêm khóa 23:

- Trước khi thêm khóa 23:

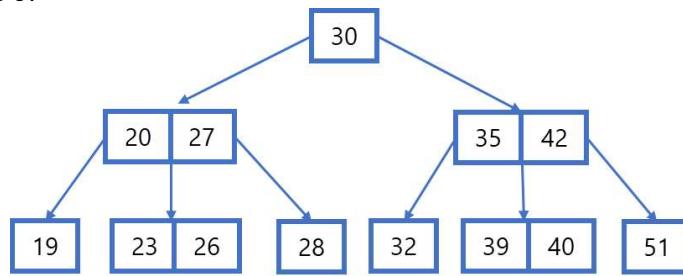


- Sau khi thêm khóa 23:

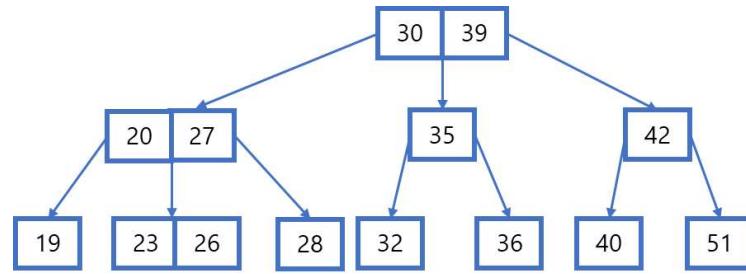


*Thêm khóa 36:

- Trước khi thêm khóa 36:

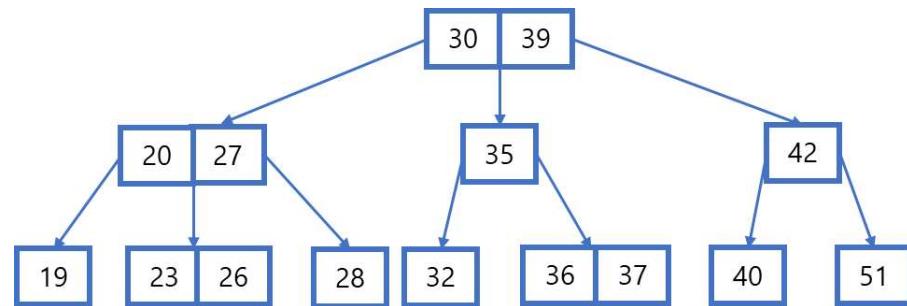


- Sau khi thêm khóa 36:

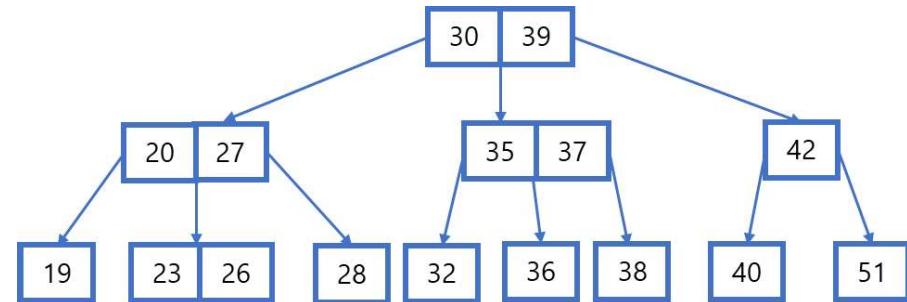


* Thêm khóa 38:

- Trước khi thêm khóa 38:

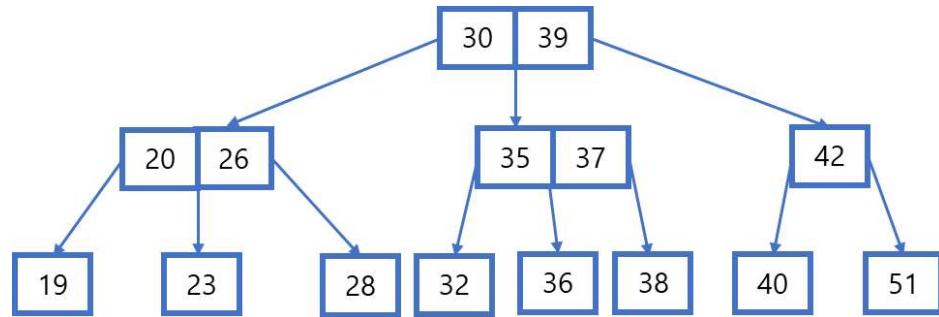


- Sau khi thêm khóa 38:

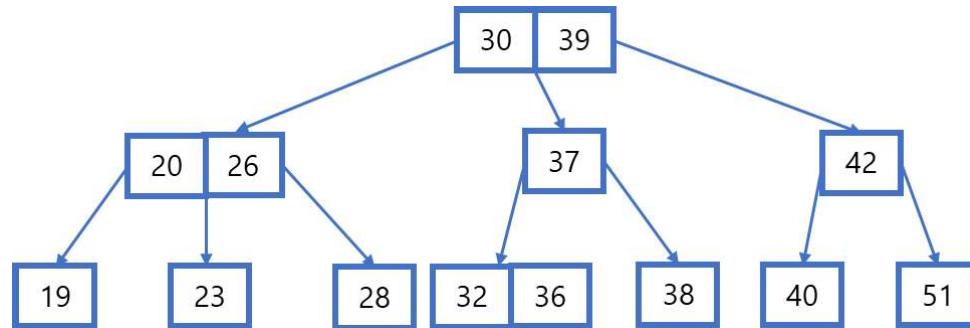


c.

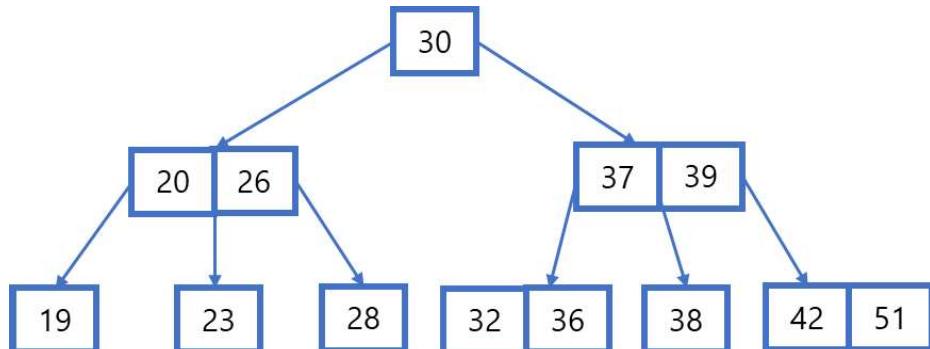
- Xóa khóa 27:



- Xóa khóa 35:



- Xóa khóa 40:



Câu 4:

a. Với $M = 7$, ta thêm các khóa Key vào theo thứ tự:

- Key = 6. Có $h(6) = 6 \% 7 = 6$

6 còn trống à Chèn 6 vào vị trí 6

- Key = 16. Có $h(16) = 16 \% 7 = 2$

2 không còn trống à Xảy ra đụng độ à Cần phải băm lại lần 1

$i = 1$. Có $\text{prob}(16, 1) = (h(16) + 1*1) \% 7 = 3$

3 còn trống à Chèn 16 vào vị trí 3

- Key = 10. Có $h(10) = 10 \% 7 = 3$

3 không còn trống à Xảy ra đụng độ à Cần phải băm lại lần 1

$i = 1$. Có $\text{prob}(10, 1) = (h(10) + 1*1) \% 7 = 4$

4 không còn trống à Xảy ra đụng độ à Cần phải băm lại lần 2

$i = 2$. Có $\text{prob}(10, 2) = (h(10) + 2*2) \% 7 = 0$

0 còn trống à Chèn 10 vào vị trí 0

Bảng băm sau khi chèn các khóa Key theo thứ tự:

	Key
0	10
1	EMP
2	2
3	16
4	4
5	EMP
6	6

b. Xóa giá trị Key = 16 trong bảng băm à Cần phải tìm vị trí có Key = 16 bằng cách dùng hàm băm.

- Key = 16. Có $h(16) = 16 \% 7 = 2$



Vị trí 2 có Key = 2 # 16 không phải giá trị cần xóa à Cần phải băm lại lần 1

i = 1. Có prob(16, 1) = (h(16) + 1*1) % 7 = 3

Vị trí 3 có Key = 16 = 16 chính là giá trị cần xóa.

Bảng băm sau khi xóa Key = 16:

	Key
0	10
1	EMP
2	2
3	DEL
4	4
5	EMP
6	6

c. Tìm giá trị Key = 10 có trong bảng băm à Tìm bảng cách dùng hàm băm

- Key = 10. Có h(10) = 10 % 7 = 3

Vị trí 3 có Key = DEL # 10 không phải giá trị cần tìm à Cần phải băm lại lần 1

i = 1. Có prob(10, 1) = (h(10) + 1*1) % 7 = 4

Vị trí 4 có Key = 4 # 10 không phải giá trị cần tìm à Cần phải băm lại lần 2

i = 2. Có prob(10, 2) = (h(10) + 2*2) % 7 = 0

Vị trí 0 có Key = 10 = 10 chính là giá trị cần tìm. Vậy Key = 10 nằm ở vị trí 0 trong bảng băm.

Câu 5:

```
#include <iostream>
#include <vector>
#include <set>
#include <map>
#include <string>
using namespace std;

int main()
{
    //Nhập
    int e;
    //cau truc du lieu theo yêu cầu bài
    map<string, map<string, int>> DIEM;
    for (int i = 0; i < e; i++)
    {
        string u, v;
        cin >> u >> v;
        int x;
```



TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN – ĐHQG-HCM BẢN HỌC TẬP CÔNG NGHỆ PHẦN MỀM

```
    cin >> x;
    DIEM[u][v] = x;
    DIEM[v][u] = x;
}
return 0;
}
```



Tài liệu tham khảo:

- [1]. Sách Data Structures And Algorithm Analysis In C++ 4th Edition, Mark Allen Weiss.
- [2]. Web kiến thức, Geeks For Geeks, link: [GeeksforGeeks | A computer science portal for geeks](https://www.geeksforgeeks.org/)
- [3]. Web mô tả thuật toán, visualgo, link: [Sorting \(Bubble, Selection, Insertion, Merge, Quick, Counting, Radix\) - VisuAlgo](https://visualgo.net/en/)
- [4]. Web mô tả thuật toán, hackerearth link: <https://www.hackerearth.com/>
- [5]. Web mô tả thuật toán, USF – University of San Francisco, link: [Data Structure Visualization \(usfca.edu\)](https://ufe.ca.usfca.edu/)
- [6]. Slide các giảng viên UIT