ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

ĐỒ ÁN CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

ĐỀ TÀI: TRIE – PREFIX TREE

Người thực hiện : Trần Vạn Tấn.

MSSV : 23521407

Lớp : IT003.O21.CTTN

TP. Hồ Chí Minh

Mục lục

1.Giới thiệu về đồ án

Bạn có bao giờ tự hỏi làm thế nào mà khi ta nhập dữ liệu vào một trình soạn thảo hoặc một trình duyệt web thì sẽ xuất hiện các từ hoặc cụm từ được đề xuất tiếp theo dựa trên những gì bạn nhập không?

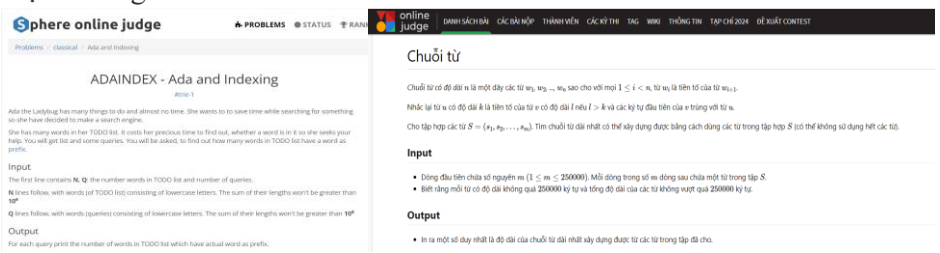
Trie sẽ giúp ta thực hiện những điều trên, và bên cạnh đó còn rất nhiều những ứng dụng khác trong thực tiễn!

Vậy chúng ta hãy cùng bàn bạc thêm về Trie thông qua đồ án này.

2.Quy trình thực hiện

2.1 Tuần 1:

- Tìm hiểu lý thuyết (Wikipedia) và cách cài đặt của cây Trie.
- Tìm bài trên các trang web lập trình thi đấu như Codeforces, LQDOJ, VNOJ, SPOJ,… và thực hành giải các bài Trie trên đó để hiểu hơn về Trie.



2.2 Tuần 2:

- Phác thảo sơ bộ khung và xây dựng mô hình đồ án.
- Tìm hiểu trên các nguồn khác nhau như VNOJ, VIBLO, geeksforgeeks:
  + Nhận xét được rằng VNOJ viết về cây Trie khá đầy đủ và dễ hiểu
  + Ở geeksforgeeks sẽ có mô tả về cách hoạt động của các thao tác trên cây trie khá bài bản và dễ hiểu
- Thực hiện đồ án trên word. Tìm hiểu giao diện word và tối ưu hóa.

3.Lịch sử ra đời, cách phát âm

Ý tưởng về trie để biểu diễn một tập hợp các chuỗi lần đầu tiên được mô tả một cách trừu tượng bởi Axel Thue vào năm 1912. Các phép thử lần đầu tiên được René de la Briandais mô tả trên máy tính vào năm 1959.

Ý tưởng này được mô tả độc lập vào năm 1960 bởi Edward Fredkin, ông đã công bố cấu trúc dữ liệu Trie vào cùng năm. Ý tưởng này chính là tiền đề cho các ứng dụng và phiên bản của cây trie hiện đại sau này.

Edward Fredkin, người đã đặt ra thuật ngữ trie, phát âm nó là /ˈtriː/ (như "tree") sau âm tiết giữa của retrieval. Tuy nhiên các tác giả khác phát âm nó là /ˈtraɪ/ (như "try") nhằm phân biệt bằng lời nói với từ "tree" được sử dụng rộng rãi để chỉ định nghĩa khác.

4.Giới thiệu về cấu trúc dữ liệu Trie/Prefix Tree.

Trie, hay một số tài liệu còn gọi là **cây tiền tố**, là một cấu trúc dữ liệu dạng **cây** hữu dụng được dùng để quản lí một tập hợp các xâu. Mặc dù dễ hiểu và dễ cài đặt, trie lại có rất nhiều ứng dụng. Do vậy, trie thường xuyên xuất hiện trong các cuộc thi lập trình ở Việt Nam nói riêng và quốc tế nói chung.

Một trie cơ bản có thể thực hiện ba thao tác sau với độ phức tạp thời gian tuyến tính:

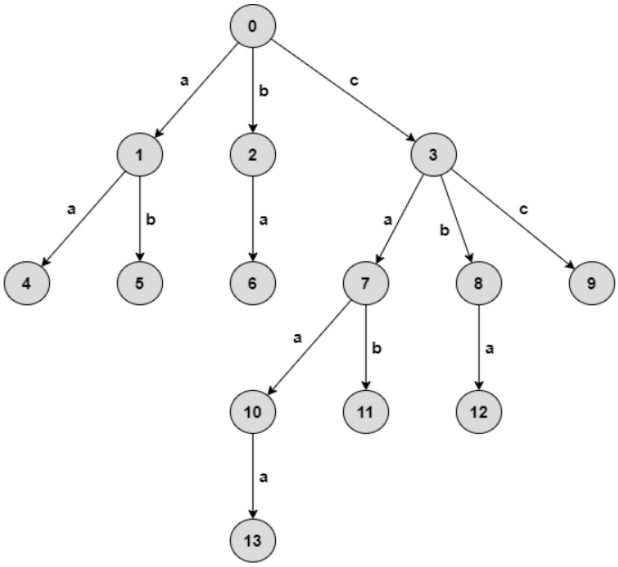
- Thêm một xâu vào tập hợp
- Xóa một xâu khỏi tập hợp
- Kiểm tra một xâu có nằm trong tập hợp đó hay không

Ta đã biết rằng Trie là một cấu trúc dữ liệu dạng cây, sử dụng rất hiệu quả trong các bài toán liên quan tới quản lý danh sách xâu kí tự. Tuy nhiên, ít ai biết rằng Trie Tree còn có một ứng dụng nữa, đó là quản lý một tập số nguyên.

5.Tính chất và cấu trúc của Trie.

Trie là một cấu trúc dữ liệu dạng cây dùng để lưu trữ một danh sách các xâu với bộ kí tự hữu hạn, cho phép việc lưu trữ các xâu hiệu quả có tiền tố giống nhau.

Hãy xem xét một ví dụ sau:



Trong một trie, mỗi cạnh được biểu diễn bằng một kí tự, mỗi đỉnh và đường đi từ gốc đến đỉnh đó biểu diễn một xâu gồm các kí tự thuộc các cạnh trên đường đi đó. Ví dụ, đỉnh 5 biểu diễn xâu `ab`, đỉnh 10 biểu diễn xâu `caa`.

Cấu trúc của trie rất dễ hiểu và cài đặt.
Gọi `child(u, c)` là đỉnh con của đỉnh u được nối bởi cạnh được biểu diễn bằng kí tự c, hoặc bằng −1 nếu đỉnh con đó không tồn tại. Xâu được thể hiện bởi đỉnh con này sẽ chính là xâu được thể

hiện bởi đỉnh u, thêm kí tự c vào cuối. Do vậy, ta chỉ cần mảng `child` này với mỗi đỉnh để duy trì cấu trúc của trie. Ví dụ, trong ảnh trên, `child(1, 'b') = 5`, `child(3, 'c') = 9`, `child(11, 'b') = -1`. [1]

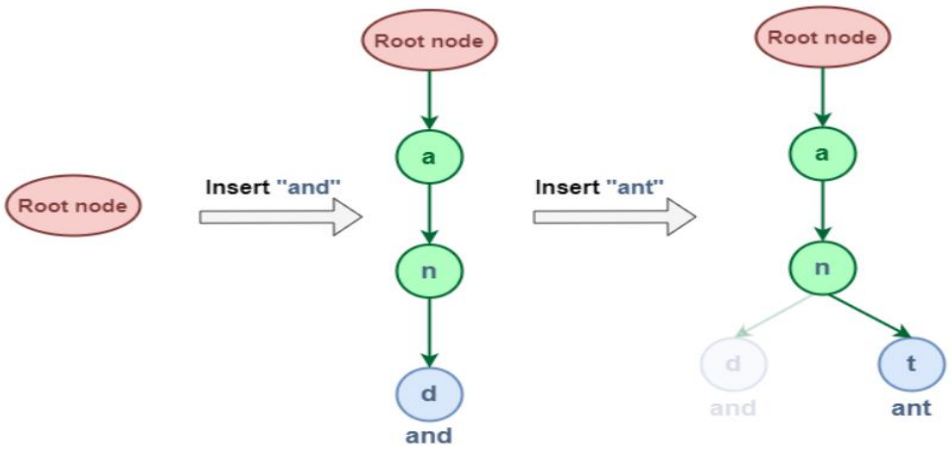
6.Các thao tác chính trên Trie

Cấu **trúc dữ liệu Trie** bao gồm các nút được kết nối bởi các cạnh. Mỗi nút đại diện cho một ký tự hoặc một phần của chuỗi. Nút gốc, điểm bắt đầu của Trie, đại diện cho một chuỗi rỗng. Mỗi cạnh phát ra từ một nút biểu thị một ký tự cụ thể. Đường dẫn từ gốc đến nút biểu thị tiền tố của chuỗi được lưu trữ trong Trie.

Một nút trong cây Trie sẽ lưu các thành phần sau:

- Các nút con, hay còn gọi là các giá trị mà nó có thể nối đến, trong trường hợp cây Trie xâu thì sẽ lưu 26 nút con tương ứng với các kí tự từ ‘a’ đến ‘z’.
- *wordEnd* của một nút dùng để xác định số xâu kết thúc tại nút này.
- *Cnt* của một nút dùng để xác định số xâu đi qua nút này

6.1 Thao tác chèn:



[4]

Chèn xâu “and” vào cây Trie: (Insert “and” in Trie data structure)

- Bắt đầu tại nút gốc (root node)
- Kí tự đầu tiên ‘a’, ban đầu nút gốc chưa có đường dẫn đến kí tự tiếp theo là ‘a’ nên ta sẽ tạo nút mới đồng thời tăng *cnt* tại nút này lên 1 đơn vị. Ta di chuyển đến nút này.
- Kí tự thứ hai ‘n’, ban đầu nút hiện tại của ta chưa có đường dẫn đến kí tự tiếp theo là ‘n’ nên ta sẽ tạo nút mới đồng thời lại tăng cnt tại nút này lên 1 đơn vị. Ta di chuyển đến nút này.

- Kí tự cuối cùng ‘d’, đường dẫn vẫn chưa xuất hiện cho kí tự tiếp theo ‘d’ nên ta sẽ vẫn tạo nút mới và tăng cnt tại nút lên 1, nhưng tại đây là vị trí cuối cùng của xâu ta thêm vào nên ta sẽ tăng *wordEnd* của nút này lên 1, chứng tỏ đã có thêm 1 xâu nữa kết thúc tại nút này.

Chèn xâu “ant” vào cây Trie: (Insert “ant” in Trie data structure)

- Bắt đầu tạo nút gốc (root node)
- Kí tự đầu tiên ‘a’, nút gốc đã có đường dẫn đến kí tự tiếp theo là ‘a’ nên ta sẽ không tạo nút mới mà tăng *cnt* tại nút này lên 1 đơn vị. Ta di chuyển vào nút đã tồn tại này.
- Kí tự thứ hai ‘n’, nút hiện tại của ta cũng đã có đường dẫn đến kí tự tiếp theo là ‘n’, vậy ta sẽ lại tăng cnt lên 1 đơn vị và di chuyên vào nút này.
- Kí tự cuối cùng ‘t’, nút hiện tại ta đang đứng có đường dẫn đến kí tự ‘d’ do thao tác thêm xâu “and” trước đó nhưng kí tự tiếp theo là ‘t’ thì chưa, nên ta sẽ tạo một nút mới ở đây, tăng cnt của nút này lên và đồng thời tăng *wordEnd* của nút này vì nó là kí tự cuối cùng trong xâu. Ta di chuyển vào nút này.

6.2 Thao tác xóa:

1.1 Không làm thay đổi cấu trúc của cây (cài đặt bằng mảng hoặc vector)

Tương tự như thao tác chèn, ta xuất phát từ gốc của cây, đi xuống các nút theo thứ tự các kí tự trong xâu cần xóa, mỗi nút ta giảm cnt đi 1 đơn vị, vì đã mất 1 xâu đi qua nút này. Khi đến nút cuối cùng của xâu cần xóa thì ta giảm wordEnd đi 1 đơn vị, chứng tỏ đã mất đi 1 xâu kết thúc tại nút này.

1.2 Làm thay đổi cấu trúc của cây (cài đặt bằng con trỏ)

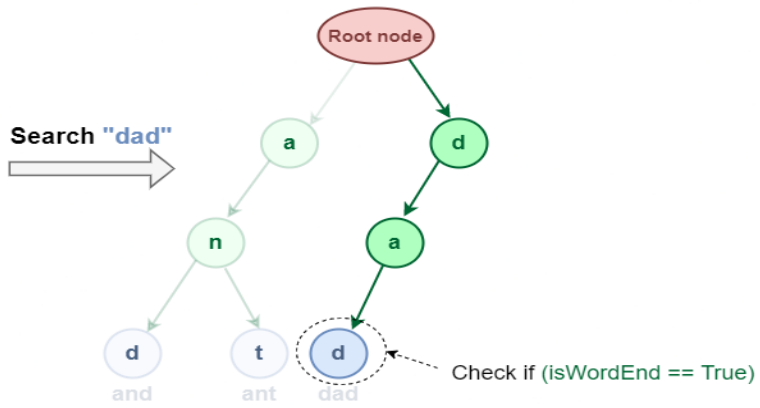
Ta thực hiện như mục trên, nhưng khi ta xét xong 1 nút mà cnt của nút đó bằng 0, chứng tỏ không còn xâu nào đi qua nút này nữa, ta xóa nút này đi.

6.3 Thao tác đếm, tìm kiếm:

Ta vẫn xuất phát từ gốc của cây, đi xuống các nút theo thứ tự các kí tự trong xâu cần tìm kiếm, khi đi đến nút cuối cùng thì giá trị wordEnd ở đó sẽ là số lần mà xâu ta cần tìm xuất hiện trong cây Trie.

Minh họa:

Giả sử ta đã chèn thành công các xâu “and”, “ant” và “dad” vào cây Trie, và bây giờ chúng ta cần tìm kiếm xâu “dad”:



- Chúng ta bắt đầu ở nút gốc.
- Chúng ta đi theo nhánh tương ứng với kí tự ‘d’.
- Chúng ta đi theo nhánh tương ứng với kí tự ‘a’.
- Chúng ta đi theo nhánh tương ứng với kí tự ‘d’.
- Chúng ta đến cuối xâu và nút hiện tại có giá trị wordEnd là 1. Điều này có nghĩa là xâu “dad” có mặt trong cây Trie. [4]

7.Cách cài đặt Trie

Có hai cách cài đặt Trie là cài đặt bằng mảng hoặc cài đặt bằng con trỏ (hay còn gọi là **Trie động**). Dưới đây sẽ trình bày cả hai cách, tuy nhiên lời khuyên là các bạn nên cài đặt bằng con trỏ, bởi vì ba lí do dưới đây**:**

- Không cần tính toán bộ nhớ mảng, tránh lãng phí dữ liệu.
- Dễ dàng cài đặt nhiều Trie một lúc bằng cách tạo một gốc mới khi cần một Trie mới.
- Có thể xóa một nút không cần thiết nữa trên Trie.

Nếu định nghĩa cấu trúc như phần trước, ta chỉ có thể thực hiện truy vấn thêm xâu vào tập hợp. Để thực hiện hai truy vấn còn lại, với mỗi đỉnh **u** trong trie, ta lưu thêm hai biến:

- `exist`: có bao nhiêu xâu là xâu được thể hiện bởi đỉnh **u**
- `cnt`: có bao nhiêu xâu có tiền tố là xâu được thể hiện bởi đỉnh **u**

Lưu ý: Tuy nhiên với từng bài toán, hai biến này có thể không cần thiết và có thể bỏ đi.

7.1 Cài đặt bằng mảng

```
const int max_nodes = ...;

struct Trie
{
    struct Node
    {
        int child[26];   // mảng lưu địa chỉ các nút con của nút hiện tại
        int exist, cnt;  // exist sẽ lưu vị trí cuối cùng của các xâu kết thúc
                         // tại đỉnh này, cnt là số các xâu đi qua đỉnh này
    } nodes[max_nodes];

    int cur = 0; // Hiện trong trie đang có bao nhiêu đỉnh.
```

```
    // Tạo nút gốc cho Trie là đỉnh 0 khi khởi tạo Trie.
    Trie() : cur(0)
    {
        memset(nodes[0].child, -1, sizeof(nodes[cur].child));
        nodes[0].exist = nodes[0].cnt = 0;
    };

    // Tạo và trả về giá trị của đỉnh mới được tạo ra.
    int new_node()
    {
        ++cur;
        memset(nodes[cur].child, -1, sizeof(nodes[cur].child));
        nodes[cur].exist = nodes[cur].cnt = 0;

        return cur;
    }

    // Thêm xâu s vào cây Trie
    void add_string(string s)
    {
        int pos = 0;
        for (auto f : s)
        {
            int c = f - 'a';
            // Nếu cạnh tương ứng chữ cái c chưa tồn tại thì ta tạo ra đỉnh mới.
            if (nodes[pos].child[c] == -1)
                nodes[pos].child[c] = new_node();

            // Có thêm một xâu trong trie có tiền tố là xâu được thể hiện
            // bằng đỉnh hiện tại.
            pos = nodes[pos].child[c];
            ++nodes[pos].cnt;
        }

        // Đã tìm được đỉnh tương ứng với xâu s, ta tăng biến exist của đỉnh lên
1.
        ++nodes[pos].exist;
    }

    // Trả về liệu đỉnh pos có thể bị xóa đi hay không, đồng thời xóa xâu s nếu
có thể.
    bool delete_string_recursive(int pos, string& s, int i)
    {
        // Nếu chưa đến đỉnh tương ứng với xâu s thì tiếp tục đệ quy xuống dưới.
        if (i != (int) s.size())
```

```
        {
            int c = s[i] - 'a';
            bool is_child_deleted = delete_string_recursive(nodes[pos].child[c],
s, i + 1);
            // Nếu đỉnh con tương ứng bị xóa thì ta gán lại đỉnh tương ứng bằng -
1.
            if (is_child_deleted)
                nodes[pos].child[c] = -1;
        }
        // Nếu đã đến đỉnh tương ứng với xâu s thì ta giảm biến exist của đỉnh đi
1
        else
            --nodes[pos].exist;

        // Nếu đỉnh đang xét không phải gốc thì ta giảm biến cnt của đỉnh đi 1
        // và kiểm tra đỉnh có bị xóa đi hay không?
        // Đỉnh bị xóa nếu không còn xâu nào đi qua nó, nói cách khác là
        // không còn xâu nào có tiền tố là xâu được thể hiện bởi đỉnh pos.
        if (pos != 0)
        {
            --nodes[pos].cnt;

            if (nodes[pos].cnt == 0)
                return true;
        }

        return false;
    }

    // Thủ tục:
    // Truyền vào xâu s.
    // Thủ tục sẽ nhận xâu s và kiểm tra xâu x có trong Trie hay không.
    // Nếu có thì xóa xâu s ra khỏi Trie.
    void delete_string(string s)
    {
        // Kiểm tra xâu s có trong Trie hay không.
        if (find_string(s) == false)
            return;

        // Gọi hàm đệ quy xóa xâu s khỏi Trie.
        delete_string_recursive(0, s, 0);
    }

    // Hàm:
    // Nhận vào xâu s.
```

```
    // Hàm sẽ trả về "true" nếu có xâu s trong cây Trie,
    // ngược lại trả về false.
    bool find_string(string s)
    {
        int pos = 0;
        for (auto f: s)
        {
            int c = f - 'a';
            if (nodes[pos].child[c] == -1)
                return false;

            pos = nodes[pos].child[c];
        }

        // Kiểm tra có xâu nào kết thúc tại đỉnh này hay không.
        return (nodes[pos].exist != 0);
    }
};
```

[2] Nguồn: Viblo Algorithm **Trie Tree (phần 1) - Xử lý xâu kí tự**

7.2 Cài đặt bằng con trỏ:

Mặc dù có nhiều ưu điểm, nhưng cài đặt bằng con trỏ cũng có nhiều nhược điểm và dễ nhầm lẫn nếu các bạn chưa vững kiến thức về cấp phát bộ nhớ động.

Nhược điểm:

- Dễ code sai nếu không thực sự hiểu con trỏ là gì.
- Tùy thuộc vào compiler mà con trỏ có thể còn tốn nhiều bộ nhớ hơn dùng mảng.

Vì thế, trước khi quyết định sử dụng con trỏ để cài đặt Trie, hãy xem lại các kiến thức dưới đây:

- Tham chiếu, Địa chỉ ảo và Con trỏ trong C++.
- Hoạt động nâng cao với con trỏ C++.
- Kĩ thuật Cấp phát bộ nhớ động.

Gần như mọi phần trong đoạn code dưới hoạt động giống phần cài đặt bằng mảng nên sẽ không chú thích lại.

```
struct Trie
```

```
{
    struct Node
    {
        Node* child[26];
        int exist, cnt;

        Node()
        {
            for (int i = 0; i < 26; i++)
                child[i] = NULL;

            exist = cnt = 0;
        }
    };

    int cur;
    Node* root;
    Trie() : cur(0)
    {
        root = new Node();
    };

    void add_string(string s)
    {
        Node* p = root;
        for (auto f: s)
        {
            int c = f - 'a';
            if (p -> child[c] == NULL)
                p -> child[c] = new Node();

            p = p -> child[c];
            p -> cnt++;
        }

        p -> exist++;
    }

    bool delete_string_recursive(Node* p, string& s, int i)
    {
        if (i != (int) s.size())
        {
            int c = s[i] - 'a';
            bool is_child_deleted = delete_string_recursive(p -> child[c], s, i +
1);
```

```
            if (is_child_deleted)
                p->child[c] = NULL;
        }
        else
            p -> exist--;

        if (p != root)
        {
            p -> cnt--;
            if (p -> cnt == 0)
            {
                // Khác với cài đặt bằng mảng, ta có thể thực sự xóa đỉnh này đi.
                delete(p);

                return true;
            }
        }
        return false;
    }

    void delete_string(string s)
    {
        if (find_string(s) == false)
            return;

        delete_string_recursive(root, s, 0);
    }

    bool find_string(string s)
    {
        Node* p = root;
        for (auto f: s)
        {
            int c = f - 'a';
            if (p -> child[c] == NULL)
                return false;

            p = p -> child[c];
        }

        return (p -> exist != 0);
    }
};
```

[2] **Nguồn: Viblo Algorithm** Trie Tree (phần 1) - Xử lý xâu kí tự

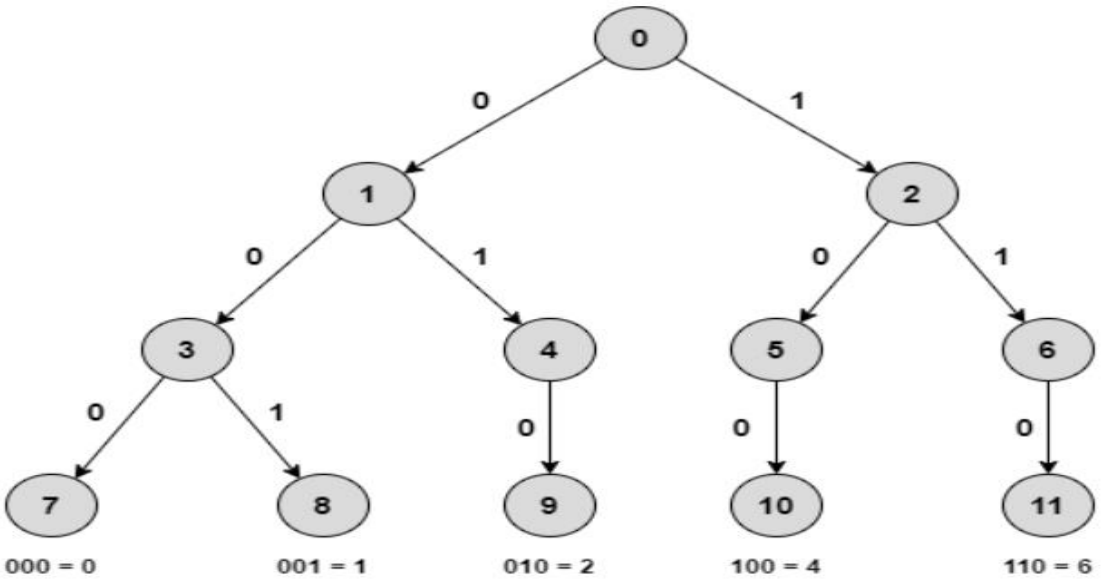
8.Trie nhị phân

8.1 Giới thiệu chung

Ta đã biết rằng Trie là một cấu trúc dữ liệu dạng cây, sử dụng rất hiệu quả trong các bài toán liên quan tới quản lý danh sách xâu kí tự. Tuy nhiên, ít ai biết rằng Trie Tree còn có một ứng dụng nữa, đó là quản lý một tập số nguyên.

Bằng cách coi mỗi số là một xâu kí tự gồm toàn kí tự 0 và 1 (tức là biểu diễn nhị phân của số đó), ta có thể quản lý tập hợp số này cùng với những thao tác hỗ trợ để xử lý những bài toán về số nguyên.

Dưới đây là một ví dụ minh họa về Trie nhị phân quản lý tập các số nguyên $\{0,1,2,4,6\}$.



[3]

Trie nhị phân có một số đặc điểm sau:

- Các số được thêm vào Trie sẽ được chuyển qua dạng nhị phân, rồi thêm các bit 0 vào đầu sao cho độ dài của chúng đều bằng nhau. Thông thường độ dài này sẽ được đặt là $\log(\max(a_i))$ với a_i là các số trong tập hợp.
- Một số nguyên sẽ bao gồm các bit trên đường đi từ nút gốc tới nút lá của Trie Tree (mỗi cạnh lưu một bit).
- Khi thêm các bit vào Trie, ta thêm theo chiều từ trái sang phải.

8.2 Cài đặt

Dưới đây là cách cài đặt bằng con trỏ:

```
const int LOG = ...; // Giá trị lớn nhất của log(a[i]).

struct Trie
{
    struct Node
    {
        Node* child[2];
        int exist, cnt;

        Node()
        {
            for (int i = 0; i < 2; ++i)
                child[i] = NULL;

            exist = cnt = 0;
        }
    };

    Node* root;
    Trie() : cur(0)
    {
        root = new Node();
    };

     void add_number(int x)
     {
        Node* p = root;

        for (int i = LOG; i >= 0; --i)
        {
            int c = (x >> i) & 1;
            if (p -> child[c] == -1)
                p -> child[c] = new Node();

            p = p -> child[c];
            p -> cnt++;
        }

        p -> exist++;
    }
```

```
    void delete_number(int x)
    {
        if (find_number(x) == false)
            return;

        Node* p = root;
        for (int i = LOG; i >= 0; --i)
        {
            int c = (x >> i) & 1;
            Node* tmp = p -> child[c];
            tmp -> cnt--;

            if (tmp -> cnt == 0)
            {
                p -> child[c] = -1;
                return;
            }

            p = tmp;
        }

        p -> exist--;
    }

    bool find_number(int x)
    {
        Node* p = root;
        for (int i = LOG; i >= 0; --i)
        {
            int c = (x & (1 << i) ? 1 : 0);

            if (p -> child[c] == NULL)
                return false;

            p = p -> child[c];
        }

        return (p -> exist != 0);
    }
};
```
[3]

9.Độ phức tạp

9.1 Thời gian:

Thời gian của các bài toán Trie sẽ là O(m) trong đó m là số kí tự trong các xâu được thêm vào Trie Xâu, hoặc tổng số bit trong các số được thêm vào Trie nhị phân.

9.2Bộ nhớ:

9.2.1 Nếu như cài đặt bằng mảng:

- Trong Trie Xâu: Vì một nút hiện tại có thể trỏ đến 26 nút con khác biểu diễn kí tự từ 'a' đến 'z' nên ta phải xác định bộ nhớ trước cho mảng là $O(m * 26)$ trong đó m là tổng số kí tự được thêm vào Trie.
- Trong Trie nhị phân: Nút hiện tại có trỏ đến 2 giá trị 0 hoặc 1 nên bộ nhớ trước cho mảng là O(m * 2).

9.2.2 Nếu như cài đặt bằng con trỏ:

- Mặc dù nút hiện tại có thể trỏ đến 1 hoặc nhiều nút khác nhưng các nút chưa được thêm vào sẽ nhận giá trị NULL vì thế mà chỉ có nút được truy cập thì mới được cấp phát động. Vậy bộ nhớ khi cài bằng con trỏ là O(m), trong đó m là tổng số kí tự, bit được thêm vào cây.

10.Ứng dụng

10.1 Trong lập trình thi đấu:

Đây là một bài toán điển hình sử dụng trie nhị phân. Đa số các bài toán liên quan tới thao tác bit sử dụng trie đều là biến thể của bài toán này.

10.1.1 Đề bài:

Cho n số nguyên không âm $a_1, a_2, \dots, a_n$ và m truy vấn , mỗi truy vấn yêu cầu tìm giá trị:

$$\max_{i=1}^{n} (a_i \oplus x)$$

với x là một số nguyên không âm cho trước, $\oplus$ là toán tử XOR bit

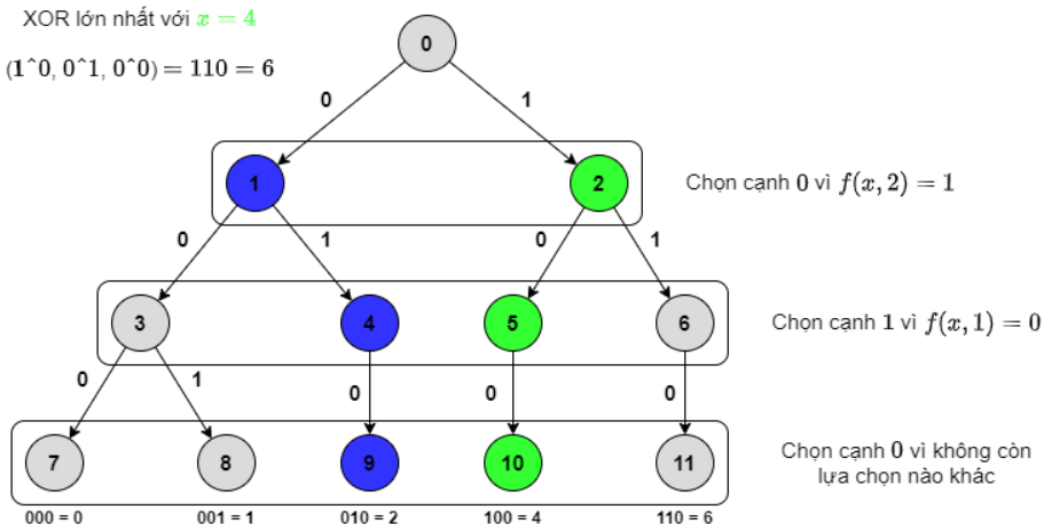
Yêu cầu: Xử lý tất cả các truy vấn?

Ý tưởng:

Đầu tiên xây dựng một Trie nhị phân với các số nguyên đã cho.

Xét lần lượt các bit từ lớn đến bé của đáp án. Coi bit đang xét là bit thứ i. Ta sẽ xây dựng đáp án một cách tham lam bằng cách cố gắng đặt bit thứ i của đáp án là 1 do $2^i > \sum_{i=0}^{j-1} 2^j$. Nói cách khác, dù đặt cả $i-1$ bit còn lại của đáp áp là 1 thì cũng không có lợi bằng việc đặt bit i là 1.

Ta sẽ lần lượt xây đáp án bằng các đi xuống từ gốc của Trie. Giả sử ta đang xây dựng bit thứ i của đáp án. Nếu đỉnh hiện tại đang xét có thể đi xuống cạnh có bit là $f(x,i) \oplus 1$ với $f(x,i)$ là bit thứ i của số x, ta sẽ đi qua cạnh đó để có được bit i trong đáp án là 1. Nếu không, ta "đành" đi xuống cạnh còn lại của đỉnh đang xét và có được bit i của đáp án là 0.



10.1.2 Cài đặt:

```
// Cài đặt bằng con trỏ, chỉ cần thêm hàm này vào trong cấu trúc Trie nhị phân.
int query(int x)
{
    int res = 0;
    Node* p = root;

    for (int i = LOG; i >= 0; --i)
    {
        int c = (x >> i) & 1;

        if (p -> child[c ^ 1] != -1)
        {
            res += (1ll << i);
            p = p -> child[c ^ 1];
        }
        else
            p = p -> child[c];
    }

    return res;
}
```

Bên cạnh đó, trie còn có rất nhiều ứng dụng trong lập trình thi đấu như: tìm xâu tiền tố, xắp xếp các xâu theo thứ tự từ điển, các bài toán liên quan đến phép xor,…

10.1.3 Các bài toán tham khảo :

Trie xâu

SPOJ - Ada and Indexing (Dễ)

SPOJ - Try to complete (Dễ)

IOI 2008 - Type Printer (Dễ)

Codeforces Gym - Know Your Statement (Trung bình)

VOI 2021 - Phần thưởng (Trung bình)

Atcoder - Prefix-tree Game (Khó)

PVHOI 2.2 - Tiền tố chung dài nhất (Khó)

Trie nhị phân

Hackerrank - XOR Key (Dễ)

SPOJ - SubXor (Dễ)

SPOJ - x-Xor It! (Dễ)

CSAcademy - Xor Submatrix (Trung bình)

Hackerrank - The Black Box (Khó)

10.2 Trong thực tế

9.3 Tìm kiếm từ điển: Trie thường được sử dụng để lưu trữ và tìm kiếm từ điển, nơi mỗi cạnh trên cây đại diện cho một kí tự, mỗi đỉnh và đường đi từ gốc đến đỉnh đó đại diện cho một xâu kí tự.
9.4 Kiểm tra tính đúng đắn của từ vựng.
9.5 Tìm kiếm IP routing.
9.6 Tìm kiếm dạng autocomplete.

11.Tài liệu tham khảo:

[1] https://vnoi.info/wiki/algo/data-structures/trie.md#gi%E1%BB%9Bi-thi%E1%BB%87u

[2] https://viblo.asia/p/trie-tree-phan-1-xu-ly-xau-ki-tu-5pPLk96n4RZ

[3] https://viblo.asia/p/trie-tree-phan-2-trie-nhi-phan-zXRJ8m3qLGq

[4] https://www.geeksforgeeks.org/trie-insert-and-search/

12.Phụ lục

12.1. Chương trình minh họa

https://www.cs.usfca.edu/~galles/visualization/Trie

12.2. Link tham khảo code (github)

https://github.com/tanprodium/TRIE-PREFIX-TREE