Ôn thi DSA

Lưu hành nội bộ - By TSR - L (Tensoract Lab)

Phần 1: Thuật toán tìm kiếm

1. **Thuật toán tìm kiếm nhị phân - Binary Search**

- **Ý tưởng:** Thuật toán tìm kiếm nhị phân là một thuật toán **tìm kiếm xác định vị trí của một giá trị cần tìm trong một mảng đã được sắp xếp**. Thuật toán **tiến hành so sánh giá trị cần tìm với phần tử đứng giữa mảng**. Nếu giá trị bằng nhau thì xuất ra vị trí ấy, ngược lại nếu hai giá trị không bằng nhau, phần nửa mảng không chứa giá trị cần tìm sẽ bị bỏ qua và tiếp tục tìm kiếm trên nửa còn lại, một lần nữa lấy phần tử ở giữa và so sánh với giá trị cần tìm, cứ thế lặp lại cho đến khi tìm thấy giá trị đó.
- **Độ phức tạp:** O(log(n)).
- **Cài đặt:**

```
int BinarySearch(int arr[], int n, int x)
{
    int left = 0, right = n - 1;
    while (left <= right)
    {
        int mid = (left + right) / 2;
        if (arr[mid] == x)
            return mid;
        else if (arr[mid] > x)
            right = mid - 1;
        else
            left = mid + 1;
    }
    return -1;
}
```

Phần 2: Thuật toán sắp xếp

1. **Thuật toán sắp xếp chọn - Selection Sort**

- **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.
- **Độ phức tạp:** O(n^2).
- **Cài đặt:**

```
void swap(int &a, int &b)
{
    int temp = a;
    a = b;
    b = temp;
}

void selectionSort(int arr[], int n)
{
    int i, j, min;
    for (int i = 0; i < n - 1; i++)
    {
        min = i;
        for (int j = i + 1; j < n; j++)
        {
            if (arr[j] < arr[min]) // Neu giam dan thi (arr[j] > arr[min])
                min = j;
        }
        if (min != i)
            swap(arr[i], arr[min]);
    }
}
```

2. **Thuật toán sắp xếp Chèn - Insertion Sort (Chèn bảo toàn)**

- **Ý tưởng:** Thuật toán sắp xếp chèn **thực hiện sắp xếp dãy số theo cách duyệt từng phần tử và chèn từng phần tử đó vào đúng vị trí trong mảng con (dãy số từ đầu đến phần tử phía trước nó)** đã sắp xếp, sao cho dãy số trong mảng sắp đã xếp đó vẫn **đảm bảo tính chất của một dãy số tăng dần(hoặc giảm dần).**
- **Độ phức tạp:** O(n^2)
- **Cài đặt:**

```
void insertionSort(int arr[], int n)
{
    int j, current;
    for (int i = 1; i < n; i++)
    {
        current = arr[i];
        for (j = i - 1; j >= 0; j--)
        {
            if (arr[j] > current) // Neu giam dan thi (arr[j] < current)
                arr[j + 1] = arr[j];
            else
                break;
        }
        arr[j + 1] = current;
    }
}
```

3. **Thuật toán sắp xếp Gộp - Merge Sort**

- **Ý tưởng: Sắp xếp trộn** là một thuật toán sắp xếp dựa trên ý tưởng **Chia để trị**. Thuật toán này sẽ **chia mảng thành hai nửa rồi đệ quy phân chia thành các mảng con nhỏ hơn, đến khi chỉ còn một phần tử thì tiến hành trộn và sắp xếp lại trên từng nửa một**. Sau đó **kết hợp chúng lại với nhau thành một mảng đã được sắp xếp**.
- **Độ phức tạp:** O(n log(n))
- **Cài đặt:**

```
void Merge(int arr[], int left, int mid, int right)
{
    int i, j, k;
    int n1 = mid - left + 1;
    int n2 = right - mid;
    int L[n1], R[n2];
    for (i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];
    i = 0;
    j = 0;
    k = left;
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j]) // Neu giam dan doi thanh (L[i] >= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
```

```
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void MergeSort(int arr[], int left, int right)
{
    if (left < right)
    {
        int mid = (left + right) / 2;
        MergeSort(arr, left, mid);
        MergeSort(arr, mid + 1, right);
        Merge(arr, left, mid, right);
    }
}
```

4. **Thuật toán sắp xếp Nhanh - Quick Sort**

- **Ý tưởng: Thuật toán QuickSort** là một thuật toán **sắp xếp nhanh,** sử dụng phương pháp **chia để trị,** bằng cách chọn **phần tử chốt**, tiến hành **phân hoạch** và **chia mảng làm 2 phần(1 phần nhỏ hơn phần tử chốt và một phần lớn hơn phần tử chốt)**. Sau đó tiến hành **đệ quy phân hoạch và sắp xếp trên 2 phần này** đến khi **không còn phần tử nào để đệ quy thì thuật toán kết thúc**, tạo thành mảng đã sắp xếp.
- **Độ phức tạp:** O(n log(n))
- **Cài đặt:**

```
void swap(int &a, int &b)
{
    int temp = a;
    a = b;
    b = temp;
}

int partionLomuto(int arr[], int left, int right)
{
```

```
    int pivot = arr[right];
    int i = left - 1;
    for (int j = left; j < right; j++)
    {
        if (arr[j] <= pivot) // Neu giam dan thi doi thanh (arr[j] >= pivot)
        {
            ++i;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[right]);
    return i + 1;
}

void QuickSort(int arr[], int left, int right)
{
    if (left < right)
    {
        int pivot = partionLomuto(arr, left, right);
        QuickSort(arr, left, pivot - 1);
        QuickSort(arr, pivot + 1, right);
    }
}
```

5. **Thuật toán sắp xếp Vun đống - Heap Sort**

- **Ý tưởng: Heap Sort** là kỹ thuật sắp xếp dựa trên so sánh dựa trên cấu trúc dữ liệu Heap. Nó tương tự như **sắp xếp chọn**, nơi đầu tiên chúng ta tìm phần tử lớn nhất và đặt phần tử lớn nhất ở cuối cây Heap. Chúng ta lặp lại quá trình tương tự cho các phần tử còn lại.
- **Độ phức tạp:** O(n log(n))
- **Cài đặt:**

```
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]) // Neu giam dan thi doi thanh (left < n
 && arr[left] < arr[largest])
        largest = left;
    if (right < n && arr[right] > arr[largest]) // Neu giam dan thi doi thanh (right <
n && arr[right] < arr[largest])
        largest = right;
    if (largest != i)
    {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}
```

```
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 3: Cấu trúc dữ liệu

1. **Danh sách liên kết đơn - Single Linked List**

- **Định nghĩa: Danh sách liên kết đơn (Single Linked List)** là một cấu trúc dữ liệu động, nó là một **danh sách mà mỗi phần tử đều liên kết với phần tử đúng sau nó trong danh sách**. Mỗi phần tử (được gọi là một node hay nút) trong danh sách liên kết đơn là một cấu trúc có hai thành phần:
  - **Thành phần dữ liệu:** lưu thông tin về bản thân phần tử đó.
  - **Thành phần liên kết:** ***lưu địa chỉ phần tử đứng sau trong danh sách***, nếu phần tử đó là phần tử cuối cùng thì thành phần này bằng **NULL**.
- **Các thao tác với danh sách liên kết đơn:**
- **Định nghĩa danh sách liên kết đơn:**

```
struct node
{
    <KDL> info;
    struct node *pNext;
};
typedef struct node NODE;

struct list
{
    NODE *pHead;
    NODE *pTail;
};
typedef struct list LIST;
```

- **Khởi tạo danh sách liên kết đơn:**

```
void Init(LIST &l)
{
    l.pHead = NULL;
```

```
    l.pTail = NULL;
}
```

- **Hàm kiểm tra danh sách liên kết đơn rỗng:**

```
int IsEmpty(LIST &l)
{
    if (l.pHead == NULL)
        return 1;
    return 0;
}
```

- **Hàm tạo Node mới trong danh sách liên kết:**

```
NODE* CreateNode(<KDL> x)
{
    NODE *p = new NODE;
    if (p == NULL) return NULL;
    p->info = x;
    p->pNext = NULL;
    return p;
}
```

- **Hàm thêm node vào đầu danh sách liên kết:**

```
void AddHead(LIST &l, NODE *p)
{
    if (l.pHead == NULL)
        l.pHead = l.pTail = p;
    else
    {
        p->pNext = l.pHead;
        l.pHead = p;
    }
}
```

- **Hàm thêm node vào cuối danh sách liên kết:**

```
void AddTail(LIST &l, NODE *p)
{
    if (l.pHead == NULL)
    {
        l.pHead = l.pTail = p;
    }
    else
    {
        l.pTail->pNext = p;
```

```
        l.pTail = p;
    }
}
```

- **Hàm thêm một node vào sau một node bất kì trong danh sách liên kết:**

```
void InsertAfterQ(LIST &l, NODE *p, NODE *q)
{
    if (q != NULL)
    {
        p->pNext = q->pNext;
        q->pNext = p;
        if (l.pTail == q)
            l.pTail = p;
    }
    else
        AddHead(l, p);
}
```

- **Hàm xoá node Head của danh sách liên kết:**

```
void RemoveHead(LIST &l)
{
    if (l.pHead != NULL)
    {
        NODE *node = l.pHead;
        l.pHead = node->pNext;
        delete node;
        if (l.pHead == NULL)
            l.pTail = NULL;
    }
}
```

- **Hàm xoá node ở sau một node bất kì trong danh sách liên kết:**

```
void RemoveAfterQ(LIST &l, NODE *q)
{
    if (q != NULL)
    {
        NODE *p = q->pNext;
        if (p != NULL)
        {
            if (l.pTail == p)
                l.pTail = q;
            q->pNext = p->pNext;
            delete p;
        }
    }
}
```

- **Hàm xoá node Tail của danh sách liên kết:**

```
void RemoveTail(LIST &l)
{
    if (l.pHead != NULL)
    {
        NODE *p = l.pHead;
        while (p->pNext != l.pTail)
            p = p->pNext;
        delete l.pTail;
        l.pTail = p;
        l.pTail->pNext = NULL;
        if (l.pTail == NULL)
            l.pHead = NULL;
    }
}
```

- **Hàm xoá node có giá trị x trong danh sách liên kết:**

```
void RemoveX(LIST &l, <KDL> x)
{
    NODE *p = l.pHead;
    NODE *q = NULL;
    while (p != NULL)
    {
        if (p->info == x)
            break;
        q = p;
        p = p->pNext;
    }
    if (p != NULL)
        RemoveAfterQ(l, q);
}
```

- **Hàm tìm kiếm node trong danh sách liên kết:**

```
Node* Search(LIST l, <KDL> x)
{
  NODE* node = l.pHead;
  while (node != NULL && node->info != x)
    node = node->pNext;
  if (node != NULL)
    return node;
  return NULL;
}
```

- **Hàm sắp xếp chọn - Selection Sort cho danh sách liên kết đơn:**

```
void SelectionSort(LIST &l)
{
    NODE *p = l.pHead;
    NODE *q = new NODE;
    NODE *min = new NODE;
    while (p != l.pTail)
    {
        min = p;
        q = p->pNext;
        while (q != NULL)
        {
            if (q->info < min->info) // Neu giam dan thi doi thanh (q->info > min->inf
o)
                min = q;
            q = q->pNext;
        }
        // Hoan doi p->info and min->info
        int temp = p->info;
        p->info = min->info;
        min->info = temp;
        p = p->pNext;
    }
}
```

- **Hàm sắp xếp nhanh - Quick Sort cho danh sách liên kết đơn:**

```
void QuickSort(LIST &l)
{
    NODE *p = new NODE;
    NODE *X = new NODE; // X la phan tu cam canh
    LIST l1, l2;
    if (l.pHead == l.pTail)
        return; // Da co thu tu
    CreateEmptyList(l1);
    CreateEmptyList(l2);
    X = l.pHead;
    l.pHead = X->pNext;
    while (l.pHead != NULL) // Chia l thanh l1 va l2
    {
        p = l.pHead;
        l.pHead = p->pNext;
        p->pNext = NULL;
        if (p->info <= X->info) // Neu giam dan thi doi thanh (p->info >= X->info)
            AddHead(l1, p);
        else
            AddHead(l2, p);
    }
    QuickSort(l1);
    QuickSort(l2);
    // Noi l1, X, l2 lai voi nhau
    if (l1.pHead != NULL)
    {
        l.pHead = l1.pHead;
        l1.pTail->pNext = X; // Noi X vao cuoi l1
```

```
    }
    else
        l.pHead = X;
    X->pNext = l2.pHead;
    if (l2.pHead != NULL) // l2 co 1 phan tu
        l.pTail = l2.pTail;
    else // l2 rong
        l.pTail = X;
}
```

- **Hàm in danh sách liên kết:**

```
void PrintList(LIST l)
{
  if (l.pHead != NULL)
  {
    NODE* node = l.pHead;
    while (node != NULL)
    {
      cout << node->info << ' ';
      node = node->pNext;
    }
  }
}
```

- **Hàm đảo ngược danh sách liên kết:**

```
void Reverse(LIST &l)
{
    NODE* prev = NULL;
    NODE* current = l.pHead;
    NODE* next = NULL;
    while (current != NULL)
    {
        next = current->pNext;
        current->pNext = prev;
        prev = current;
        current = next;
    }
    l.pHead = prev;
}
```

- **Hàm huỷ danh sách liên kết:**

```
void DestroyList(LIST &l)
{
    NODE *node = l.pHead;
    while (node != NULL)
    {
```

```
        RemoveHead(l);
        node = l.pHead;
    }
    l.pTail = NULL;
}
```

2. **Ngăn xếp - Stack**

- **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.
- **Các thao tác với ngăn xếp:**
  - **Push**: Thêm một phần tử vào đỉnh của ngăn xếp, số phần tử của ngăn xếp tăng lên 1.
  - **Pop**: Xóa bỏ phần tử đầu tiên ở đỉnh của ngăn xếp, số phần tử của ngăn xếp giảm đi 1.
  - **Top**: Lấy giá trị của phần tử đầu tiên ở đỉnh của ngăn xếp, số phần tử của ngăn xếp không thay đổi.
  - **IsEmpty**: Kiểm tra ngăn xếp trống hay không. Ngăn xếp trống là ngăn xếp không có phần tử nào.
  - **Size**: Lấy số lượng phần tử stack đang có.
- **Cài đặt:**
- **Định nghĩa ngăn xếp:**

```
struct node
{
    <KDL> info;
    struct node *pNext;
};
typedef struct node NODE;

struct stack
{
    NODE *pHead;
};
typedef struct stack STACK;
```

- **Hàm tạo node mới cho ngăn xếp:**

```
NODE *CreateNode(<KDL> init)
{
    NODE *node = new NODE;
    if (node == NULL)
        return NULL;
    node->info = init;
    node->pNext = NULL;
    return node;
}
```

- **Hàm khởi tạo ngăn xếp mới:**

```
void CreateStack(STACK &s)
{
    s.pHead = NULL;
}
```

- **Hàm kiểm tra ngăn xếp rỗng:**

```
int IsEmpty(STACK s)
{
    if (s.pHead == NULL)
        return 1;
    return 0;
}
```

- **Hàm thêm một phần tử vào ngăn xếp (Push):**

```
void Push(STACK &s, NODE *node)
{
    if (s.pHead == NULL)
        s.pHead = node;
    else
    {
        node->pNext = s.pHead;
        s.pHead = node;
    }
}
```

- **Hàm lấy một phần tử ra khỏi ngăn xếp (Pop):**

```
<KDL> Pop(STACK &s)
{
    if (IsEmpty(s) == true)
        return 0;
```

```
    NODE *node = s.pHead;
    <KDL> info = node->info; // lưu trữ lại giá trị của node
    s.pHead = node->pNext;
    delete node;
    return info;
}
```

- **Hàm trả về số lượng nút của ngăn xếp:**

```
int Size(STACK s)
{
    int count = 0;
    NODE *p = s.pHead;
    while (p != NULL)
    {
        count++;
        p = p->pNext;
    }
    return count;
}
```

- **Hàm lấy giá trị phần tử trên cùng của ngăn xếp (Top):**

```
<KDL> Top(STACK s)
{
    if (IsEmpty(s) == true)
        return 0;
    return s.pHead->info;
}
```

- **Hàm in ngăn xếp:**

```
void PrintStack(STACK s)
{
    while (IsEmpty(s) == false)
    {
        cout << Pop(s) << " ";
    }
}
```

3. **Hàng đợi - Queue**

- **Định nghĩa: 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.

- **Các thao tác với hàng đợi:**
  - **EnQueue**: Thêm phần tử vào cuối *(rear)* của Queue.
  - **DeQueue**: Xóa phần tử khỏi đầu *(front)* của Queue. Nếu Queue rỗng thì thông báo lỗi.
  - **IsEmpty**: Kiểm tra Queue rỗng.
  - **Front**: Lấy giá trị của phần tử ở đầu*(front)* của Queue. Lấy giá trị không làm thay đổi Queue.
  - **Size:** Lấy số lượng phần tử ngăn xếp hiện có.
- **Cài đặt:**
- **Định nghĩa hàng đợi:**

```
struct node
{
    <KDL> info;
    struct node *pNext;
};
typedef struct node NODE;

struct queue
{
    NODE *pHead;
    NODE *pTail;
};
typedef struct queue QUEUE;
```

- **Hàm khởi tạo ngăn xếp mới:**

```
void CreateQueue(QUEUE &q)
{
    q.pHead = NULL;
    q.pTail = NULL;
}
```

- **Hàm tạo Node mới cho hàng đợi:**

```
NODE *CreateNode(int x)
{
    NODE *p = new NODE;
    if (p == NULL)
        return NULL;
    p->info = x;
    p->pNext = NULL;
```

```
    return p;
}
```

- **Hàm kiểm tra hàng đợi rỗng:**

```
int IsEmpty(QUEUE q)
{
    if (q.pHead == NULL)
        return 1;
    return 0;
}
```

- **Hàm thêm phần tử vào cuối hàng đợi (EnQueue):**

```
void EnQueue(QUEUE &q, NODE *node)
{
    if (IsEmpty(q) == true)
    {
        q.pHead = node;
        q.pTail = node;
    }
    else
    {
        q.pTail->pNext = node;
        q.pTail = node;
    }
}
```

- **Hàm lấy phần tử ở đầu hàng đợi (DeQueue):**

```
<KDL> DeQueue(QUEUE &q)
{
    if (IsEmpty(q) == true)
        return 0;
    NODE *node = q.pHead;
    <KDL> info = node->info;
    q.pHead = node->pNext;
    delete node;
    if (q.pHead == NULL)
        q.pTail = NULL;
    return info;
}
```

- **Hàm lấy phần tử ở đầu hàng đợi:**

```
<KDL> Front(QUEUE q)
{
```

```
    if (IsEmpty(q) == true)
        return 0;
    return q.pHead->info;
}
```

- **Hàm trả về số lượng nút của hàng đợi:**

```
int Size(QUEUE q)
{
    int count = 0;
    NODE *p = q.pHead;
    while (p != NULL)
    {
        count++;
        p = p->pNext;
    }
    return count;
}
```

- **Hàm in ra hàng đợi:**

```
void PrintQueue(QUEUE q)
{
    while (IsEmpty(q) == false)
    {
        cout << DeQueue(q) << " ";
    }
}
```

4. **Cây nhị phân và cây nhị phân tìm kiếm - Binary Search Tree**

- **Định nghĩa:**
  - **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 **T1, T2, ..., Tn.** Trong đó các **Ti** cũng là một cây **(cây con).**
  - **Cây nhị phân:** Cây nhị phân là một trường hợp đặc biệt của cấu trúc cây và nó cũng phổ biến nhất. Đúng như tên gọi của nó, cây nhị phân có **bậc là 2** và mỗi nút trong cây nhị phân **đều có bậc không quá 2**.
  - **Cây nhị phân tìm kiếm: Cây nhị phân tìm kiếm là cây nhị phân** mà trong đó, **các phần tử của cây con bên trái đều nhỏ hơn phần tử hiện hành** và

các phần tử của cây con bên phải đều lớn hơn phần tử hiện hành. Do tính chất này, cây nhị phân tìm kiếm **không được có phần tử cùng giá trị.**

- **Cài đặt:**
- **Định nghĩa cấu trúc node và cây:**
  - Cấu trúc 1 nút gồm:
    - Trường chứa dữ liệu của nút (info).
    - Con trỏ lưu địa chỉ nút gốc của cây con bên trái (left).
    - Con trỏ lưu địa chỉ nút gốc của cây con bên (right).

```
struct Node
{
    <KDL> info;
    struct Node *left;
    struct Node *right;
};
typedef Node* Tree;
```

- **Hàm tạo node mới cho cây nhị phân:**

```
Node *CreateNode(<KDL> init)
{
    Node *p = new Node;
    if(p == NULL)
      return NULL;
    p->info = init;
    p->left = NULL;
    p->right = NULL;
    return p;
}
```

- **Hàm khởi tạo cây:**

```
void CreateTree(Tree &root)
{
    root = NULL;
}
```

- **Hàm duyệt cây nhị phân:**
  - **Duyệt tiền tự (Node - Left - Right):**

```
void NLR(Tree root)
{
    if (root != NULL)
    {
        // Xử lý nút gốc (root)
        NLR(root->left);
        NLR(root->right);
    }
}
```

- **Duyệt trung tự (Left - Node - Right):**

```
void LNR(Tree root)
{
    if (root != NULL)
    {
        LNR(root->left);
        // Xử lý nút gốc (root)
        LNR(root->right);
    }
}
```

- **Duyệt hậu tự (Left - Right - Node):**

```
void LRN(Tree root)
{
    if (root != NULL)
    {
        LRN(root->left);
        LRN(root->right);
        // Xử lý nút gốc (root)
    }
}
```

- **Hàm huỷ cây nhị phân:**

```
void DestroyTree(Tree &root)
{
    if (root != NULL)
    {
        DestroyTree(root->left);
        DestroyTree(root->right);
        delete root;
    }
}
```

- **Hàm thêm phần tử vào cây nhị phân tìm kiếm:**

```
void AddNode(Tree &root, Node *node)
{
    if (root != NULL)
    {
        if (root->info == node->info)
            return;
        if (node->info < root->info)
            AddNode(root->left, node);
        else
            AddNode(root->right, node);
    }
    else
    {
        root = node;
    }
}
```

- **Hàm tìm kiếm node trong cây nhị phân tìm kiếm:**

```
Node *FindNode(Tree root, <KDL> x)
{
    if (root != NULL)
    {
        if (root->info == x)
            return root;
        if (x < root->info)
            return FindNode(root->left, x);
        return FindNode(root->right, x);
    }
    return NULL;
}
```

- **Hàm tính chiều cao của cây nhị phân:**

```
int HeightBST(Tree root)
{
    if (root == NULL)
        return 0;
    int left = HeightBST(root->left);
    int right = HeightBST(root->right);
    if (left > right)
        return left + 1;
    return right + 1;
}
```

- **Hàm đếm số nút của cây nhị phân:**

```
int CountNode(Tree root)
{
    if (root == NULL)
        return 0;
    return 1 + CountNode(root->left) + CountNode(root->right);
}
```

- **Hàm đếm số nút có 1 cây con của cây nhị phân:**

```
int CountNodeHasOneChild(Tree root)
{
    if (root == NULL)
        return 0;
    if (root->left == NULL && root->right != NULL)
        return 1 + CountNodeHasOneChild(root->right);
    if (root->left != NULL && root->right == NULL)
        return 1 + CountNodeHasOneChild(root->left);
    return CountNodeHasOneChild(root->left) + CountNodeHasOneChild(root->right);
}
```

- **Hàm đếm số nút lá có 2 cây con của cây nhị phân:**

```
int CountNodeHasTwoChild(Tree root)
{
    if (root == NULL)
        return 0;
    if (root->left != NULL && root->right != NULL)
        return 1 + CountNodeHasTwoChild(root->left) + CountNodeHasTwoChild(root->righ
t);
    return CountNodeHasTwoChild(root->left) + CountNodeHasTwoChild(root->right);
}
```

- **Hàm đếm số nút lá của cây nhị phân:**

```
int CountLeaf(Tree root)
{
    if (root == NULL)
        return 0;
    if (root->left == NULL && root->right == NULL)
        return 1;
    return CountLeaf(root->left) + CountLeaf(root->right);
}
```

- **Hàm tính tổng các nút ở cùng một mức k của cây nhị phân:**

```
int SumNodeAtSameLevel(Tree root, int k, int level = 0)
{
    if (root == NULL)
        return 0;
    if (level == k)
        return root->info;
    return SumNodeAtSameLevel(root->left, k, level + 1) + SumNodeAtSameLevel(root->rig
ht, k, level + 1);
}
```

- **Hàm huỷ một nút trong cây nhị phân tìm kiếm:**
  - Để hủy một nút có khóa X trong cây nhị phân tìm kiếm, chúng ta cần giải quyết ba trường hợp sau:

1. Nút X là nút lá, ta xóa đi mà không làm ảnh hưởng đến các nút khác.
2. Nút X có 1 cây con, chúng ta chỉ cần nối nút cha của X với nút con của X.
3. Nút X có đầy đủ 2 cây con: vì X có đầy đủ 2 nút nên nếu ta xóa đi, ta sẽ bị mất toàn bộ cây con. Do đó chúng ta cần tìm phần tử thế mạng cho X mà vẫn đảm bảo được cây nhị phân tìm kiếm, sau đó mới xóa X đi.

  - Đối với hai trường hợp đầu thì dễ, tuy nhiên, với trường hợp thứ 3, chúng ta cần phải giải quyết vấn đề tìm phần tử thế mạng cho X, chúng ta sẽ có hai cách thực hiện như sau:

1. Nút thế mạng là **nút có khóa nhỏ nhất (min right) của cây con bên phải X**.
   - **Hàm tìm và thay thế nút thế mạng (nhỏ nhất bên phải):**

```
// p la nut can thay the, root la cay dang xet
void FindAndReplaceMinRight(Tree &p, Tree &root)
{
    if (root->left != NULL)
        FindAndReplaceMinRight(p, root->left);
    else
    {
        p->info = root->info;
        p = root;
        root = root->right;
    }
}
```

2. Nút thế mạng là **nút có khóa lớn nhất (max left) của cây con bên trái X**.
   - **Hàm tìm và thay thế nút thế mạng (lớn nhất bên trái):**

```
// p la nut can thay the, root la cay dang xet
void FindAndReplaceMaxLeft(Tree &p, Tree &root)
{
    if (root->right != NULL)
        FindAndReplaceMaxLeft(p, root->right);
    else
    {
        p->info = root->info;
        p = root;
        root = root->left;
    }
}
```

- **Hàm hoàn chỉnh:**

```
void DeleteNode(Tree &root, <KD x)
{
    if (root != NULL)
    {
        if (root->info > x)
            DeleteNode(root->left, x);
        else if (root->info < x)
            DeleteNode(root->right, x);
        else
        {
            Node *p = root;
            if (root->left == NULL)
                root = root->right;
            else if (root->right == NULL)
                root = root->left;
            else
                FindAndReplaceMaxLeft(p, root->left); // Cach 1
            // FindAndReplaceMinRight(p, root->right); // Cach 2
            delete p;
        }
    }
}
```

5. **Bảng băm - Hash Table**

- **Định nghĩa:**
    - Bảng băm hay HashTable là một cấu trúc mà khi người dùng thực hiện truy xuất một phần tử qua khóa thì nó sẽ được ánh xạ vào thông qua hàm băm (Hash function).
    - **Hàm băm hay là Hash function** là hàm thực hiện việc **ánh xạ khóa k** nào đó vào trong bảng băm **(h(k))**. Một hàm băm tốt thỏa mản các tiêu chí sau:
        - Tốc độ tính toán nhanh.

        - Các khóa được phân bố đều trong bảng.
        - Ít xảy ra đụng độ.
    - Quá trình ánh xạ khóa vào bảng băm được thực hiện thông qua hàm băm (Hashing). Một bảng băm tốt cần phải có hàm băm tốt. Bảng băm là một mảng có M vị trí được đánh số từ 0 đến M - 1.
    - Giải quyết – xử lý va chạm (đụng độ):
        - Khái niệm đụng độ:
            - Đụng độ là hiện tượng các khóa khác nhau nhưng khi băm cho ra cùng địa chỉ như nhau.
            - Khi key1 # key2 mà f(key1) = f(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: Có hướng giải quyết: Các nút bị băm cùng địa chỉ (đụng độ/ xung đột) sẽ được gom thành một danh sách liên kết đơn.
            - Phương pháp băm lại:
                - Phương pháp dò tuyến tính.
                - Phương pháp dò bậc hai.

- **Cài đặt:**
- TH1: **Cài đặt theo hướng xử lý đụng độ bằng Seperate Chaining:**
- **Định nghĩa cấu trúc một nút trong bảng băm:**

```
struct Node
{
    int key;
    Node* pNext;
};
```

- **Định nghĩa cấu trúc bảng băm:**

```
#define M 100 // Thay 100 bang kich thuoc bat ky

typedef Node *HashTable[M];
```

```
// Su dung bang bam
HashTable mHashTable;
```

- **Hàm khởi tạo các node cho bảng băm:**

```
void InitHashTable(HashTable &HT)
{
    for (int i = 0; i < M; i++)
        HT[i] = NULL;
}
```

- **Định nghĩa hàm băm:**

```
int Hash(int k)
{
    return k % M;
}
```

- **Hàm tạo Node cho bảng băm:**

```
Node *CreateNode(int x)
{
    Node *p = new Node;
    if (p == NULL)
        return NULL;
    p->info = x;
    p->pNext = NULL;
    return p;
}
```

- **Hàm thêm một nút vào bảng băm:**
  - Để thêm một nút, ta cần xác định vị trí sẽ thêm qua hàm băm h(k), sau đó thêm vào danh sách liên kết ở vị trí h(k) đó. Việc đụng độ sẽ được giải quyết do nếu đụng độ thì khóa sẽ được tự thêm vào sau danh sách liên kết đơn.
  - **Hàm hoàn chỉnh:**

```
void AddTail(Node *&l, int k)
{
    Node *p = CreateNode(k);
    if (l == NULL)
    {
        l = p;
    }
```

```
    else
    {
        Node *q = l;
        while (q->pNext != NULL)
            q = q->pNext;
        q->pNext = p;
    }
}

void InsertNode(HashTable &HT, int x)
{
    int index = Hash(x);
    AddTail(HT[index], x);
}
```

- **Hàm tìm kiếm một khoá trong bảng băm:**

```
Node *SearchNode(HashTable HT, int x)
{
    int index = Hash(x);
    Node *p = HT[index];
    while (p != NULL)
    {
        if (p->info == x)
            return p;
        p = p->pNext;
    }
    return NULL;
}
```

- **Hàm xoá một nút trong bảng băm:**
    - Để xóa một phần tử ra khỏi bảng băm, đầu tiên ta cũng phải xác định h(k), sau đó tìm xem nó nằm ở đâu trong danh sách liên kết đơn tại vị trí h(k) đó rồi thực hiện xóa nó đi.
    - **Hàm hoàn chỉnh:**

```
void DeleteHead(Node *&l)
{
    if(l != NULL)
    {
        Node *p = l;
        l = l->pNext;
        delete p;
    }
}

void DeleteAfter(Node *&q)
{
    Node *p = q->pNext;
```

```
    if (p != NULL)
    {
        q->pNext = p->pNext;
        delete p;
    }
}

void DeleteNode(HashTable &HT, int x)
{
    int index = Hash(x);
    Node *p = HT[index];
    Node *q = p;
    while (p != NULL && p->info != x)
    {
        q = p;
        p = p->pNext;
    }
    if (p == HT[index])
        DeleteHead(HT[index]);
    else
        DeleteAfter(q);
}
```

- **Hàm in bảng băm:**

```
void PrintHashTable(HashTable HT)
{
    for (int i = 0; i < MAX; i++)
    {
        Node *p = HT[i];
        cout << "HT[" << i << "]: ";
        while (p != NULL)
        {
            cout << p->info << " ";
            p = p->pNext;
        }
        cout << endl;
    }
}
```

- **Hàm duyệt bảng băm:**

```
void Traverse(Node *p)
{
    while (p != NULL)
    {
        cout << p->key << ' ';
        p = p->next;
    }
    cout << endl;
}
```

```
void TraverseHashTable(HashTable HT)
{
    for (int i = 0; i < M; i++)
    {
        cout << "Bucket " << i << ": ";
        Traverse(HT[i]);
    }
}
```

- **TH2: Cài đặt theo hướng xử lý đụng độ bằng Linear Probing:**
- **Định nghĩa cấu trúc 1 nút cho bảng băm:**

```
struct Node
{
    int info;
};
typedef struct Node NODE;
```

- **Định nghĩa bảng băm:**
- Trong kỹ thuật xử lý va chạm này, chúng ta sẽ không dùng **danh sách liên kết** để lưu trữ mà chỉ có bản thân array đó thôi.

```
# define M 100 // Thay 100 bang kich thuoc khac

NODE *HT[M];
```

- **Hàm khởi tạo bảng băm:**

```
void CreateHashTable(NODE *HT[])
{
    for (int i = 0; i < MAX; i++)
        HT[i] = NULL;
}
```

- **Hàm băm:**
  - **Xem thêm các hàm băm cho string tại:**

  Hash table

```
int Hash(int x)
{
```

```
    return x % MAX;
}
```

- **Hàm tạo Node cho bảng băm:**

```
Node *CreateNode(int x)
{
    Node *p = new Node;
    if (p == NULL)
        return NULL;
    p->info = x;
    return p;
}
```

- **Hàm thêm node vào bảng băm:**

```
void InsertNode(NODE *HT[], int x)
{
    int index = Hash(x);
    NODE *p = CreateNode(x);
    while (HT[index] != NULL)
    {
        index = Hash(index + 1);
    }
    HT[index] = p;
}
```

- **Hàm tìm node trong bảng băm:**

```
NODE *SearchNode(NODE *HT[], int x)
{
    int index = Hash(x);
    while (HT[index]->info != x && HT[index]->info != NULL)
    {
        index = Hash(index + 1);
    }
    if (HT[index] != NULL)
        return HT[index];
    return NULL;
}
```

- **Hàm xoá node trong bảng băm:**

```
void DeleteNode(NODE *HT[], int x)
{
    int index = Hash(x);
    while (HT[index]->info != x && HT[index]->info != NULL)
```

```
    {
        index = Hash(index + 1);
    }
    if (HT[index]->info == x)
        HT[index] = NULL;
}
```

- **Hàm in bảng băm:**

```
void PrintHashTable(NODE *HT[])
{
    for (int i = 0; i < MAX; i++)
    {
        if (HT[i] != NULL)
            cout << i << ": " << HT[i]->info << endl;
        else
            cout << i << ": NULL" << endl;
    }
    cout << endl;
}
```

6. **Đồ thị - Graph**

- **Định nghĩa:**
  - **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.
    - **Lưu ý:**
      - Đề thi trong trường **chủ yếu** sẽ cho đồ thị vô hướng.
      - Biểu diễn đồ thị bằng danh sách kề sẽ tối ưu nhất cả về chi phí tính toán lẫn lưu trữ (được dùng phổ biến).
- **Biểu diễn đồ thị trên máy tính**:
  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.

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.
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).

- **Cài đặt:**
- **Biểu diễn đồ thị dưới dạng danh sách kề (Adjustency List) từ input dạng Danh sách cạnh:**

```
int n, m;              // n la so dinh, m la so canh
vector<int> adj[1001]; // Thay 1001 bang kich thuoc khac, tao 1001 vector<int>

void Input(vector<int> adj[])
{
    cin >> n >> m;
    for (int i = 0; i < m; i++)
    {
        int x, y;
        cin >> x >> y;
        adj[x].push_back(y);
        adj[y].push_back(x); // Do thi vo huong
    }
}
```

- **Biểu diễn đồ thị dưới dạng danh sách kề (Adjustency List) từ input dạng Ma trận kề:**

```
int n;                  // n la so dinh
int matrix[1001][1001]; //  ma tran ke
vector<int> adj[1001];  // Thay 1001 bang kich thuoc khac
```

```
void Input(int a[][1001], vector<int> adj[])
{
    cin >> n;
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= n; j++)
        {
            cin >> a[i][j];
        }
    }
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= n; j++)
            if (a[i][j] == 1)
                adj[i].push_back(j);
    }
}
```

- **Biểu diễn đồ thị dạng danh sách kề (Adjustency List) từ input dạng danh sách kề:**

```
void InputFromAdjList(vector<int> adj[], int n)
{
    int u, v;
    for (int i = 1; i <= n; i++)
    {
        cin >> u;
        while (cin >> v)
        {
            adj[u].push_back(v);
            adj[v].push_back(u); // Do thi vo huong
        }
    }
}
```

- **Hàm in danh sách kề của đồ thị:**

```
void Print(vector<int> adj[])
{
    for (int i = 1; i <= n; i++)
    {
        cout << i << ": ";
        for (int j = 0; j < adj[i].size(); j++)
            cout << adj[i][j] << " ";
        cout << endl;
    }
}
```

- **Hàm tìm kiếm theo chiều sâu để duyệt từ một đỉnh đến các đỉnh còn lại - Depth First Search (DFS)**:

```
int n, m;              // n la so dinh, m la so canh
bool visited[1001];    // Thay 1001 bang kich thuoc khac
vector<int> adj[1001]; // Thay 1001 bang kich thuoc khac, tao 1001 vector<int>

void Input(vector<int> adj[], bool visited[])
{
    cin >> n >> m;
    for (int i = 0; i < m; i++)
    {
        int x, y;
        cin >> x >> y;
        adj[x].push_back(y);
        adj[y].push_back(x);
    }
    memset(visited, false, sizeof(visited));
}

void DFS(vector<int> adj[], bool visited[], int u)
{
    visited[u] = true;
    cout << u << " ";
    for (int i = 0; i < adj[u].size(); i++)
    {
        int v = adj[u][i];
        if (visited[v] == false)
            DFS(v);
    }
}
```

- **Hàm tìm kiếm theo chiều rộng để duyệt từ một đỉnh đến các đỉnh còn lại - Breadth First Search (BFS)**:

```
int n, m;              // n la so dinh, m la so canh
bool visited[1001];    // Thay 1001 bang kich thuoc khac
vector<int> adj[1001]; // Thay 1001 bang kich thuoc khac, tao 1001 vector<int>

void Input(vector<int> adj[], bool visited[])
{
    cin >> n >> m;
    for (int i = 0; i < m; i++)
    {
        int x, y;
        cin >> x >> y;
        adj[x].push_back(y);
        adj[y].push_back(x);
    }
    memset(visited, false, sizeof(visited));
}

void BFS(vector<int> adj[], bool visited[], int u)
{
    queue<int> q;
    q.push(u);
```

```
    visited[u] = true;
    while (q.empty() == false)
    {
        int v = q.front();
        q.pop();
        cout << v << " ";
        for (int i = 0; i < adj[v].size(); i++)
        {
            int x = adj[v][i];
            if (visited[x] == false)
            {
                q.push(x);
                visited[x] = true;
            }
        }
    }
}
```

- **Hàm tìm đường đi giữa 2 đỉnh bất kì của đồ thị bằng DFS:**

```
vector<int> adj[1001];
int a[1001][1001];
bool visited[1001];
int parent[1001];
memset(parent, -1, sizeof(parent));
memset(visited, false, sizeof(visited));

void DFSPath(int u, vector<int> adj[], bool visited[], int parent[])
{
    visited[u] = true;
    for (int i = 0; i < adj[u].size(); i++)
    {
        int v = adj[u][i];
        if (visited[v] == false)
        {
            parent[v] = u;
            DFS_Path(v, adj, visited, parent);
        }
    }
}

// Ham in ra duong di (truy vet) giua 2 dinh
void FindPath(int u, int v, vector<int> adj[], bool visited[], int parent[])
{
    DFSPath(u, adj, visited, parent);
    if (parent[v] == -1)
        cout << "Khong co duong di";
    else
    {
        vector<int> path;
        while (v != u)
        {
            path.push_back(v);
            v = parent[v];
        }
```

```
        path.push_back(u);
        reverse(path.begin(), path.end());
        for (int i = 0; i < path.size(); i++)
            cout << path[i] << " ";
    }
}
```

- **Hàm tìm đường đi giữa 2 đỉnh bất kì bằng BFS:**

```
vector<int> adj[1001];
int a[1001][1001];
bool visited[1001];
int parent[1001];
memset(parent, -1, sizeof(parent));
memset(visited, false, sizeof(visited));

void BFSPath(int u, vector<int> adj[], bool visited[], int parent[])
{
    queue<int> q;
    q.push(u);
    visited[u] = true;
    while (q.empty() == false)
    {
        int v = q.front();
        q.pop();
        for (int i = 0; i < adj[v].size(); i++)
        {
            int x = adj[v][i];
            if (visited[x] == false)
            {
                q.push(x);
                visited[x] = true;
                parent[x] = v;
            }
        }
    }
}

// Ham in ra duong di (truy vet) giua 2 dinh
void FindPath(int u, int v, vector<int> adj[], bool visited[], int parent[])
{
    DFSPath(u, adj, visited, parent);
    if (parent[v] == -1)
        cout << "Khong co duong di";
    else
    {
        vector<int> path;
        while (v != u)
        {
            path.push_back(v);
            v = parent[v];
        }
        path.push_back(u);
        reverse(path.begin(), path.end());
        for (int i = 0; i < path.size(); i++)
```

```
            cout << path[i] << " ";
    }
}
```

- **Hàm duyệt đồ thị có trọng số bằng thuật toán Dijstra:**

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

using ll = long long;
const int INF = 1e9;
const int MAX = 1001;

int n,m; // n la so dinh, m la so canh
vector<pair<int, int>> adj[MAX]; // Adjacency list

void Input(vector<pair<int, int>> adj[], int m, int parent[])
{
    int u, v, w;
    for (int i = 0; i < m; i++)
    {
        cin >> u >> v >> w;
        adj[u].push_back(make_pair(v, w));
        adj[v].push_back(make_pair(u, w)); // Do thi vo huong
    }
}

void Dijstra(vector<pair<int, int>> adj[], int n, int s)
{
    vector<ll> d(n + 1, INF);
    d[s] = 0;
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> Q;
// Min heap {Khoang cach, dinh}
    Q.push(make_pair(0, s));
// Khoang cach tu s den s la 0
    // Duyet cac dinh
    while (Q.empty() == false)
    {
        // Chon khoang cach tu s den cac dinh la nho nhat
        pair<int, int> top = Q.top();
        Q.pop();
        int u = top.second; // Dinh
        int du = top.first; // Khoang cach tu s den u
        if (du > d[u])
            continue;
        // Relaxation: Cap nhat khoang cach tu s den cac dinh ke cua u
        for (int i = 0; i < adj[u].size(); i++)
        {
            pair<int, int> neighbor = adj[u][i];
            int v = neighbor.first;  // Dinh ke cua u
            int w = neighbor.second; // Khoang cach tu u den v
            if (d[v] > d[u] + w)
            {
```

```
                d[v] = d[u] + w;
                Q.push(make_pair(d[v], v));
            }
        }
    }
    // In ra khoang cach tu s den cac dinh
    for (int i = 1; i <= n; i++)
        cout << d[i] << " ";
}
```

- **Hàm tìm đường đi ngắn nhất trong đồ thị có trọng số bằng thuật toán Dijstra:**

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

using ll = long long;
const int INF = 1e9;
const int MAX = 1001;

int n,m; // n la so dinh, m la so canh
vector<pair<int, int>> adj[MAX]; // Adjacency list
int parent[MAX];
memset(parent, -1, sizeof(parent));

void Input(vector<pair<int, int>> adj[], int m)
{
    int u, v, w;
    for (int i = 0; i < m; i++)
    {
        cin >> u >> v >> w;
        adj[u].push_back(make_pair(v, w));
        adj[v].push_back(make_pair(u, w)); // Do thi vo huong
    }
}

void DijstraPath(vector<pair<int, int>> adj[], int n, int s, int d, int parent[])
{
    vector<ll> d(n + 1, INF);
    d[s] = 0;
    parent[s] = s;
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> Q;
// Min heap {Khoang cach, dinh}
    Q.push(make_pair(0, s));
// Khoang cach tu s den s la 0
    // Duyet cac dinh
    while (Q.empty() == false)
    {
        // Chon khoang cach tu s den cac dinh la nho nhat
        pair<int, int> top = Q.top();
        Q.pop();
        int u = top.second; // Dinh
```

```
        int du = top.first; // Khoang cach tu s den u
        if (du > d[u])
            continue;
        // Relaxation: Cap nhat khoang cach tu s den cac dinh ke cua u
        for (int i = 0; i < adj[u].size(); i++)
        {
            pair<int, int> neighbor = adj[u][i];
            int v = neighbor.first;  // Dinh ke cua u
            int w = neighbor.second; // Khoang cach tu u den v
            if (d[v] > d[u] + w)
            {
                d[v] = d[u] + w;
                Q.push(make_pair(d[v], v));
                parent[v] = u;
            }
        }
    }
}

void FindPath(vector < pair<int, int> adj[], int n, int s, int d, int parent[])
{
    DijstraPath(adj, n, s, d, parent);
    vector<int> path;
    int u = d;
    if (parent[u] == -1)
    {
        cout << "Khong co duong di";
        return;
    }
    else
    {
        while (u != s)
        {
            path.push_back(u);
            u = parent[u];
        }
        path.push_back(s);
        reverse(path.begin(), path.end());
        for (int i = 0; i < path.size(); i++)
            cout << path[i] << " ";
    }
}
```

- **Hàm đếm số thành phần liên thông của đồ thị bằng DFS hoặc BFS:**

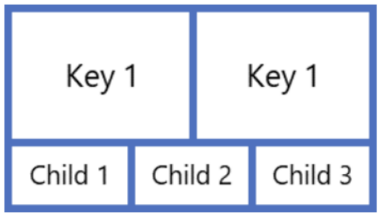
```
int n, m;              // n la so dinh, m la so canh
bool visited[1001];    // Thay 1001 bang kich thuoc khac
vector<int> adj[1001]; // Thay 1001 bang kich thuoc khac, tao 1001 vector<int>
memset(visited, false, sizeof(visited));

void Input(vector<int> adj[], bool visited[])
{
    cin >> n >> m;
    for (int i = 0; i < m; i++)
    {
```

```
        int x, y;
        cin >> x >> y;
        adj[x].push_back(y);
        adj[y].push_back(x);
    }
}

void DFS(vector<int> adj[], bool visited[], int u)
{
    visited[u] = true;
    cout << u << " ";
    for (int i = 0; i < adj[u].size(); i++)
    {
        int v = adj[u][i];
        if (visited[v] == false)
            DFS(v);
    }
}

void BFS(vector<int> adj[], bool visited[], int u)
{
    queue<int> q;
    q.push(u);
    visited[u] = true;
    while (q.empty() == false)
    {
        int v = q.front();
        q.pop();
        cout << v << " ";
        for (int i = 0; i < adj[v].size(); i++)
        {
            int x = adj[v][i];
            if (visited[x] == false)
            {
                q.push(x);
                visited[x] = true;
            }
        }
    }
}

int CountConnectedComponents(vector<int> adj[], int n, bool visited[])
{
    int count = 0;
    for (int i = 1; i <= n; i++)
    {
        if (visited[i] == false)
        {
            ++count;
            BFS(adj, visited, i); // DFS(adj, visited, i);
        }
    }
    return count;
}

bool IsConnected(vector<int> adj[], int n, bool visited[])
{
    int count = CountConnectedComponents(adj, n, visited);
```

```
    if (count == 1)
        return true;
    return false;
}
```

7. **B - Tree (Cây B hoặc cây m-phân)**

- **Định nghĩa:**
- **Một nút**:
  - Gồm **khoá và các cây con.**
  - **Số cây con = Số khoá + 1;**



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

  - **Quy tắc cần nhớ:**

TÓM LẠI, cho m ≥ 3 là bậc của cây B-tree, ta có:

[m/2]	≤	**số node con** của mỗi node (khác root và lá)	≤ m
2	≤	**số node con** của root (khác lá)	≤ m
[m/2]-1	≤	**số khóa** của mỗi node khác root	≤ m-1
1	≤	**số khóa** của node root	≤ m-1

  - **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).

- **Thao tác tách Nút (split node):**
  - 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 mớ**i, **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.**
- **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 n**hường 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.**
- **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.**

- **Tính toán (Bắt buộc phải làm):**
  - **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 (Chương trình UIT chỉ xét cây B-Tree bậc lẻ).**
    - **Số khóa tối đa = m - 1**
    - **Số khóa tối thiểu = Số cây con tối thiểu – 1**
- **Thuật toán thêm khoá cho B - Tree:**
  - 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.**
- **Thuật toán xoá khoá:**
  - **Xóa khóa tại mọi nút được quy đổi về xóa khóa trong một nút lá**. V**ì 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:**
  - **TH1:** Nếu một nút kề nút vừa mất khóa có nhiều hơn **k** khóa, chuyển khóa đó sang cho nút vừa xóa đi 1 khóa.
  - **TH2:** 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.

FULL CODE TẠI:

https://github.com/crazyads69/DSA