**Sắp xếp nổi bọt (Bubble sort)**

Đây là thuật toán cơ bản nhất cho việc sắp xếp.

**Ý tưởng**

* Xét lần lượt các cặp 2 phần tử liên tiếp. Nếu phần tử đứng sau nhỏ hơn phần tử đứng trước, ta đổi chỗ 2 phần tử. Nói cách khác, phần tử nhỏ nhất sẽ **nổi** lên trên.
* Lặp lại đến khi không còn 2 phần tử nào thỏa mãn. Có thể chứng minh được số lần lặp không quá N−1, do một phần tử chỉ có thể **nổi** lên trên không quá N−1 lần.

**Ưu điểm**

* Code đơn giản, dễ hiểu
* Không tốn thêm bộ nhớ

**Nhược điểm**

* Độ phức tạp O(N2), không đủ nhanh với dữ liệu lớn.

**Code**

**for** (**int** i **=** 0; i **<** n; i**++**)

**for** (**int** j **=** 0; j **<** n **-** 1; j**++**)

**if** (a[j] **>** a[j**+**1]) {

swap(a[j], a[j**+**1]);

}

**Sắp xếp chèn (Insertion Sort)**

**Ý tưởng**

Ý tưởng chính của thuật toán là ta sẽ sắp xếp lần lượt từng đoạn gồm 1 phần tử đầu tiên, 2 phần tử đầu tiên, …, NN phần tử.

Giả sử ta đã sắp xếp xong ii phần tử của mảng. Để sắp xếp i+1 phần tử đầu tiên, ta tìm vị trí phù hợp của phần tử thứ i+1 và "chèn" nó vào đó.

**Ưu điểm**

* Nếu danh sách đã gần đúng thứ tự, Insertion Sort sẽ chạy rất nhanh. Ví dụ bạn cần sắp xếp Highscore trong game.

**Nhược điểm**

* Độ phức tạp O(N2), không đủ nhanh với dữ liệu lớn.

**Code**

**for** (**int** i **=** 1; i **<** n; i**++**) {

*// Tìm vị trí phù hợp cho i*

**int** j **=** i;

**while** (j **>** 0 **&&** data[i] **<** data[j**-**1]) **--**j;

*// Đưa i về đúng vị trí*

**int** tmp **=** data[i];

**for** (**int** k **=** i; k **>** j; k**--**)

data[k] **=** data[k**-**1];

data[j] **=** tmp;

}

**Sắp xếp vun đống (HeapSort)**

**Ý tưởng**

Ta lưu mảng vào CTDL [Heap](https://vnoi.info/wiki/translate/wcipeg/Binary-Heap).

Ở mỗi bước, ta lấy ra phần tử nhỏ nhất trong heap, cho vào mảng đã sắp xếp.

**Ưu điểm**

* Cài đặt đơn giản nếu đã có sẵn thư viện Heap.
* Chạy nhanh, độ phức tạp O(N∗logN).

**Nhược điểm**

* Không ổn định

**Code**

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

{

int largest = i; // Initialize largest as root

int l = 2\*i + 1; // left = 2\*i + 1

int r = 2\*i + 2; // right = 2\*i + 2

// If left child is larger than root

if (l < n && arr[l] > arr[largest])

largest = l;

// If right child is larger than largest so far

if (r < n && arr[r] > arr[largest])

largest = r;

// If largest is not root

if (largest != i)

{

swap(arr[i], arr[largest]);

// Recursively heapify the affected sub-tree

heapify(arr, n, largest);

}

}

// main function to do heap sort

void heapSort(int arr[], int n)

{

// Build heap (rearrange array)

for (int i = n / 2 - 1; i >= 0; i--)

heapify(arr, n, i);

// One by one extract an element from heap

for (int i=n-1; i>0; i--)

{

// Move current root to end

swap(arr[0], arr[i]);

// call max heapify on the reduced heap

heapify(arr, i, 0);

}

}

**Sắp xếp trộn (Merge sort)**

**Ý tưởng**

Sắp xếp trộn hoạt động kiểu đệ quy:

* Đầu tiên chia dữ liệu thành 2 phần, và sắp xếp từng phần.
* Sau đó gộp 2 phần lại với nhau. Để gộp 2 phần, ta làm như sau:
  + Tạo một dãy AA mới để chứa các phần tử đã sắp xếp.
  + So sánh 2 phần tử đầu tiên của 2 phần. Phần tử nhỏ hơn ta cho vào AA và xóa khỏi phần tương ứng.
  + Tiếp tục như vậy đến khi ta cho hết các phần tử vào dãy AA.

**Ưu điểm**

* Chạy nhanh, độ phức tạp O(N∗logN)O(N∗logN).
* Ổn định

**Nhược điểm**

* Cần dùng thêm bộ nhớ để lưu mảng A.

**Code**

**int** a[MAXN]; *// mảng trung gian cho việc sắp xếp*

*// Sắp xếp các phần tử có chỉ số từ left đến right của mảng data.*

**void** **mergeSort**(**int** data[MAXN], **int** left, **int** right) {

**if** (data.length **==** 1) {

*// Dãy chỉ gồm 1 phần tử, ta không cần sắp xếp.*

**return** ;

}

**int** mid **=** (left **+** right) **/** 2;

*// Sắp xếp 2 phần*

mergeSort(data, left, mid);

mergeSort(data, mid**+**1, right);

*// Trộn 2 phần đã sắp xếp lại*

**int** i **=** left, j **=** mid **+** 1; *// phần tử đang xét của mỗi nửa*

**int** cur **=** 0; *// chỉ số trên mảng a*

**while** (i **<=** mid **||** j **<=** right) { *// chừng nào còn 1 phần chưa hết phần tử.*

**if** (i **>** mid) {

*// bên trái không còn phần tử nào*

a[cur**++**] **=** data[j**++**];

} **else** **if** (j **>** right) {

*// bên phải không còn phần tử nào*

a[cur**++**] **=** data[i**++**];

} **else** **if** (data[i] **<** data[j]) {

*// phần tử bên trái nhỏ hơn*

a[cur**++**] **=** data[i**++**];

} **else** {

a[cur**++**] **=** data[j**++**];

}

}

*// copy mảng a về mảng data*

**for** (**int** i **=** 0; i **<** cur; i**++**)

data[left **+** i] **=** a[i];

}

**Sắp xếp cơ số (RadixSort)**

**Ý tưởng**

Khác với tất cả các thuật toán nêu trên, RadixSort không sử dụng việc so sánh 2 phần tử.

* Đầu tiên, thuật toán sẽ chia các phần tử thành các nhóm, dựa trên chữ số cuối cùng (hoặc dựa theo bit cuối cùng, hoặc vài bit cuối cùng).
* Sau đó ta đưa các nhóm lại với nhau, và được danh sách sắp xếp theo chữ số cuối của các phần tử. Quá trình này lặp đi lặp lại với chữ số át cuối cho tới khi tất cả vị trí chữ số đã sắp xếp.

**Ưu điểm**

* Có thể chạy nhanh hơn các thuật toán sắp xếp sử dụng so sánh. Ví dụ nếu ta sắp xếp các số nguyên 32 bit, và chia nhóm theo 1 bit, thì độ phức tạp là O(N)O(N). Trong trường hợp tổng quát, độ phức tạp là O(N∗log(max(ai)))

**Nhược điểm**

* Không thể sắp xếp số thực.

## Code

**int** getMax(**int** arr[], **int** n)

{

**int** mx = arr[0];

**for** (**int** i = 1; i < n; i++)

**if** (arr[i] > mx)

            mx = arr[i];

**return** mx;

}

**void** countSort(**int** arr[], **int** n, **int** **exp**)

{

**int** output[n]; // output array

**int** i, count[10] = { 0 };

    // Store count of occurrences in count[]

**for** (i = 0; i < n; i++)

        count[(arr[i] / **exp**) % 10]++;

    // Change count[i] so that count[i] now contains actual

    //  position of this digit in output[]

**for** (i = 1; i < 10; i++)

        count[i] += count[i - 1];

    // Build the output array

**for** (i = n - 1; i >= 0; i--) {

        output[count[(arr[i] / **exp**) % 10] - 1] = arr[i];

        count[(arr[i] / **exp**) % 10]--;

**for** (i = 0; i < n; i++)

        arr[i] = output[i];

}

**void** radixsort(**int** arr[], **int** n)

{

    // Find the maximum number to know number of digits

**int** m = getMax(arr, n);

    // Do counting sort for every digit. Note that instead

    // of passing digit number, exp is passed. exp is 10^i

    // where i is current digit number

**for** (**int** **exp** = 1; m / **exp** > 0; **exp** \*= 10)

        countSort(arr, n, **exp**);

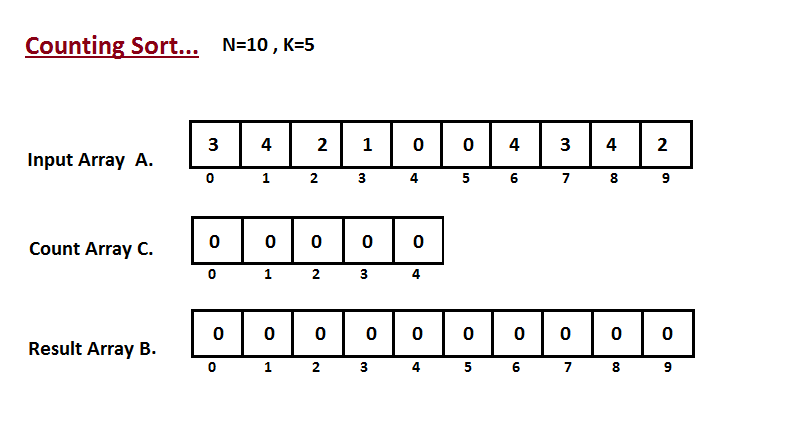
}

Ý tưởng của Counting sort

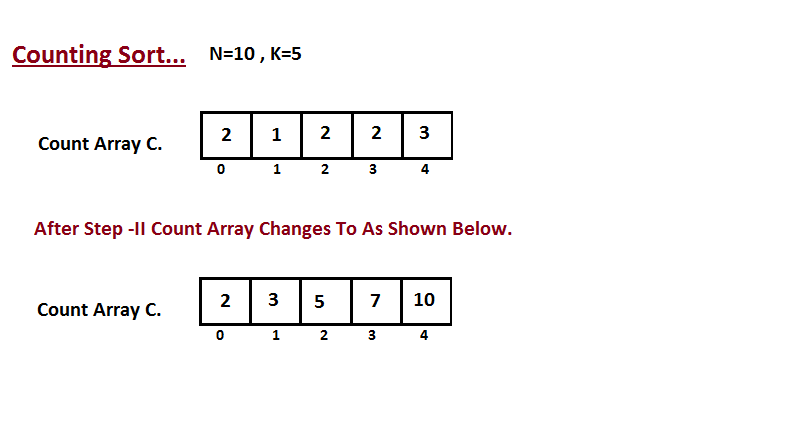
Hình ảnh dưới đây cho chúng ta thấy cách hoạt động của thuật toán sắp xếp này.

**Bước 1:**

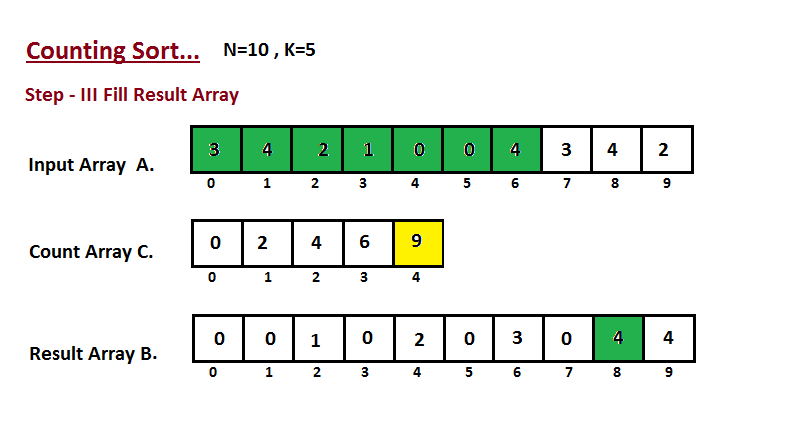
Trong bước đầu tiên, chúng tôi đếm số lần xuất hiện của từng phần tử trong mảng cần sắp xếp A. Kết quả được lưu vào mảng C.



**Bước 2:**Ở bước này, chúng ta cần xem xét sửa đổi giá trị của C. C[i] thể hiện giới hạn trên của chỉ số của phần tử i sau khi sắp xếp.



**Bước 3:**Duyệt qua từng phần tử của A và đặt nó vào đúng chỉ số của mảng chứa các giá trị đã sắp xếp B dựa vào C.



# Cây đỏ đen

## Định nghĩa

Cây đỏ đen là một cây nhị phân tìm kiếm (BST) tuân thủ các quy tắc sau: (hình 2)

(1) Mọi node phải là đỏ hoặc đen.

(2) Node gốc và các node lá (NIL) phải luôn luôn đen.

(3) Nếu một node là đỏ, những node con của nó phải đen.

(4) Mọi đường dẫn từ gốc đến một lá phải có cùng số lượng node đen.

Khi chèn (hay xóa) một node mới, cần phải tuân thủ các quy tắc trên -gọi là quy tắc đỏ đen. Nếu được tuân thủ, cây sẽ được cân bằng.

## Nhận xét

Cây đỏ-đen với n nút trong có chiều cao tối đa là 2log2(n+1).

Các nút là NIL luôn là nút đen.

Một cạnh dẫn đến một nút đen gọi là cạnh đen (black edge).

Nếu không tồn tại nút cha hay nút con của một nút thì những con trỏ tương ứng sẽ trở về NIL.

Một lính canh (sent) sẽ đại diện cho tất cả các nút NIL.

Code Insert: Text

Description automatically generated

Text, letter

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Diagram

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Text

Description automatically generated

**Trường hợp 1**

**Trường hợp 1:** **N** là gốc mới. Trong trường hợp này chúng ta dừng lại. Ta đã giải phóng một nút đen khỏi mọi đường đi và gôc mới lại là đen. Không tính chất nào bị vi phạm.

void delete\_case1(struct node \*n) {

if (n->parent == NULL)

return;

else

delete\_case2(n);

}

*Chú ý*: Trong các trường hợp 2, 5, và 6, ta quy ước **N** là con trái của cha **P**. Nếu no là con phải, *left* và *right* sẽ tráo đổi cho nhau. Tuy nhiên code ví dụ làm cho cả hai trường hợp.

**Trường hợp 2**

|  |
| --- |
| [Diagram of case 2](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:Red-black_tree_delete_case_2.png)  **Trường hợp 2:** **S** là đỏ. Trong trường hợp này tráo đổi màu của **P** và **S**, và sau đó [quay trái](https://vi.wikipedia.org/w/index.php?title=Ph%C3%A9p_quay&action=edit&redlink=1) tai **P**, nó sẽ làm cho **S** trở thành nút ông của **N**. Chú ý rằng **P** có màu đen và có một con màu đỏ. Tất cả các đường đi có số các nút đen giống nhau, bây giờ **N** có một anh em màu đen và cha màu đỏ, chúng ta có thể tiếp tục với các trường hợp 4, 5, hoặc 6. (anh em mới của nó là đen ví chỉ có một con của nút đỏ **S**.) Trong các trường hợp sau la sẽ gọi anh em mới của **N'** là **S**. |

void delete\_case2(struct node \*n) {

if (sibling(n)->color == RED) {

n->parent->color = RED;

sibling(n)->color = BLACK;

if (n == n->parent->left)

rotate\_left(n->parent);

else

rotate\_right(n->parent);

}

delete\_case3(n);

}

**Trường hợp 3**

|  |
| --- |
| [Diagram of case 3](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:Red-black_tree_delete_case_3.png)  **Trường hợp 3:** **P**, **S**, và các con của **S** là đen. Trong trường hợp này, chúng ta gán lại cho **S** màu đỏ. Kết quả là mọi đường đi qua **S**, (tất nhiên chúng không qua **N**,có ít hơn một nút đen. Vì việc xóa đi cha trước đây của **N'** làm tất cả các đương đi qua **N** bớt đi một nút đen, nên chúng bằng nhau. Tuy nhiên tất cả các đường đi qua **P** bây giờ có ít hơn một nút đen so với các đường không qua **P**, do đó Tính chất 5 (Tất cả các đường đi từ gốc tới các nút lá có cùng số nút đen) sẽ bị vi phạm. Để sửa chữa nó chúng ta lại tái cân bằng tại **P**, bắt đầu từ trường hợp 1. |

void delete\_case3(struct node \*n) {

if (n->parent->color == BLACK &&

sibling(n)->color == BLACK &&

sibling(n)->left->color == BLACK &&

sibling(n)->right->color == BLACK)

{

sibling(n)->color = RED;

delete\_case1(n->parent);

}

else

delete\_case4(n);

}

**Trường hợp 4**[[sửa](https://vi.wikipedia.org/w/index.php?title=C%C3%A2y_%C4%91%E1%BB%8F_%C4%91en&veaction=edit&section=10) | [sửa mã nguồn](https://vi.wikipedia.org/w/index.php?title=C%C3%A2y_%C4%91%E1%BB%8F_%C4%91en&action=edit&section=10)]

**Trường hợp 4:** **S** và các con của **S** là đen nhưng **P** là đỏ. Trong trường hợp này, chúng ta đổi ngược màu của **S** và **P**. Điều này không ảnh hưởng tới số nút đen trên các đường đi không qua **N**, nhưng thêm một nút đen trên các đường đi qua **N**, thay cho nút đen đã bị xóa trên các đường này.

|  |
| --- |
|  |

void delete\_case4(struct node \*n) {

if (n->parent->color == RED &&

sibling(n)->color == BLACK &&

sibling(n)->left->color == BLACK &&

sibling(n)->right->color == BLACK)

{

sibling(n)->color = RED;

n->parent->color = BLACK;

}

else

delete\_case5(n);

}

**Trường hợp 5**[[sửa](https://vi.wikipedia.org/w/index.php?title=C%C3%A2y_%C4%91%E1%BB%8F_%C4%91en&veaction=edit&section=11) | [sửa mã nguồn](https://vi.wikipedia.org/w/index.php?title=C%C3%A2y_%C4%91%E1%BB%8F_%C4%91en&action=edit&section=11)]

[](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:Red-black_tree_delete_case_5.png)

**Trường hợp 5:** **S** là đen, con trái của **S** là đỏ, con phải của **S** là đen, còn **N** là con trái của cha nó. Trong trường hợp này chúng ta quay phải tại **S**, khi đó con trái của **S** trở thành cha của **S** và **N** là anh em mới của nó. Sau đó ta tráo đổi màu của **S** và cha mới của nó. Tất cả các đường đi sẽ có số nút đen như nhau, nhưng bây giờ **N** có một người anh em đen mà con phải của nó lại là đỏ, chúng ta chuyển sang Trường hợp 6. Hoặc **N** hoặc cha của nó bị tác động bởi việc dịch chuyên này.

(Lưu ý trong trường hợp 6, ta đặt lại nút anh em mới của **N** là **S**.)

|  |
| --- |
|  |

void delete\_case5(struct node \*n) {

if (n == n->parent->left &&

sibling(n)->color == BLACK &&

sibling(n)->left->color == RED &&

sibling(n)->right->color == BLACK)

{

sibling(n)->color = RED;

sibling(n)->left->color = BLACK;

rotate\_right(sibling(n));

}

else if (n == n->parent->right &&

sibling(n)->color == BLACK &&

sibling(n)->right->color == RED &&

sibling(n)->left->color == BLACK)

{

sibling(n)->color = RED;

sibling(n)->right->color = BLACK;

rotate\_left(sibling(n));

}

delete\_case6(n);

}

**Trường hợp 6**[[sửa](https://vi.wikipedia.org/w/index.php?title=C%C3%A2y_%C4%91%E1%BB%8F_%C4%91en&veaction=edit&section=12) | [sửa mã nguồn](https://vi.wikipedia.org/w/index.php?title=C%C3%A2y_%C4%91%E1%BB%8F_%C4%91en&action=edit&section=12)]

|  |
| --- |
| [Diagram of case 6](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:Red-black_tree_delete_case_6.png)  **Trường hợp 6:** **S** là đen, con phải của **S** là đỏ và **N** là con trái của nút cha **P**. Trong trường hợp này chúng ta quay trái tại **P**, khi đó **S** trở thành cha của **P** và con phải của **S**. Chúng ta hoán đổi màu của **P** và **S**, và gán cho con phải của **S** màu đen. Cây con giữ nguyên màu của gốc do đó Tính chất 4 (Cả hai con của nút đỏ là đen) và Tính chất 5 không bị vi phạm trong cây con này. Tuy nhiên, **N** bây giờ có thêm một nút đen tiền nhiệm: hoặc **P** mới bị tô đen, nó đã là đen và **S** là nút ông của nó trở thành đen. Như cậy các đương đi qua **N** có thêm một nút đen.  Trong lúc đó, với một đường đi không đi qua **N**, có hai khả năng:   * đi qua nút anh em của **N**. Khi đó cả trước và sau khi quay nó phải đi qua **S** và **P**, khi thay đổi màu sắc hai nút này đã tráo đổi màu cho nhau. Như vây đường đi này không bị thay đổi số nút đen. * đi qua nút bác của **N'**, là con phải của **S**. Khi đó trước khi quay nó đi qua **S**, cha của **S**, và con phải của **S**, nhưng sau khi quay nó chỉ đi qua nút **S** và con phải của **S**, khi này **S** đã nhận màu cũ của cha **P** còn con phải của **S'**s đã đổi màu từ đỏ thành đen. Kết quả là số các nút đen trên đường đi này không thay đổi.   Như vậy, số các nút đen trên các đường đi là không thay đổi. Do đó các tính chất 4 và 5 đã được khôi phục. Nút trắng trong hình vẽ có thể là đỏ hoặc đen, nhưng phải ghi lại trước và sau khi thay đổi. |

void delete\_case6(struct node \*n) {

sibling(n)->color = n->parent->color;

n->parent->color = BLACK;

if (n == n->parent->left) {

/\* Here, sibling(n)->right->color == RED \*/

sibling(n)->right->color = BLACK;

rotate\_left(n->parent);

}

else

{

/\* Here, sibling(n)->left->color == RED \*/

sibling(n)->left->color = BLACK;

rotate\_right(n->parent);

}

}

Nhắc lại rằng các hàm này có lời gọi đệ quy. Thêm nữa lời gọi không đệ quy sẽ được goi sau một phép quay, do đó số lần thực hiện các phép quay là không đổi (không quá 3).

# B-cây

A picture containing timeline

Description automatically generated

A picture containing text, person, screenshot

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

# Cây AVL

Node \*rightRotate(Node \*y) {

Node \*x = y->left;

Node \*T2 = x->right;

x->right = y;

y->left = T2;

y->height = max(height(y->left),

height(y->right)) +

1;

x->height = max(height(x->left),

height(x->right)) +

1;

return x;

}

// Rotate left

Node \*leftRotate(Node \*x) {

Node \*y = x->right;

Node \*T2 = y->left;

y->left = x;

x->right = T2;

x->height = max(height(x->left),

height(x->right)) +

1;

y->height = max(height(y->left),

height(y->right)) +

1;

return y;

}

// Get the balance factor of each node

int getBalanceFactor(Node \*N) {

if (N == NULL)

return 0;

return height(N->left) -

height(N->right);

}

// Insert a node

Node \*insertNode(Node \*node, int key) {

// Find the correct postion and insert the node

if (node == NULL)

return (newNode(key));

if (key < node->key)

node->left = insertNode(node->left, key);

else if (key > node->key)

node->right = insertNode(node->right, key);

else

return node;

// Update the balance factor of each node and

// balance the tree

node->height = 1 + max(height(node->left),

height(node->right));

int balanceFactor = getBalanceFactor(node);

if (balanceFactor > 1) {

if (key < node->left->key) {

return rightRotate(node);

} else if (key > node->left->key) {

node->left = leftRotate(node->left);

return rightRotate(node);

}

}

if (balanceFactor < -1) {

if (key > node->right->key) {

return leftRotate(node);

} else if (key < node->right->key) {

node->right = rightRotate(node->right);

return leftRotate(node);

}

}

return node;

}

// Node with minimum value

Node \*nodeWithMimumValue(Node \*node) {

Node \*current = node;

while (current->left != NULL)

current = current->left;

return current;

}

// Delete a node

Node \*deleteNode(Node \*root, int key) {

// Find the node and delete it

if (root == NULL)

return root;

if (key < root->key)

root->left = deleteNode(root->left, key);

else if (key > root->key)

root->right = deleteNode(root->right, key);

else {

if ((root->left == NULL) ||

(root->right == NULL)) {

Node \*temp = root->left ? root->left : root->right;

if (temp == NULL) {

temp = root;

root = NULL;

} else

\*root = \*temp;

free(temp);

} else {

Node \*temp = nodeWithMimumValue(root->right);

root->key = temp->key;

root->right = deleteNode(root->right,

temp->key);

}

}

if (root == NULL)

return root;

// Update the balance factor of each node and

// balance the tree

root->height = 1 + max(height(root->left),

height(root->right));

int balanceFactor = getBalanceFactor(root);

if (balanceFactor > 1) {

if (getBalanceFactor(root->left) >= 0) {

return rightRotate(root);

} else {

root->left = leftRotate(root->left);

return rightRotate(root);

}

}

if (balanceFactor < -1) {

if (getBalanceFactor(root->right) <= 0) {

return leftRotate(root);

} else {

root->right = rightRotate(root->right);

return leftRotate(root);

}

}

return root;

}

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Data Structure | Time Complexity | | | | | | | | | Space Complexity | |
|  | Average | | | | Worst | | | | | Worst | |
|  | Access | Search | Insertion | Deletion | Access | Search | Insertion | Deletion | |  | |
| [Array](http://en.wikipedia.org/wiki/Array_data_structure) | Θ(1) | Θ(n) | Θ(n) | Θ(n) | O(1) | O(n) | O(n) | O(n) | O(n) | |
| [Stack](http://en.wikipedia.org/wiki/Stack_(abstract_data_type)) | Θ(n) | Θ(n) | Θ(1) | Θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) | |
| [Queue](http://en.wikipedia.org/wiki/Queue_(abstract_data_type)) | Θ(n) | Θ(n) | Θ(1) | Θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) | |
| [Singly-Linked List](http://en.wikipedia.org/wiki/Singly_linked_list#Singly_linked_lists) | Θ(n) | Θ(n) | Θ(1) | Θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) | |
| [Doubly-Linked List](http://en.wikipedia.org/wiki/Doubly_linked_list) | Θ(n) | Θ(n) | Θ(1) | Θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) | |
| [Skip List](http://en.wikipedia.org/wiki/Skip_list) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(n) | O(n) | O(n) | O(n) | O(n log(n)) | |
| [Hash Table](http://en.wikipedia.org/wiki/Hash_table) | N/A | Θ(1) | Θ(1) | Θ(1) | N/A | O(n) | O(n) | O(n) | O(n) | |
| [Binary Search Tree](http://en.wikipedia.org/wiki/Binary_search_tree) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(n) | O(n) | O(n) | O(n) | O(n) | |
| [Cartesian Tree](https://en.wikipedia.org/wiki/Cartesian_tree) | N/A | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | N/A | O(n) | O(n) | O(n) | O(n) | |
| [B-Tree](http://en.wikipedia.org/wiki/B_tree) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) | |
| [Red-Black Tree](http://en.wikipedia.org/wiki/Red-black_tree) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) | |
| [Splay Tree](https://en.wikipedia.org/wiki/Splay_tree) | N/A | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | N/A | O(log(n)) | O(log(n)) | O(log(n)) | O(n) | |
| [AVL Tree](http://en.wikipedia.org/wiki/AVL_tree) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) | |
| [KD Tree](http://en.wikipedia.org/wiki/K-d_tree) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(n) | O(n) | O(n) | O(n) | O(n) | |

**Array Sorting Algorithms**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm | Time Complexity | | | Space Complexity |
|  | Best | Average | Worst | Worst |
| [Quicksort](http://en.wikipedia.org/wiki/Quicksort) | Ω(n log(n)) | Θ(n log(n)) | O(n^2) | O(log(n)) |
| [Mergesort](http://en.wikipedia.org/wiki/Merge_sort) | Ω(n log(n)) | Θ(n log(n)) | O(n log(n)) | O(n) |
| [Timsort](http://en.wikipedia.org/wiki/Timsort) | Ω(n) | Θ(n log(n)) | O(n log(n)) | O(n) |
| [Heapsort](http://en.wikipedia.org/wiki/Heapsort) | Ω(n log(n)) | Θ(n log(n)) | O(n log(n)) | O(1) |
| [Bubble Sort](http://en.wikipedia.org/wiki/Bubble_sort) | Ω(n) | Θ(n^2) | O(n^2) | O(1) |
| [Insertion Sort](http://en.wikipedia.org/wiki/Insertion_sort) | Ω(n) | Θ(n^2) | O(n^2) | O(1) |
| [Selection Sort](http://en.wikipedia.org/wiki/Selection_sort) | Ω(n^2) | Θ(n^2) | O(n^2) | O(1) |
| [Tree Sort](https://en.wikipedia.org/wiki/Tree_sort) | Ω(n log(n)) | Θ(n log(n)) | O(n^2) | O(n) |
| [Shell Sort](http://en.wikipedia.org/wiki/Shellsort) | Ω(n log(n)) | Θ(n(log(n))^2) | O(n(log(n))^2) | O(1) |
| [Bucket Sort](http://en.wikipedia.org/wiki/Bucket_sort) | Ω(n+k) | Θ(n+k) | O(n^2) | O(n) |
| [Radix Sort](http://en.wikipedia.org/wiki/Radix_sort) | Ω(nk) | Θ(nk) | O(nk) | O(n+k) |
| [Counting Sort](https://en.wikipedia.org/wiki/Counting_sort) | Ω(n+k) | Θ(n+k) | O(n+k) | O(k) |
| [Cubesort](https://en.wikipedia.org/wiki/Cubesort) | Ω(n) | Θ(n log(n)) | O(n log(n)) | O(n) |

# Đồ thị

|  |  |
| --- | --- |
| Prim | Kruskal |
| //Lol just copied from hackerearth website  #include <iostream>  #include <vector>  #include <queue>  #include <functional>  #include <utility>  using namespace std;  const int MAX = 1e4 + 5;  typedef pair<long long, int> PII;  bool marked[MAX];  vector <PII> adj[MAX];  long long prim(int x)  {      priority\_queue<PII, vector<PII>, greater<PII> > Q;      int y;      long long minimumCost = 0;      PII p;      Q.push(make\_pair(0, x));      while(!Q.empty())      {          // Select the edge with minimum weight          p = Q.top();          Q.pop();          x = p.second;          // Checking for cycle          if(marked[x] == true)              continue;          minimumCost += p.first;          marked[x] = true;          for(int i = 0;i < adj[x].size();++i)          {              y = adj[x][i].second;              if(marked[y] == false)                  Q.push(adj[x][i]);          }      }      return minimumCost;  }  int main()  {      int nodes, edges, x, y;      long long weight, minimumCost;      cin >> nodes >> edges;      for(int i = 0;i < edges;++i)      {          cin >> x >> y >> weight;          adj[x].push\_back(make\_pair(weight, y));          adj[y].push\_back(make\_pair(weight, x));      }      // Selecting 1 as the starting node      minimumCost = prim(1);      cout << minimumCost << endl;      return 0;  } | vector <tuple<int,int,int> > edges ; // weight,node A,node B  sort (edges.begin(), edges.end ()) ;  int total\_weight = 0;  for (auto e : edges) {      int weight, a, b;      tie (weight,a,b) = e ;      if (root(a) == root(b)) // taking this edge will cause a cycle          continue;      total\_weight += weight ; // take this edge      connect (a, b) ; // connect them in the UFDS  } |
| BFS |
| vector<int> g[100005];  queue<pair<int, int> > q;  int dist[1000005];  fill(dist, dist+1000005, -1);  while (!q.empty()) {      int v = q.front().first;      int d = q.front().second;      q.pop();      if (dist[v] != -1) continue; //Visited      dist[v] = d;      for (int i = 0; i < g[v].size(); i++) {          q.push(make\_pair(g[v][i], d+1));      }  }  Time Comple­xity: O(|V| + |E|) Space Comple­xity: O(b^d) |
| DFS |
| bool vis[N];  vector<int> adjList[N];  void dfs(int node) {      if (vis[node]) return;      vis[node] = true;      for (int a = 0; a < (int)adjList[node].size(); ++a)          dfs(adjList[node][a]);  }  ///Iterative////////////////////////////////  bool vis[N];  vector<int> adjList[N];  stack<int> S;  while (!S.empty()) {      int node = S.top();      S.pop();      if (vis[node]) continue;      vis[node] = true;      for (int a = 0; a < (int)adjList[node].size(); ++a)          S.push(adjList[node][a]);  }  DFS uses O(d) space, where d is the depth of the graph |

Dijkstra:

vector<pair<int,int> > adjList[10000]; // node, weight

priority\_queue<pair<int,int>, vector<pair<int,int> >, greater<pair<int,int> > > pq; // distance, node

int dist[10000];

memset(dist, -1, sizeof(dist));

pq.push(make\_pair(0, source)); dist[0] = 0;

while(!pq.empty()){

    pair<int,int> c = pq.top();

    pq.pop();

    if(c.first != dist[c.second]) continue;

    for(auto i : adjList[c.second]){

        if(dist[i.first] == -1 || dist[i.first] > c.first + i.second){

            dist[i.first] = c.first + i.second;

            pq.push(make\_pair(dist[i.first], i.first));

        }

    }

}

Time Complexity of our implem­ent­ation: O(E log E)  
Space Comple­xity: O(V+E)

# Sắp xếp Topo

Bước 1: Tìm tất cả các đỉnh có bậc vào 0 rồi add vào queue.

Bước 2: Tiến hành xoá đỉnh đầu tiên trong queue và các cạnh có liên quan, lấy đỉnh đó ra khỏi queue.

Bước 3: Lặp lại từ bước 1, nếu hàng đợi rỗng thì kết thúc thuật toán.

# Bảng băm

3.1. Separate chaining (open hashing)

Separate chaining là một kỹ thuật xử lý và chạm phổ biến nhất. Nó thường được cài đặt với danh sách liên kết. Để lưu giữ một phần tử trong bảng băm, bạn phải thêm nó vào một danh sách liên kết ứng với chỉ mục của nó. Nếu có sự va chạm xảy ra, các phần tử đó sẽ nằm cùng trong 1 danh sách liên kết(xem ảnh để hiểu hơn).

Như vậy, kỹ thuật này sẽ đạt được tốc độ tìm kiếm O(1) trong trường hợp tối ưu và O(N) nếu tất cả các phần tử ở cùng 1 danh sách liên kết duy nhất. Đó là do có điều kiện 3 trong tiêu chí hàm băm tốt.

3.2. Linear probing (open addressing or closed hashing)

Trong kỹ thuật xử lý va chạm này, chúng ta sẽ không dùng linklist để lưu trữ mà chỉ có bản thân array đó thôi.

Khi thêm vào bảng băm, nếu chỉ mục đó đã có phần tử rồi; Giá trị chỉ mục sẽ được tính toán lại theo cơ chế tuần tự. Giả sử rằng chỉ mục là chỉ số của mảng, khi đó, việc tính toán chỉ mục cho phần tử được tính theo cách sau:

index = index % hashTableSize  
index = (index + 1) % hashTableSize  
index = (index + 2) % hashTableSize  
index = (index + 3) % hashTableSize

Và cứ thế theo cách như vậy chừng nào index thu được chưa có phần tử được sử dụng. Tất nhiên, không gian chỉ mục phải được đảm bảo để luôn có chỗ cho phần tử mới.

Như trong ví dụ dưới đây, chuỗi Hashing bị trùng chỉ mục(2) ở lần đầu tính chỉ mục. Do đó, nó được đẩy lên vị trí trống ở phía sau(3).

3.3. Quadratic Probing

Ý tưởng cũng khá giống Linear Probing, nhưng cách tính chỉ mục có khác đôi chút:

index = index % hashTableSize  
index = (index + 12) % hashTableSize  
index = (index + 22) % hashTableSize  
index = (index + 32) % hashTableSize

Và cứ thế cho tới khi tìm được chỉ mục trống.

**Cài đặt bảng băm dùng Quadratic Probing**

Giả sử rằng:

* Không có nhiều hơn 20 phần tử trong tập dữ liệu
* Hàm băm sẽ trả về một số nguyên từ 0 đến 19
* Tập dữ liệu phải là các phần tử duy nhất

3.4. Double hashing

Vẫn giống 2 kỹ thuật ngay phía trên, chỉ khác ở công thức tính khi xảy ra va chạm như sau:

index = (index + 1 \* indexH) % hashTableSize;  
index = (index + 2 \* indexH) % hashTableSize;

Và cứ tiếp tục cho tới khi tìm được chỉ mục chưa được sử dụng.

**Cài đặt bảng băm dùng Double hashing**

Giả sử rằng:

* Không có nhiều hơn 20 phần tử trong tập dữ liệu
* Hàm băm sẽ trả về một số nguyên từ 0 đến 19
* Tập dữ liệu phải là các phần tử duy nhất