THUẬT TOÁN SINH

Nội dung:

1. Giới thiệu
2. Mã giả
3. Sinh xâu nhị phân
4. Sinh tổ hợp chập K của N phần tử
5. Sinh hoán vị
6. Sinh phân hoạch
7. Sinh tập con bằng toán tử bit

1. Giới thiệu:

Thuật toán sinh là một phương pháp vét cạn được dùng với các bài toán liệt kê hoặc đếm cấu hình, thỏa các yêu cầu sau:

- Có thể xác định được cấu hình đầu tiên và cấu hình cuối cùng.
- Tìm được thuật toán để từ cấu hình hiện tại sinh ra cấu hình kế tiếp.

Các bài toán phổ biến sử dụng phương pháp sinh:

- Liệt kê xâu nhị phân
- Liệt kê hoán vị
- Liệt kê tập con
- Sinh phân hoạch

2. Mã giả:

Mã giả:

```
<Bước 1> : Khởi tạo
    <Khởi tạo cấu hình đầu tiên>
<Bước 2> : Lặp
while(<Chưa gặp cấu hình cuối cùng>){
   <Đưa ra cấu hình hiện tại>
   <Sinh ra cấu hình kế tiếp>
}
<Đưa ra cấu hình cuối cùng>
```

3. Sinh xâu nhị phân:

Phân tích bài toán

Đề bài: Liệt kê xâu nhị phân có độ dài N

OUTPUT (N=3)
000
001
010
011
100
101
110
111

Cấu hình đầu tiên: N bit 0

Cấu hình cuối cùng: N bit 1

3. Sinh xâu nhị phân:

Các biến toàn cục và hàm khởi tạo cấu hình đầu tiên

```
int n, X[100]; // lưu cấu hình
bool final = false; // check cấu hình cuối
void init(){
    for(int i = 1; i <= n; i++){
        X[i] = 0;
    }
}
```

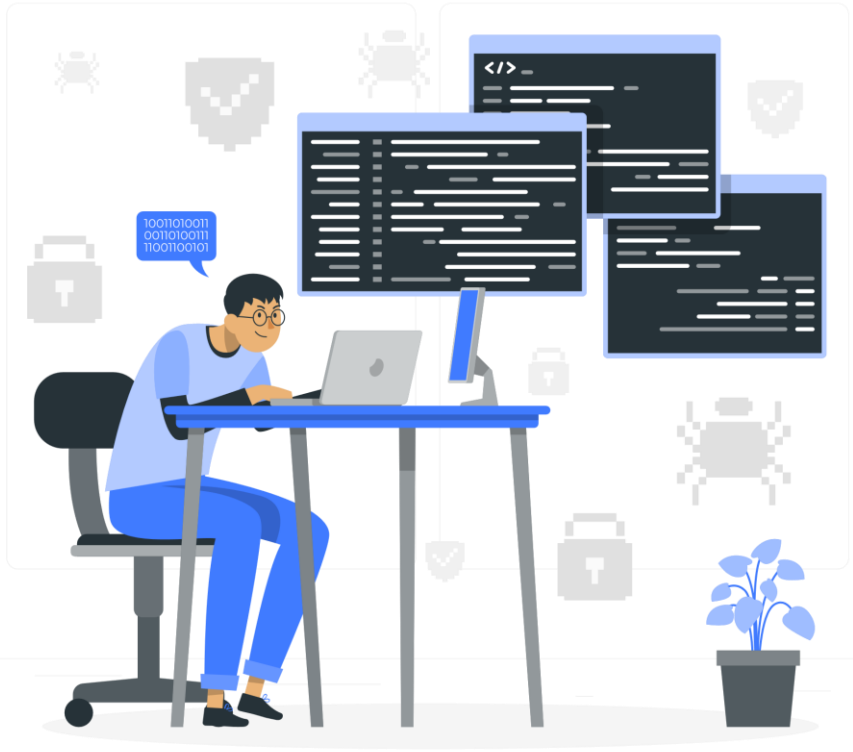
Hàm sinh cấu hình kế tiếp:

```
void sinh(){
    int i = n;
    while(i >= 1 && X[i] == 1){
        X[i] = 0;
        --i;
    }
    if(i == 0){
        final = true;
    }
    else{
        X[i] = 1;
    }
}
```

3. Sinh xâu nhị phân:

Hàm main:

```
int main(){
    n = 3;
    init();
    while(!final){
        for(int i = 1; i <= n; i++){
            cout << X[i];
        }
        cout << endl;
        sinh();
    }
}
```



4. Sinh tổ hợp chập K của N phần tử:

Phân tích bài toán

Đề bài: Cho các số tự nhiên từ 1 đến N, nhiệm vụ của bạn là liệt kê các tập con có K phần tử của tập N phần tử này theo thứ tự từ điển tăng dần.

Cấu hình đầu tiên: 123...K

OUTPUT (N=5, K=3)

123	145
124	234
125	235
134	245
135	345

Cấu hình cuối cùng: N-K+1, N-K+2,...N

4. Sinh tổ hợp chập K của N phần tử:

Các biến toàn cục và hàm khởi tạo cấu hình đầu tiên

```
int n, X[100]; // lưu cấu hình
bool final = false; // check cấu hình cuối
void init(){
    for(int i = 1; i <= k; i++){
        X[i] = i;
    }
}
```

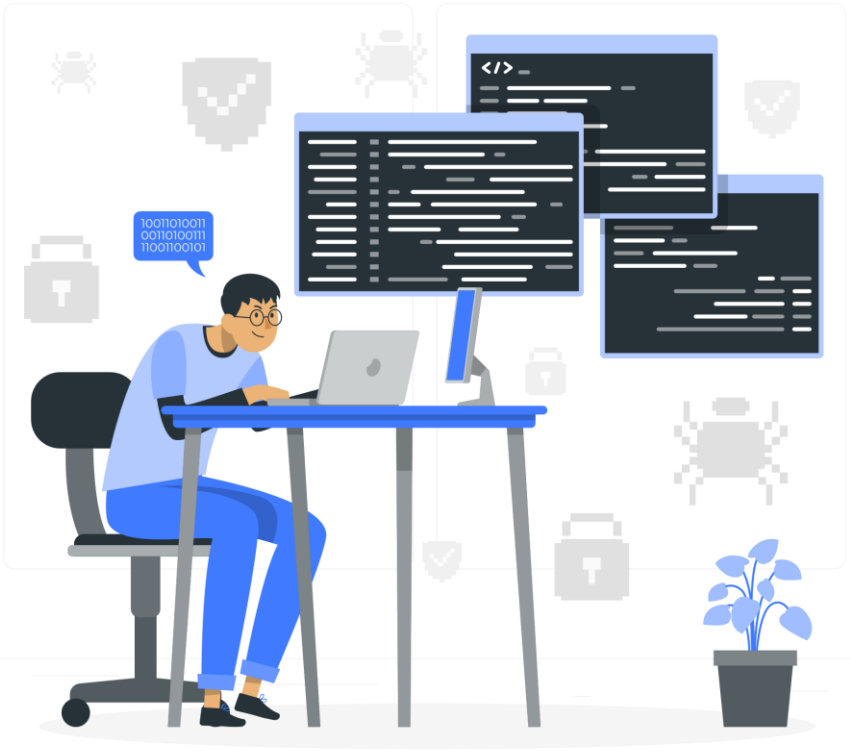
Hàm sinh cấu hình kế tiếp:

```
void sinh(){
    int i = k;
    while(i >= 1 && X[i] == n - k + i){
        --i;
    }
    if(i == 0){
        final = true;
    }
    else{
        X[i]++;
        for(int j = i + 1; j <= k; j++){
            X[j] = X[j - 1] + 1;
        }
    }
}
```

4. Sinh tổ hợp chập K của N phần tử:

Hàm main:

```
int main(){
    n = 5, k = 3;
    init();
    while(!final){
        for(int i = 1; i <= k; i++){
            cout << X[i];
        }
        cout << endl;
        sinh();
    }
}
```



5. Sinh hoán vị:

Phân tích bài toán

Đề bài: Sinh ra các hoán vị của các số tự nhiên từ 1 đến N

OUTPUT (N=3)
123
132
213
231
312
321

Cấu hình đầu tiên: 123...N

Cấu hình cuối cùng: N,N-1,N-2...1

5. Sinh hoán vị:

Các biến toàn cục và hàm khởi tạo cấu hình đầu tiên

```
int n, X[100]; // lưu cấu hình
bool final = false; // check cấu hình cuối
void init(){
    for(int i = 1; i <= n; i++){
        X[i] = i;
    }
}
```

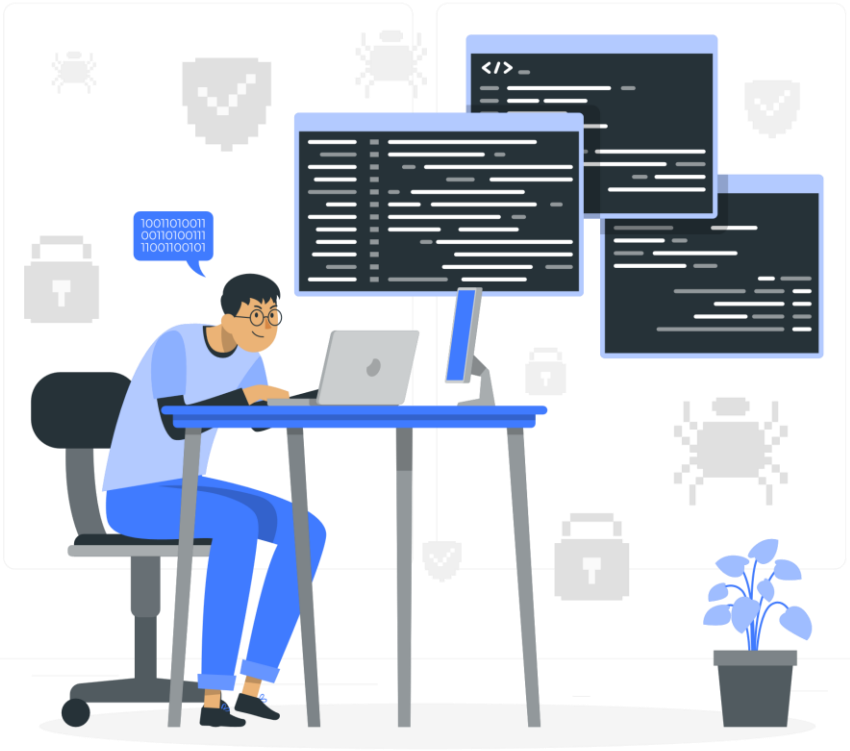
Hàm sinh cấu hình kế tiếp:

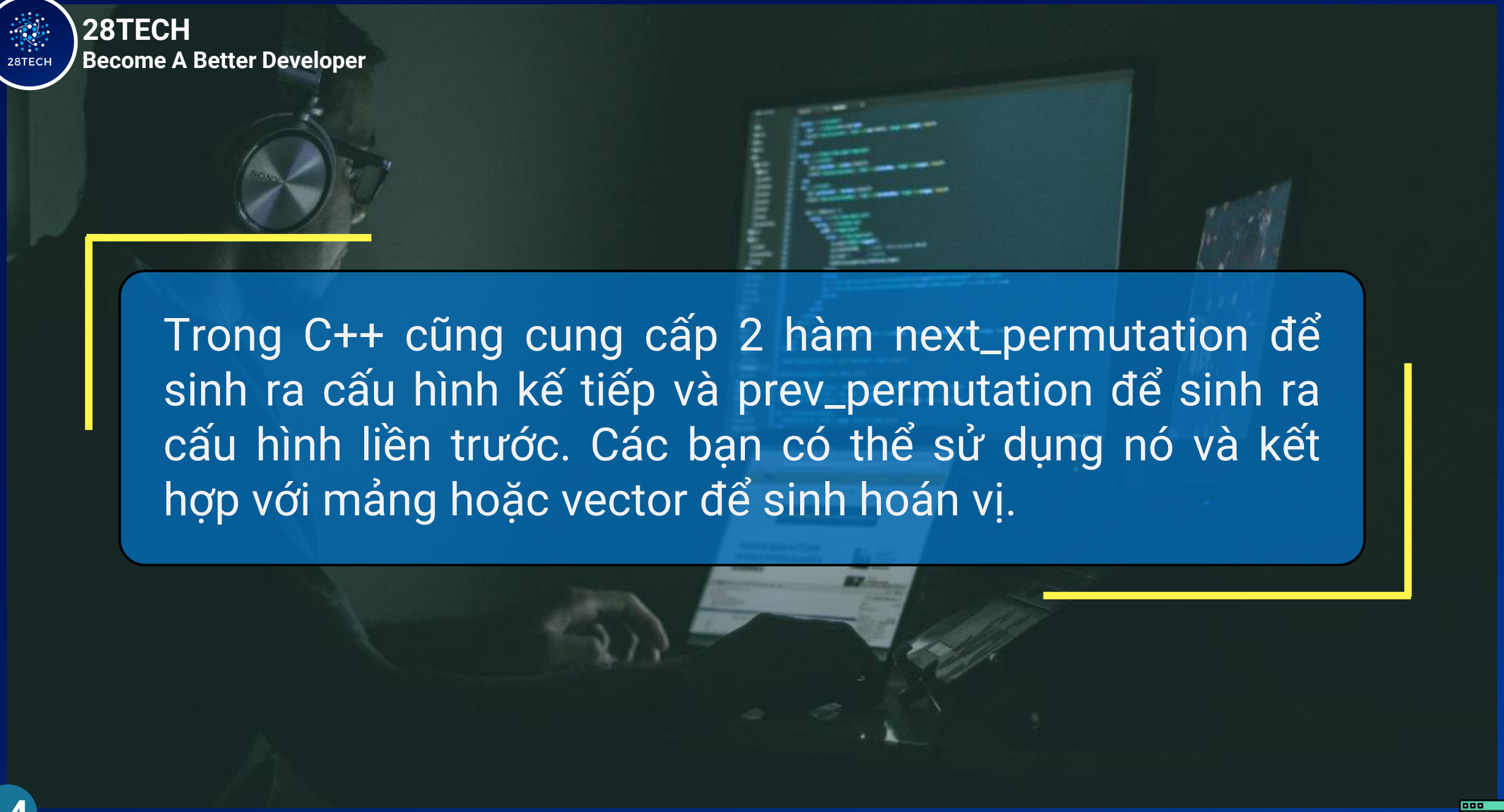
```
void sinh(){
    int i = n;
    while(i >= 1 && X[i] > X[i + 1]){
        --i;
    }
    if(i == 0){
        final = true;
    }
    else{
        int j = n;
        while(X[i] > X[j]) --j;
        swap(X[i], X[j]);
        reverse(X + i + 1, X + n + 1);
    }
}
```

5. Sinh hoán vị:

Hàm main:

```
int main(){
    n = 4;
    init();
    while(!final){
        for(int i = 1; i <= n; i++){
            cout << X[i];
        }
        cout << endl;
        sinh();
    }
}
```





Trong C++ cũng cung cấp 2 hàm next_permutation để sinh ra cấu hình kế tiếp và prev_permutation để sinh ra cấu hình liền trước. Các bạn có thể sử dụng nó và kết hợp với mảng hoặc vector để sinh hoán vị.

5. Sinh hoán vị:

Hàm next_permutation áp dụng với mảng, đối với vector và string các bạn thay bằng iterator begin và end.

Ví dụ:

```
int main(){
    int a[] = {1, 2, 3, 4};
    do{
        for(int i = 0; i < 4; i++){
            cout << a[i];
        }
        cout << endl;
    }while(next_permutation(a, a + 4));
}
```

5. Sinh hoán vị:

Hàm prev_permutation để sinh hoán vị theo thứ tự ngược

Ví dụ:

```
int main(){
    int a[] = {4, 3, 2, 1};
    do{
        for(int i = 0; i < 4; i++){
            cout << a[i];
        }
        cout << endl;
    }while(prev_permutation(a, a + 4));
}
```

6. Sinh phân hoạch:

Phân tích bài toán

Đề bài: In ra các cách phân tích N dưới dạng tổng các số tự nhiên nhỏ hơn hoặc bằng N không xét đến thứ tự.

Cấu hình đầu tiên: N

Cấu hình cuối cùng: N số 1

OUTPUT (N=5)
5
4 + 1
3 + 2
2 + 2 + 1
2 + 1 + 1 + 1
1 + 1 + 1 + 1 + 1

6. Sinh phân hoạch:

Các biến toàn cục và hàm khởi tạo cấu hình đầu tiên

```
int n, X[100]; // lưu cấu hình
int cnt; //Lưu sl số hạng trong phân tích
bool final = false; // check cấu hình cuối
void init(){
    cnt = 1;
    X[1] = n;
}
```

Lưu cấu hình của số ban đầu

Hàm sinh cấu hình kế tiếp:

```
void sinh(){
    int i = cnt;
    while(i >= 1 && X[i] == 1){
        --i;
    }
    if(i == 0){
        final = true;
    }
    else{
        int tmp = cnt - i + 1;
        --X[i];
        cnt = i;
        int q = tmp / X[i];
        int r = tmp % X[i];
        if(q != 0){
            for(int j = 1; j <= q; j++){
                X[i + j] = X[i];
            }
            cnt += q;
        }
        if(r != 0){
            ++cnt;
            X[cnt] = r;
        }
    }
}
```

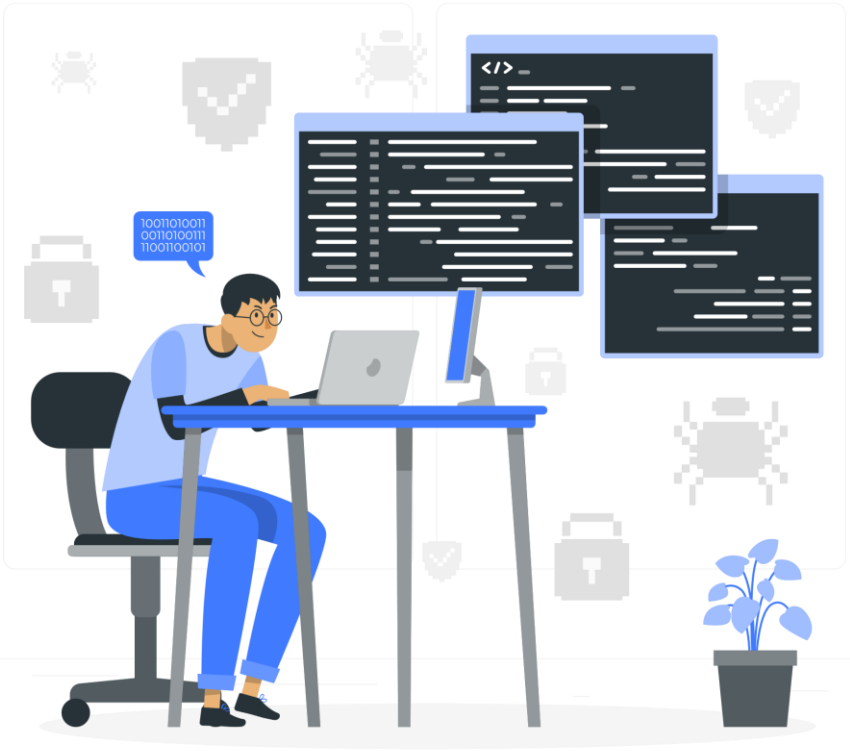
lấy ra giá trị còn thiếu

Bù 1 sau khi trừ

6. Sinh phân hoạch:

Hàm main:

```
int main(){
    n = 5;
    init();
    while(!final){
        for(int i = 1; i <= cnt; i++){
            cout << X[i] << ' ';
        }
        cout << endl;
        sinh();
    }
}
```



7. Sinh tập con bằng toán tử bit

Để sinh tập con bạn có thể sử dụng sinh nhị phân, mỗi cấu hình nhị phân tương ứng với 1 tập con của tập N phần tử. Ví dụ tập {1, 2, 3}

Cấu hình	Tập con
000	Tập rỗng
001	{3}
010	{2}
011	{2, 3}
100	{1}
101	{1,3}
110	{1,2}
111	{1,2,3}

7. Sinh tập con bằng toán tử bit

Sử dụng toán tử bit:

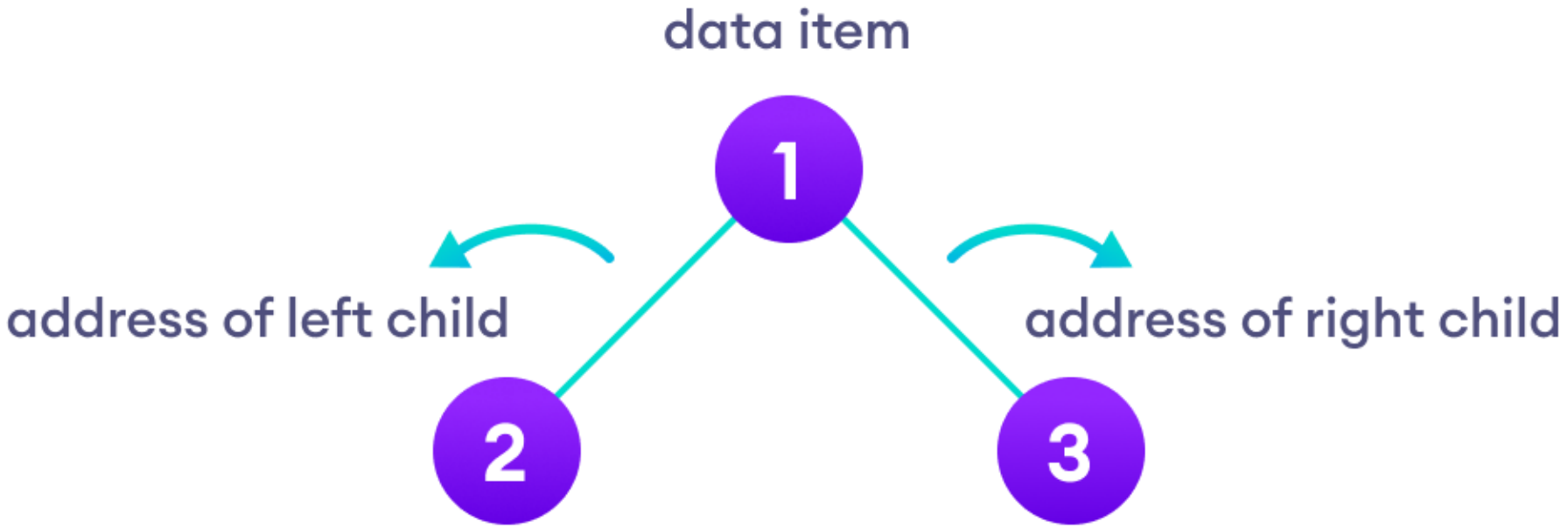
```
#include <bits/stdc++.h>
using namespace std;
int main(){
    int a[] = {1, 2, 3};
    int n = 3;
    for(int i = 0; i < (1 << 3); i++){
        for(int j = 0; j < 3; j++){
            if(i & (1 << j)){
                cout << a[j] << ' ';
            }
        }
        cout << endl;
    }
}
```

GIỚI THIỆU CÂY NHỊ PHÂN

1. Giới thiệu cây nhị phân:

Cây nhị phân là một cấu trúc dữ liệu mà mỗi node cha có tối đa 2 node con thường gọi là node con bên trái và node con bên phải.

Mỗi node trên cây nhị phân gồm 3 phần: Phần dữ liệu, địa chỉ của node con bên trái, địa chỉ của node con bên phải.



2. Các khái niệm cơ bản:

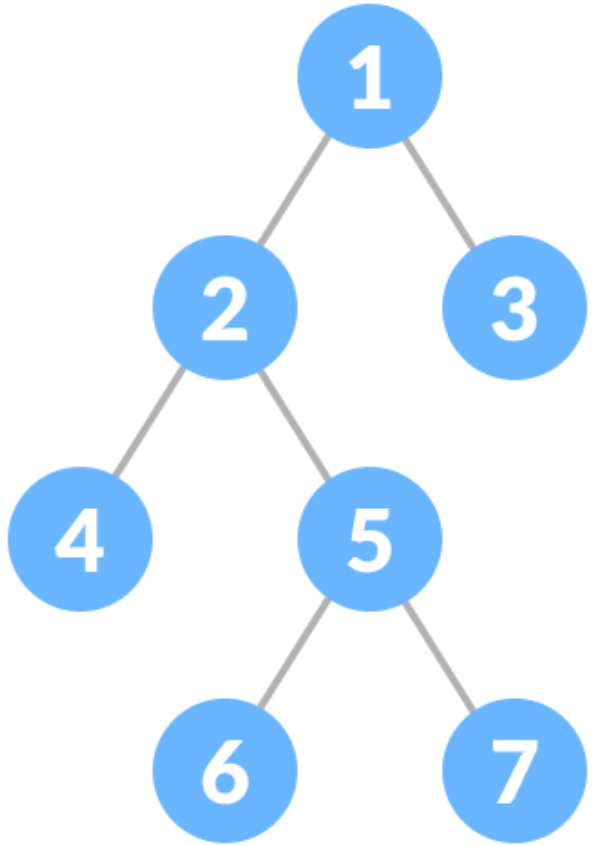
- **Node gốc (root):** Node đầu tiên tạo nên cây nhị phân.
- **Node cha (parent):** Node A là node cha của node B nếu B là node con bên trái hoặc bên phải của node A.
- **Node lá (leaf):** Node không có node con.
- **Node trung gian (Internal):** Node có node con bên trái hoặc node con bên phải hoặc cả 2.
- **Node trước (Ancestor):** Node A gọi là node trước của node B nếu B thuộc cây con bên trái hoặc cây con bên phải với gốc A.
- **Node sau trái (left descendent):** Node B gọi là node sau trái của node A nếu B thuộc cây con bên trái với gốc A.
- **Node sau phải (right descendent):** Node B gọi là node sau phải củ node A nếu B thuộc cây con bên phải với gốc A.

2. Các khái niệm cơ bản:

- **Bậc của node (degree):** Số cây con tối đa của một node, đối với cây nhị phân thì bậc của một node tối đa là 2
- **Mức của node (level):** Node gốc có mức 0, các node khác trên cây có mức bằng mức của node cha cộng thêm 1
- **Chiều sâu của node:** Bằng số cạnh trên đường đi từ node gốc tới node đang xét
- **Chiều sâu của cây:** Bằng mức cao nhất của 1 node lá trên cây hay chiều sâu lớn nhất của 1 node
- **Chiều cao của node:** Bằng số cạnh lớn nhất từ 1 node lá tới node đang xét
- **Chiều cao của cây:** Bằng chiều cao của node gốc

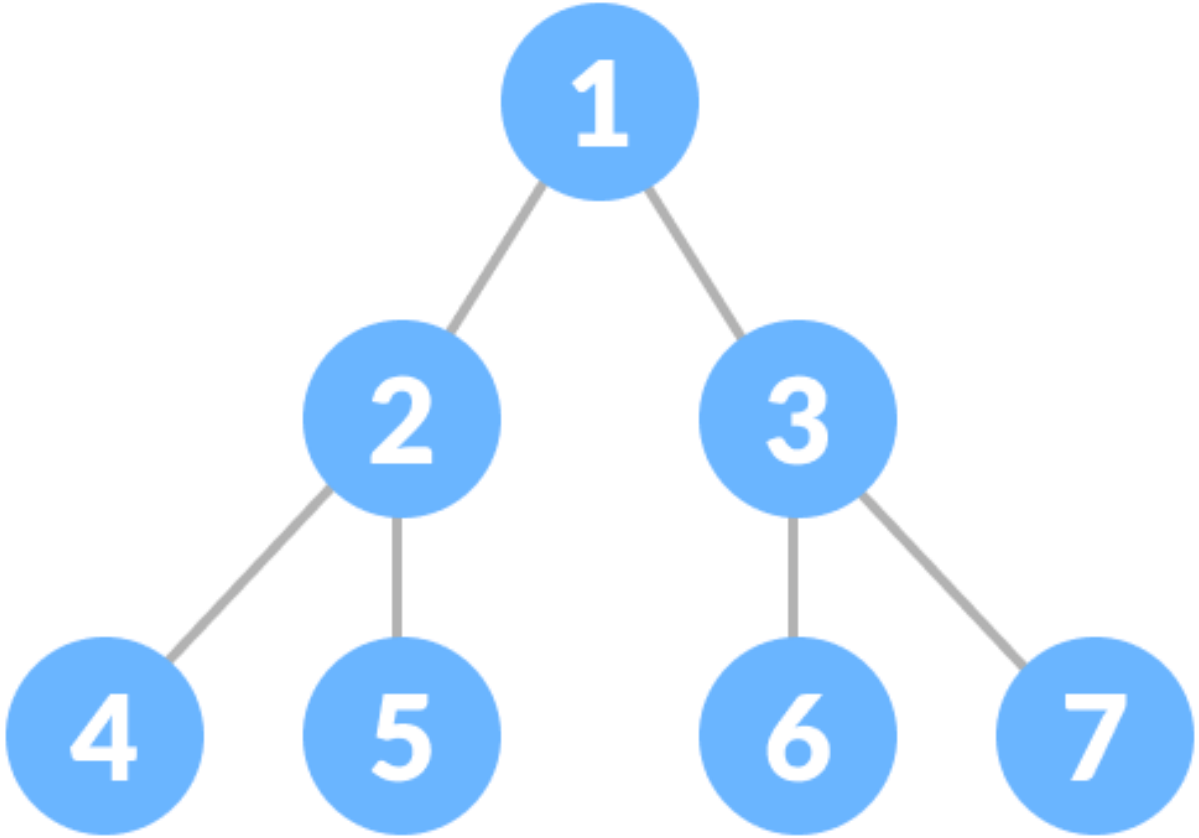
3. Các kiểu cây nhị phân:

Cây nhị phân đầy đủ (full binary tree): Mỗi node cha có 0 hoặc 2 node con



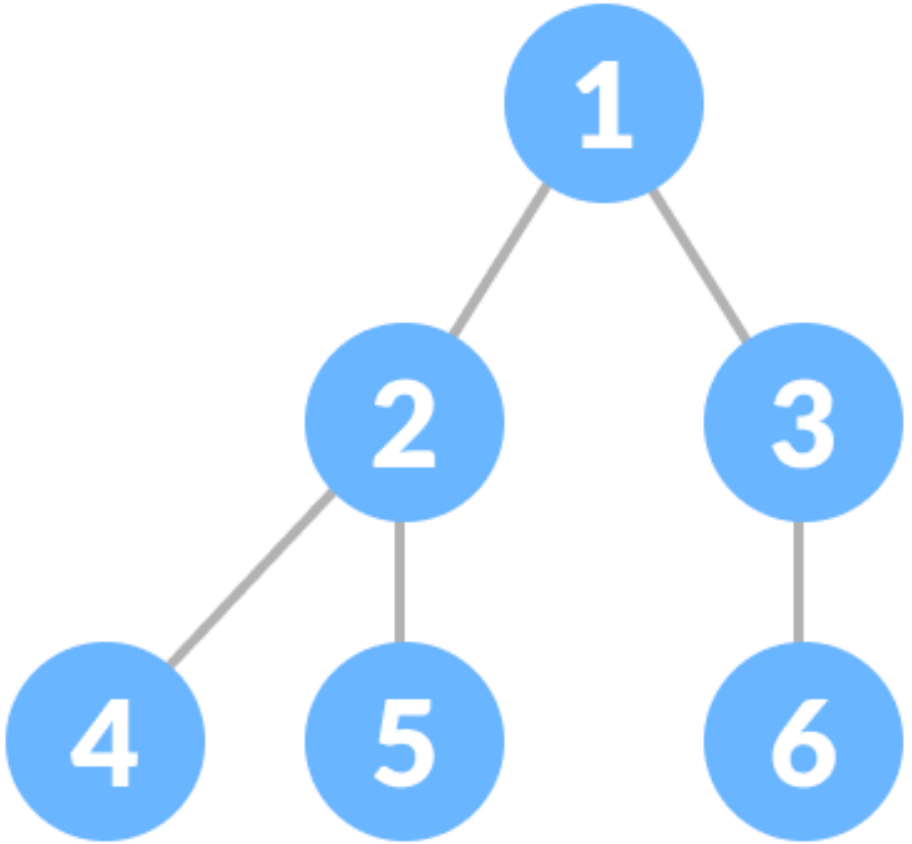
3. Các kiểu cây nhị phân:

Cây nhị phân hoàn hảo (Perfect binary tree): Mỗi node trung gian đều có 2 con và mọi node lá đều có cùng mức.



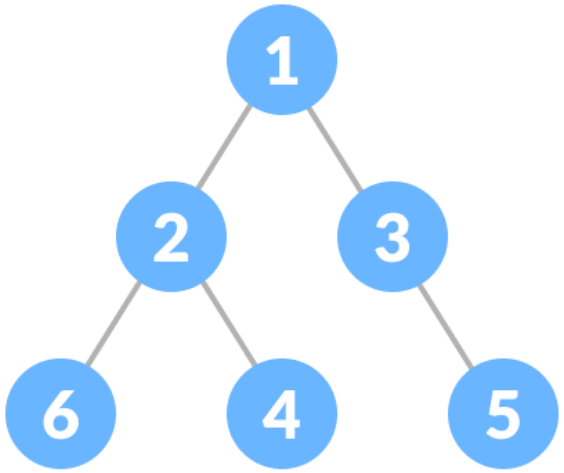
3. Các kiểu cây nhị phân:

Cây nhị phân hoàn chỉnh (complete binary tree): Tất cả các node lá phải tinh gọn về bên trái; Node lá cuối cùng không nhất thiết phải có node anh em bên phải.

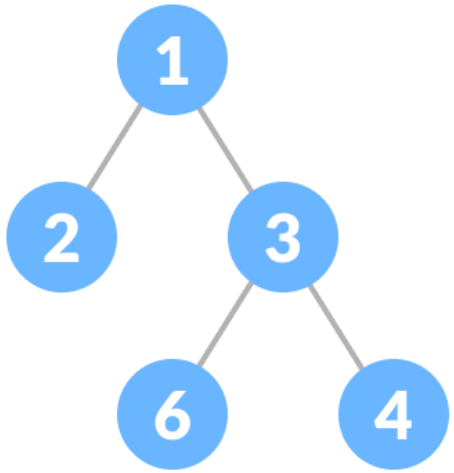


3. Các kiểu cây nhị phân:

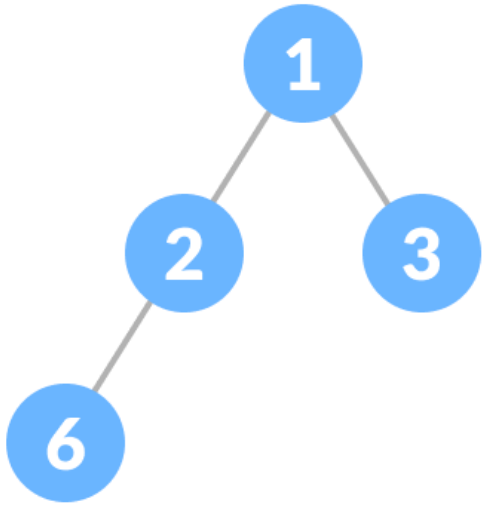
Phân biệt Full và Complete binary tree:



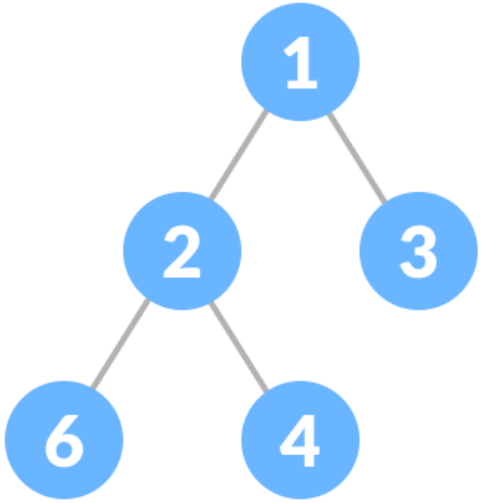
✗ Full Binary Tree
✗ Complete Binary Tree



✓ Full Binary Tree
✗ Complete Binary Tree



✗ Full Binary Tree
✓ Complete Binary Tree

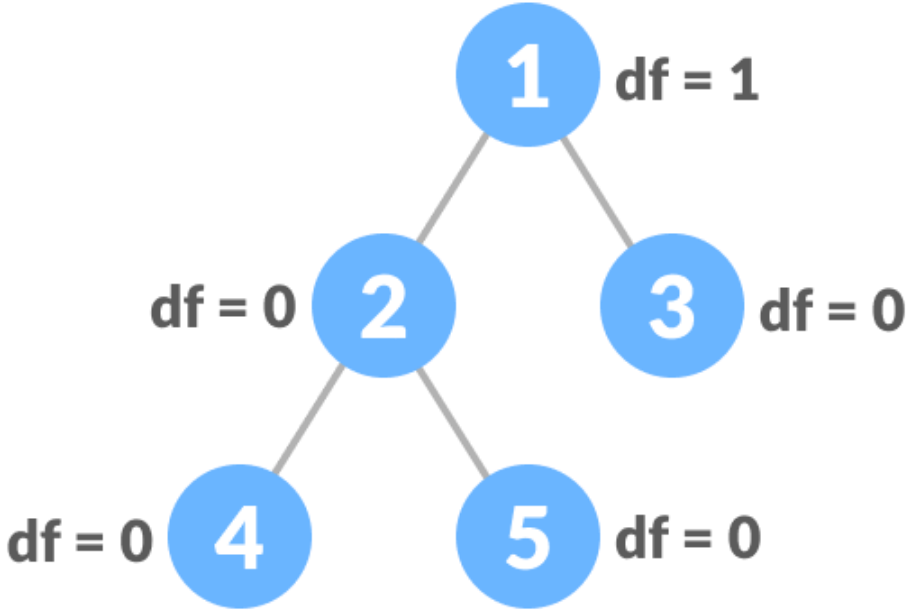


✓ Full Binary Tree
✓ Complete Binary Tree

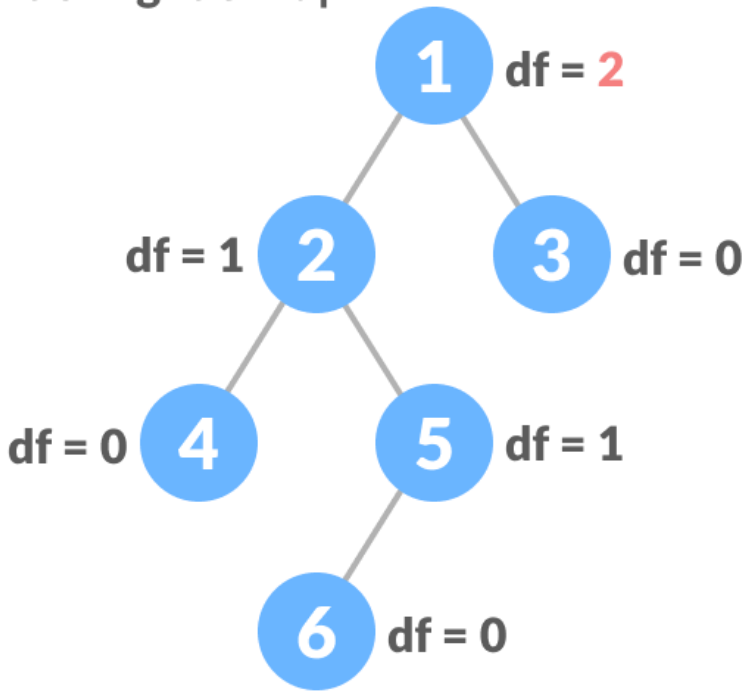
3. Các kiểu cây nhị phân:

Cây nhị phân cân bằng (Balanced binary tree): Mọi node trên cây có độ cao của cây con bên trái và độ cao của cây con bên phải chênh lệch nhau không quá 1.

df = |height of left child - height of right child|



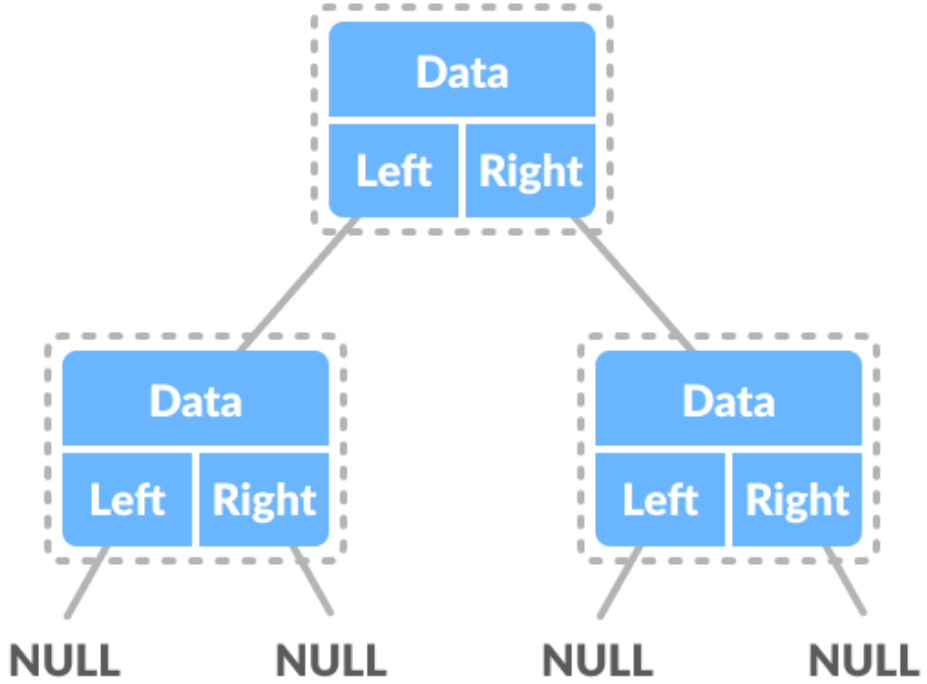
Cây nhị phân cân bằng



Cây nhị phân không cân bằng

4. Biểu diễn cây nhị phân:

Để quản lý cây nhị phân ta chỉ cần lưu địa chỉ của node gốc, tương tự như DSLK thì các node trên cây nhị phân đều được cấp phát động. Ngoài ra ta có thể biểu diễn cây nhị phân dưới dạng liên tục bằng cách sử dụng mảng 1 chiều, trong đó node gốc ở chỉ số 0, node cha ở chỉ số p thì node con bên trái ở chỉ số 2p + 1, node con bên phải ở chỉ số 2p + 2.



4. Biểu diễn cây nhị phân:

Định nghĩa cấu trúc node

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

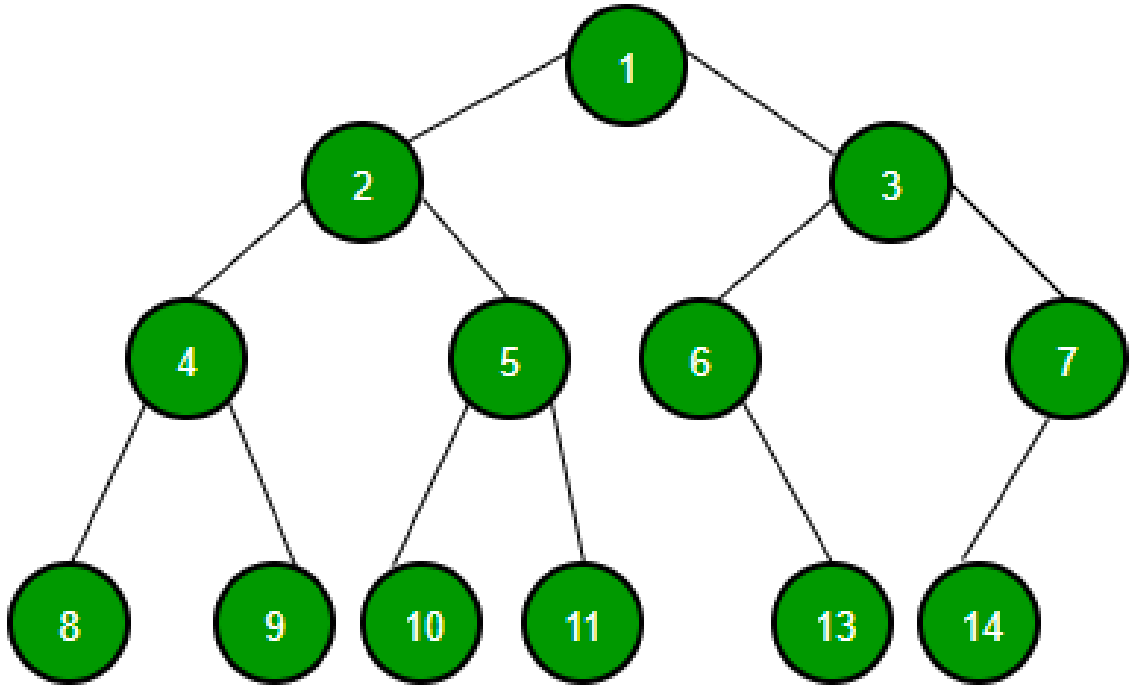
Cấp phát node mới

```
node *makeNode(int x){
    node *newNode = new node;
    newNode->data = x;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}
```

5. Các thứ tự duyệt cây phổ biến:

a. Duyệt giữa - Inorder (left - root - right):

Thứ tự duyệt:
8 - 4 - 9 - 2 - 10 - 5 - 11 - 1 - 6 - 13 - 3 - 14 - 7



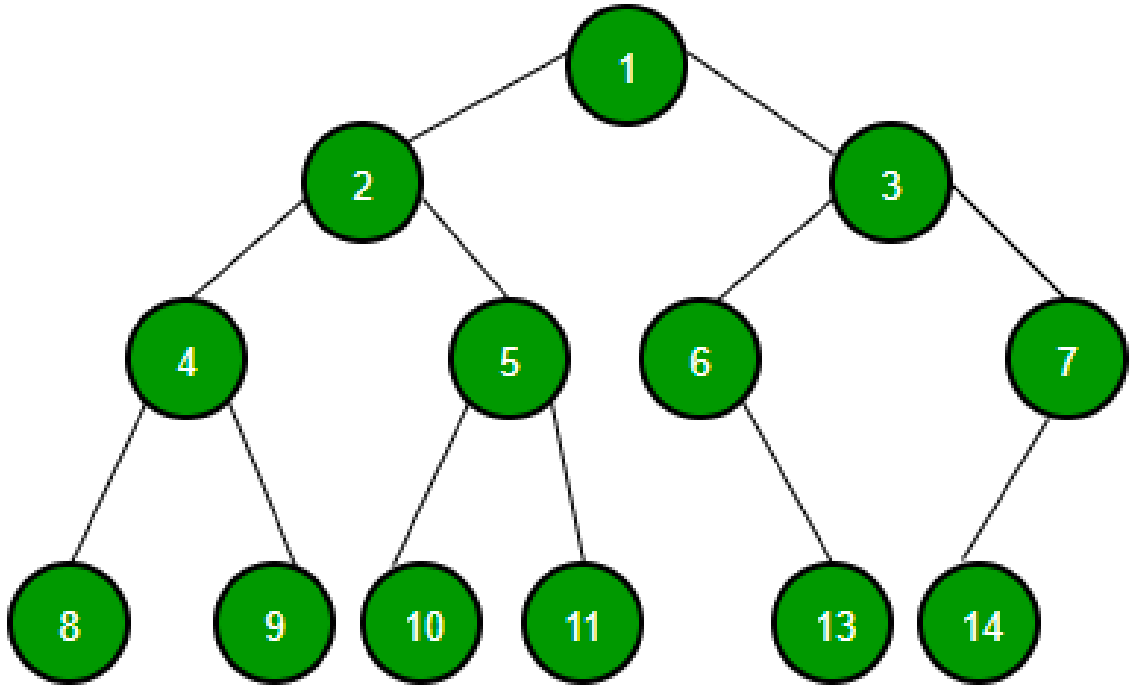
Code:

```
void inorder(node *root){
    if(root != NULL){
    inorder(root->left);
    cout << root->data << ' ';
    inorder(root->right);
    }
}
```

5. Các thứ tự duyệt cây phổ biến:

b. Duyệt trước - Preorder (root - left - right):

Thứ tự duyệt:
1 - 2 - 4 - 8 - 9 - 5 - 10 - 11 - 3 - 6 - 13 - 7 - 14



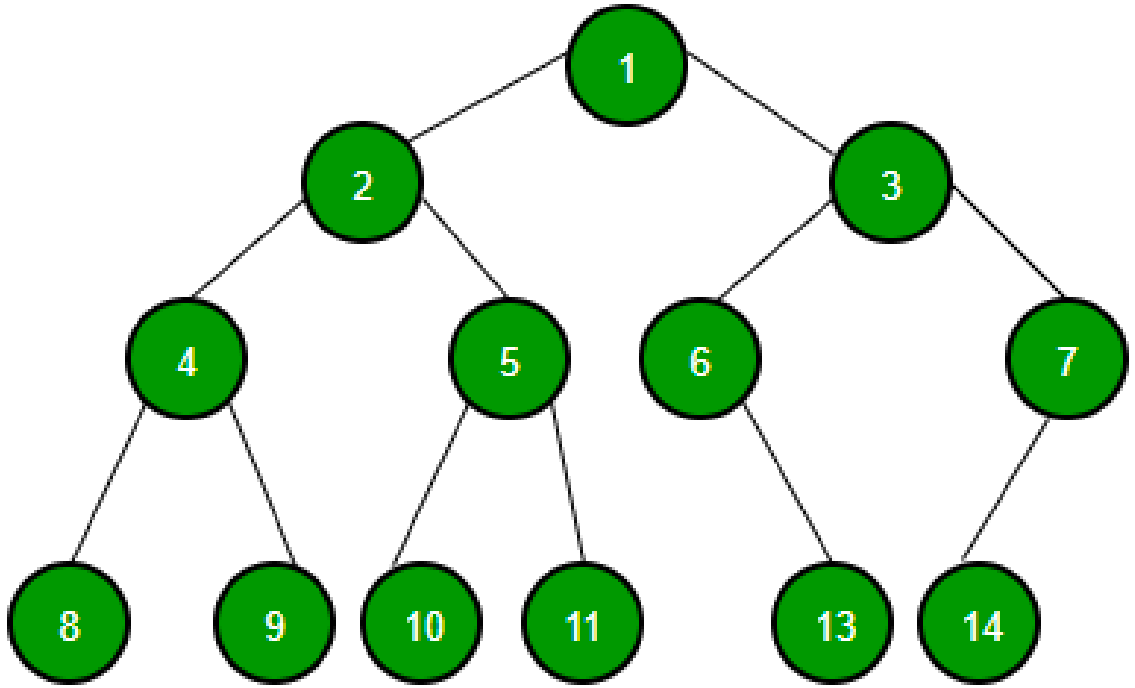
Code:

```
void preorder(node *root){
    if(root != NULL){
    cout << root->data << ' ';
    preorder(root->left);
    preorder(root->right);
    }
}
```

5. Các thứ tự duyệt cây phổ biến:

c. Duyệt sau - Postorder (left - right - root):

Thứ tự duyệt:
8 - 9 - 4 - 10 - 11 - 5 - 2 - 13 - 6 - 14 - 7 - 3 - 1



Code:

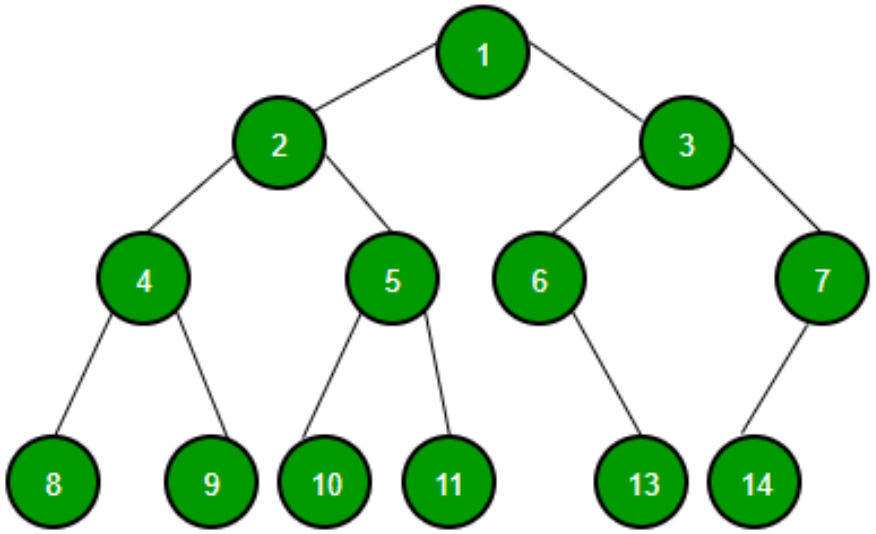
```
void postorder(node *root){
    if(root != NULL){
    postorder(root->left);
    postorder(root->right);
    cout << root->data << ' ';
    }
}
```

5. Các thứ tự duyệt cây phổ biến:

d. Duyệt theo mức - Level Order:

Duyệt theo mức trên cây nhị phân chính là áp dụng BFS trên cây nhị phân.

Thứ tự duyệt:
1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - 11 - 13 - 14



Code:

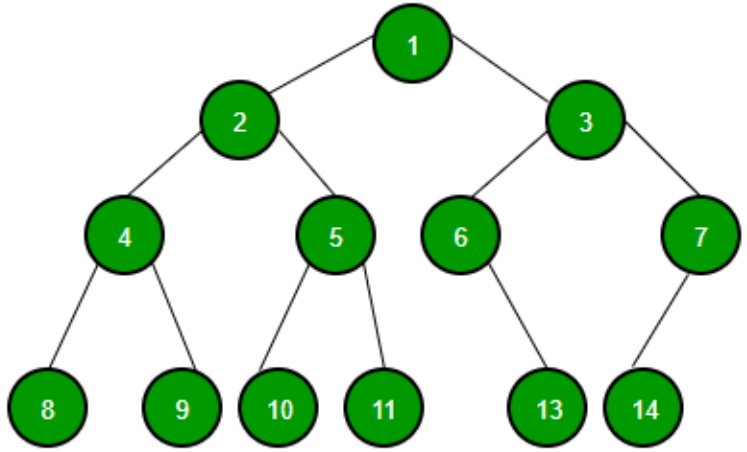
```
void levelOrder(node *root){
    queue<node*> q;
    q.push(root);
    cout << root->data << ' ';
    while(!q.empty()){
        node *tmp = q.front(); q.pop();
        if(tmp->left != NULL){
            q.push(tmp->left);
            cout << tmp->left->data << ' ';
        }
        if(tmp->right != NULL){
            q.push(tmp->right);
            cout << tmp->right->data << ' ';
        }
    }
}
```

5. Các thứ tự duyệt cây phổ biến:

e. Duyệt xoắn ốc - Spiral Order:

Để duyệt xoắn ốc ta sử dụng 2 ngăn xếp, ngăn xếp s1 để in các node theo thứ tự từ trái sang phải, ngăn xếp s2 để in các node theo thứ tự từ phải sang trái.

Thứ tự duyệt:
1 - 2 - 3 - 7 - 6 - 5 - 4 - 8 - 9 - 10 - 11 - 13 - 14



Code:

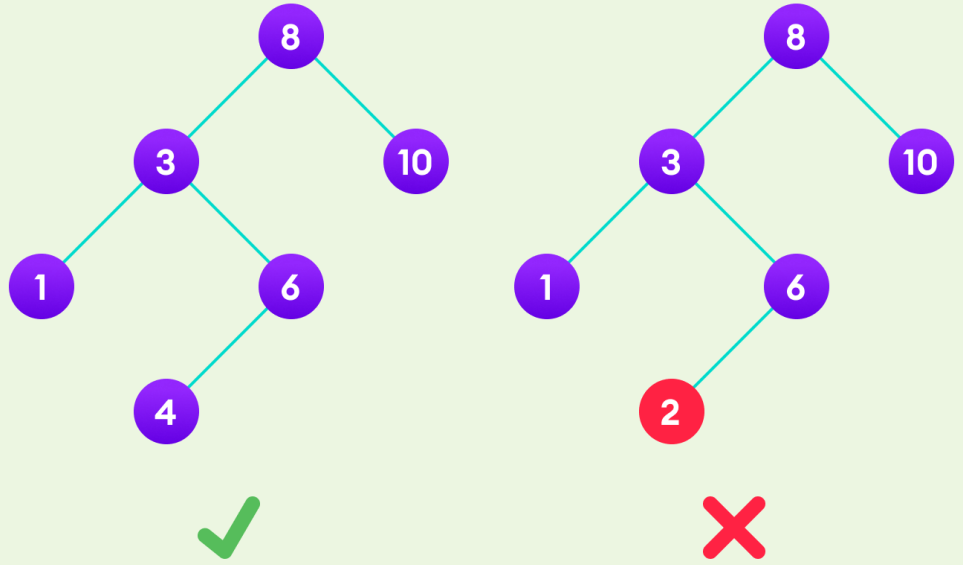
```
void spiral(node *root){
   stack<node*> s1, s2;
   s1.push(root);
   while(!s1.empty() || !s2.empty()){
      while(!s1.empty()){
         root *top = s1.top(); s1.pop();
         cout << top->data << ' ';
         if(top->right != NULL){
            s2.push(top->right);
         if(top->left != NULL)
            s2.push(top->left);
      }
      while(!s2.empty()){
         root *top = s2.top(); s2.pop();
         cout << top->data << ' ';
         if(top->left != NULL){
            s1.push(top->left);
         if(top->right != NULL){
            s1.push(top->right);
      }
   }
}
```

6. Cây nhị phân tìm kiếm:

Cây nhị phân tìm kiếm - Binary search tree (BST) là cây nhị phân có thể tìm kiếm sự xuất hiện của 1 phần tử trên cây với độ phức tạp logN.

Các tính chất của BST:

- Tất cả node con thuộc cây con bên trái nhỏ hơn node gốc.
- Tất cả node con thuộc cây con bên phải lớn hơn node gốc.
- Cây con bên trái và bên phải cũng là cây BST.



6. Cây nhị phân tìm kiếm:

a. Thao tác tìm kiếm:

Do tính chất của BST nên khi tìm kiếm ta có thể nhanh chóng xác định được giá trị cần tìm kiếm thuộc cây con bên trái hay cây con bên phải của node gốc.

Có 3 trường hợp:

Trường hợp 1: Giá trị cần tìm kiếm bằng giá trị của node đang xét, ta kết luận tìm thấy.

Trường hợp 2: Nếu giá trị cần tìm kiếm nhỏ hơn giá trị node đang xét, tìm kiếm ở cây con bên trái.

Trường hợp 3: Tìm kiếm ở cây con bên phải nếu giá trị tìm kiếm lớn hơn giá trị node gốc.

Code:

```
bool search(node *root, int key){
    if(root == NULL) return false;
    if(root->data == key) return true;
    else if(root->data < key)
        return search(root->right, key);
    else
        return search(root->left, key);
}
```

6. Cây nhị phân tìm kiếm:

b. Thao tác chèn:

Khi chèn một giá trị mới vào cây, cây vẫn phải duy trì được tính chất là cây nhị phân tìm kiếm.

Thuật toán:

- Kiểm tra giá trị cần chèn với giá trị của node gốc, nếu giá trị cần chèn nhỏ hơn giá trị node gốc thì đi sang cây con bên trái, ngược lại đi sang cây con bên phải.
- Khi gặp node lá, chèn giá trị cần chèn vào node con bên trái hoặc bên phải tùy theo giá trị.

Code:

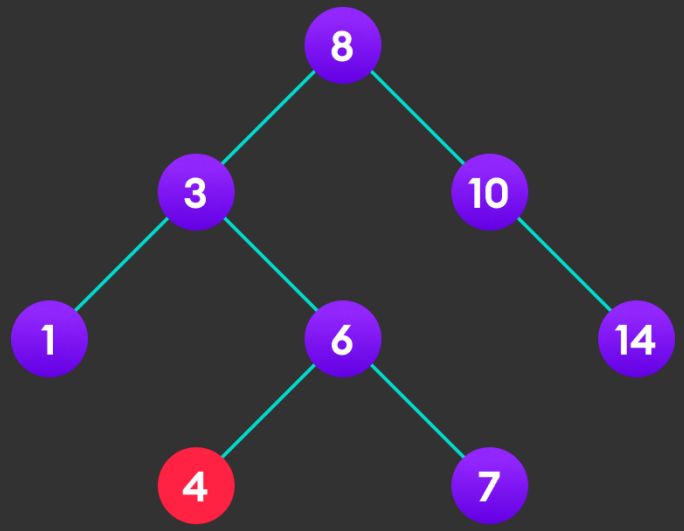
```
node *insert(node *root, int key){
    if(root == NULL){
        return makeNode(key);
    }
    if(key < root->data){
        root->left = insert(root->left, key);
    }
    else if(key > root->data){
        root->right = insert(root->right, key);
    }
    return root;
}
```

6. Cây nhị phân tìm kiếm:

c. Thao tác xóa:

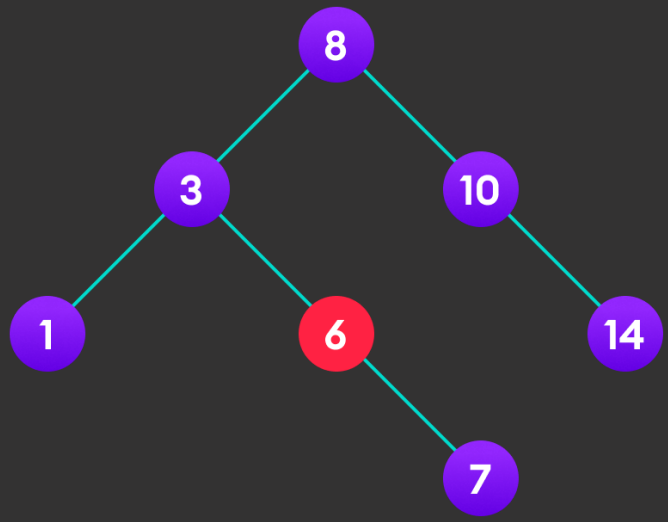
Trường hợp 1:

Node cần xóa là node lá, ta chỉ cần giải phóng node này:



Trường hợp 2:

Node cần xóa có 1 node con, khi đó gán giá trị của node cần xóa bằng giá trị của node con sau đó giải phóng node con:



6. Cây nhị phân tìm kiếm:

c. Thao tác xóa:

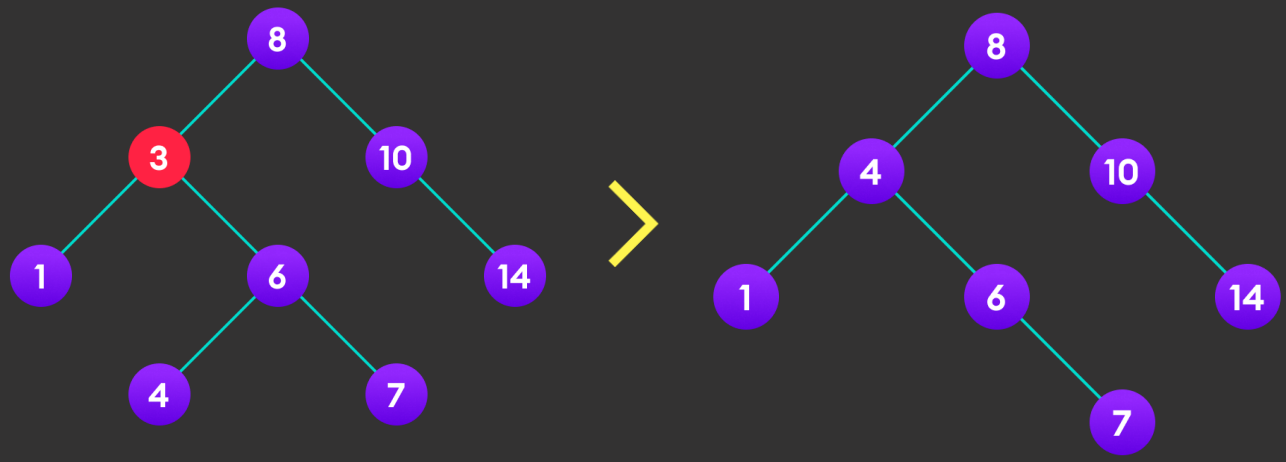
Trường hợp 3:

Node cần xóa có 2 node con

Bước 1. Tìm node con X nhỏ nhất lớn hơn node cần xóa

Bước 2. Gán giá trị node cần xóa thành giá trị của node X

Bước 3. Xóa node X khỏi cây



6. Cây nhị phân tìm kiếm:

c. Thao tác xóa:

Code:

```
node* minNode(node *root){
   node *temp = root;
   while(temp != NULL && temp->left != NULL){
      temp = temp->left;
   }
   return temp;
}

node* deleteNode(node *root, int key){
   if(root == NULL) return root;
   if(key < root->data){
   root->left = deleteNode(root->left, key);
   }
   else if(key > root->data){
      root->right = deleteNode(root->right, key);
   }
      else{
         if(root->left == NULL){
            node *tmp = root->right;
            delete root;
            return tmp;
         }
         else if(root->right == NULL){
            node *tmp = root->left;
            delete root;
            return tmp;
         }
         else{
            node *tmp = minNode(root);
            root->data = tmp->data;
            root->right = deleteNode(root->right, tmp->data);
         }
      }
   }
```