Chương 5 - Cây

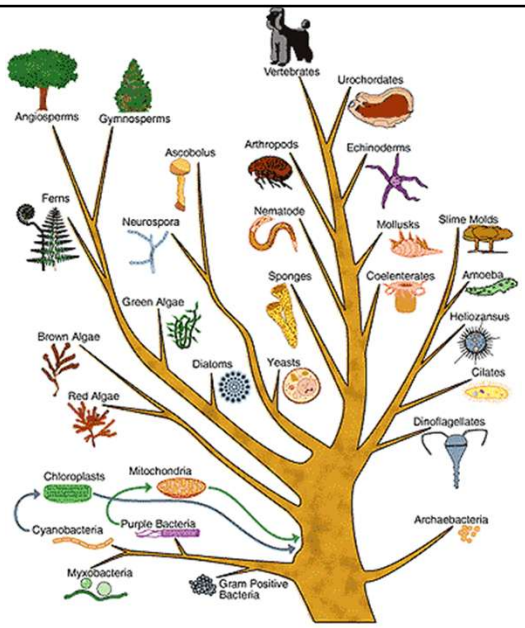
Cây tổng quát, biểu diễn cây, duyệt cây, prefix tree,
cây nhị phân, suffix tree

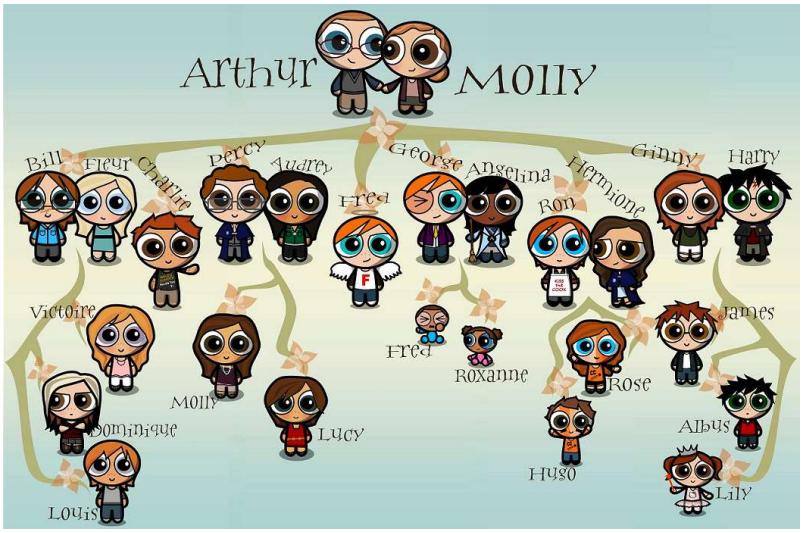
Nội dung

- Khái niệm cơ bản
- Biểu diễn cây tổng quát
- Một số bài toán cơ bản trên cây tổng quát
- Cây nhị phân
- Ứng dụng của cây nhị phân
- Một số bài toán trên cây nhị phân
- Một số mở rộng của cây tổng quát

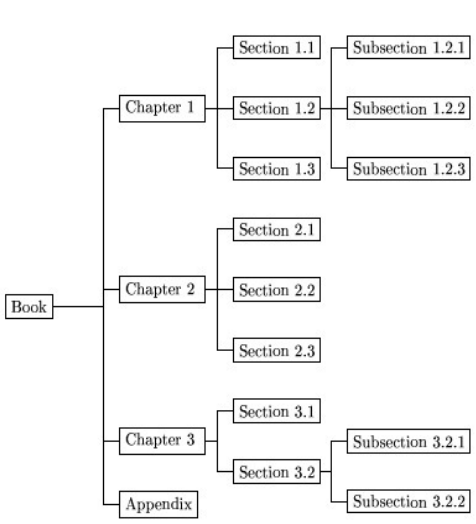
Khái niệm

- Cây - tree
- Là kiểu cấu trúc dữ liệu phi tuyến
- Biểu diễn mối *quan hệ có thứ bậc* giữa các đối tượng
- Quan hệ giữa các nút là quan hệ cha-con (tổ tiên- con cháu) hoặc ngang hàng





- Cây phả hệ - family tree

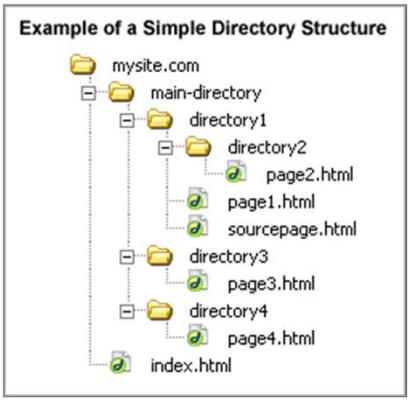


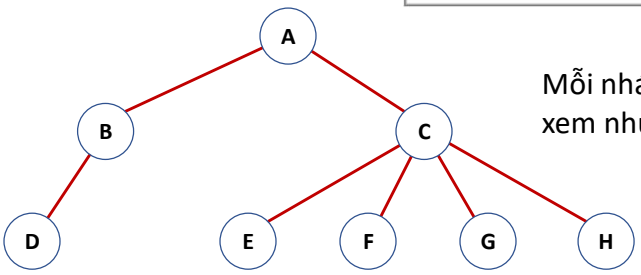
Mục lục sách

Cây thư mục

Khái niệm cơ bản

- Các nút ở mức ngoài cùng được gọi là *nút lá – leaf node*
- Các nút không phải nút lá được gọi là *nút trong – internal node*
- *Nút gốc* – là nút mà không có nút nào đứng trên nó

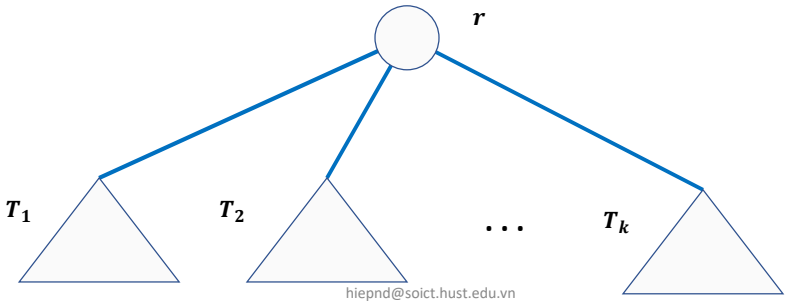




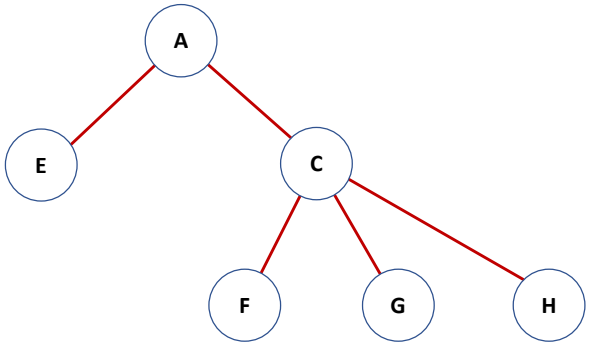
Mỗi nhánh có thể được xem như là một cây con

Khái niệm cơ bản

- Định nghĩa: Cây là một tập hợp gồm các nút.
  - Tập này có thể rỗng,
  - nếu không thì nó bao gồm một nút $\boldsymbol{r}$ được gọi là gốc và một tập rỗng hoặc khác rỗng các cây con $\boldsymbol{T_1}, \boldsymbol{T_2}, .., \boldsymbol{T_k}$ mà có nút gốc được nối trực tiếp bằng một cạnh từ nút gốc r



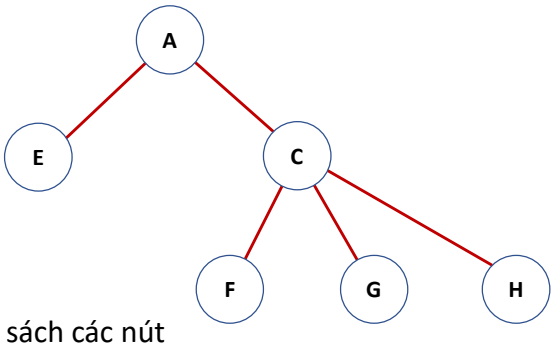
Khái niệm cơ bản



- Các nút cùng cha là anh em (sibling)
  - E và C, hoặc F, G và H
- Nút gốc: là nút không có nút nào đứng trên
  - A là gốc
- Nút lá: nút không có nút con
  - E, F, G, H
- Nút trong: Nút có ít nhất 1 con
  - A,C

Cây tổng quát: 1 nút có thể không có con, hoặc có thể có vô số con

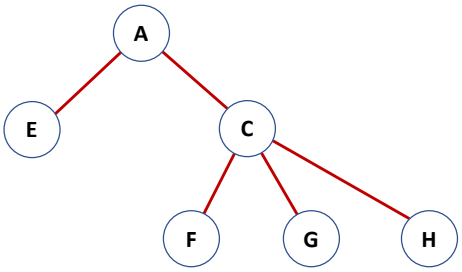
Khái niệm cơ bản



- Đường đi từ nút n_i tới nút n_j trên cây là một danh sách các nút $n_i, n_{i+1}, .., n_j$ trong đó nút n_i là cha của nút n_{i+1}.
- Độ dài đường đi: là số cạnh trên đường đi.
  - Đường đi qua n đỉnh thì có độ dài $n - 1$
  - Đường đi từ nút đến chính nó có độ dài là 0

- Đường đi từ A đến H là A, C, H có độ dài 2
- ~~Đường đi từ C đến C có độ dài 0~~

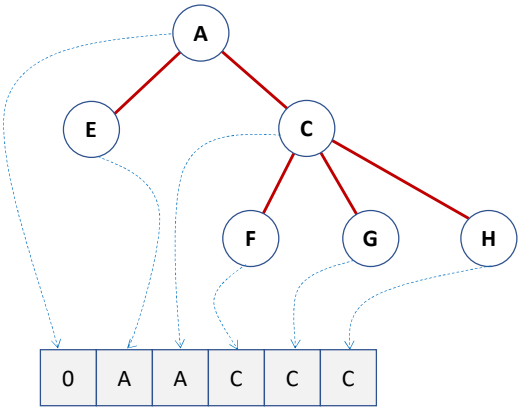
Khái niệm cơ bản



- Độ sâu/mức (depth/level) của nút: độ dài đường đi từ nút gốc đến chính nó
  - Độ sâu của nút gốc A là 0, của nút G, H là 2
- Độ cao (height) của nút: là độ dài đường đi lớn nhất từ nó đến một nút lá
  - Độ cao của nút lá E, F, G, H là 0, của nút A là 2
- Độ cao của cây: là độ cao của nút gốc
- Độ sâu của cây: là độ sâu lớn nhất của nút lá trên cây

Biểu diễn cây

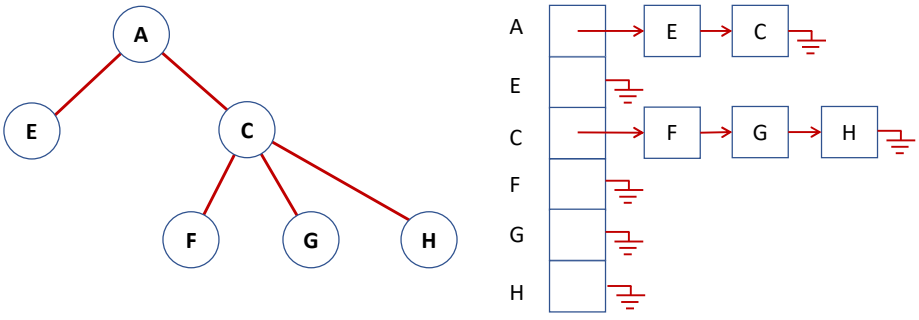
- Biểu diễn cây: thông qua nhãn nút cha của nó (cấu trúc liên tiếp)



- Lưu dần các nút theo mức, mức 0 sẽ lưu đầu tiên
- Tìm nút cha của nút hiện tại nhanh
- Thêm/xóa nút phức tạp, thường phải dịch phần tử

Biểu diễn cây

- Biểu diễn cây: danh sách các nút con (mảng của các danh sách liên kết)



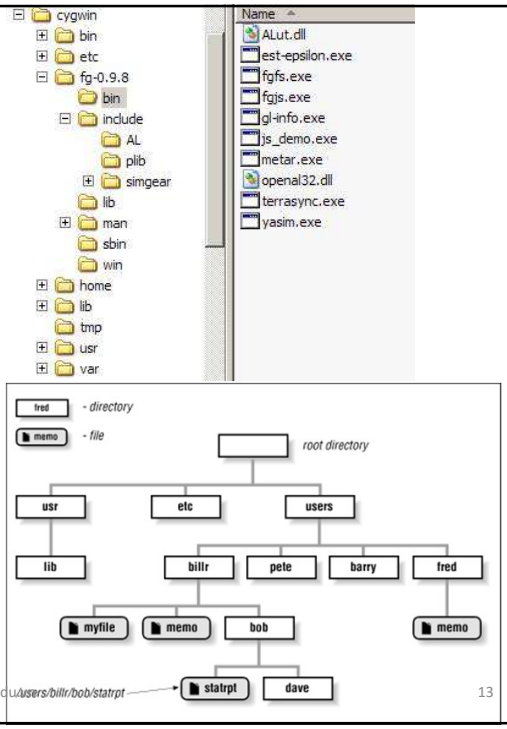
- Danh sách các nút con của nút hiện tại biểu diễn bằng danh sách liên kết đơn
- Thông thường, con trái nhất sẽ đứng đầu danh sách
- Nút không có con thì danh sách tương ứng là rỗng

Biểu diễn cây

- Biểu diễn cây: Con đầu tiên và các nút anh chị em kế tiếp (cấu trúc liên kết)

```
struct TreeNode
{
    DATA_TYPE info;
    struct TreeNode *firstChild;
    struct TreeNode *nextSibling;
}
```

Trong thực tế bổ sung thêm con trỏ tới nút cha để thêm và xóa nút nhanh hơn



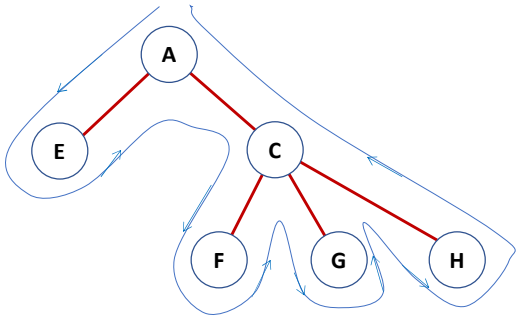
Duyệt cây

- **Thuật toán duyệt cây:**
  - Phân biệt bởi việc thăm nút con trước hay là xử lý dữ liệu tại nút hiện tại trước
  - Các nút con thông thường sẽ được thăm theo thứ tự con trái nhất trước sau đó đến các nút con kế tiếp
  - Cây tổng quát có các phương án duyệt cây thông dụng
    - Duyệt theo thứ tự trước – gọi đệ quy hoặc dùng stack
    - Duyệt theo thứ tự sau – gọi đệ quy hoặc dùng stack
    - Duyệt theo mức – dùng hàng đợi

Duyệt cây

- **Duyệt cây theo thứ tự trước** (pre-order traversal): xử lý dữ liệu trên nút hiện tại trước, sau đó xử lý tiếp đến các nút con của nó

```
preOrder(root){
   if(root == NULL) return;
   process(root);
   for each p = r1, r2, .., rk {
      preOrder(p);
   }
}
```

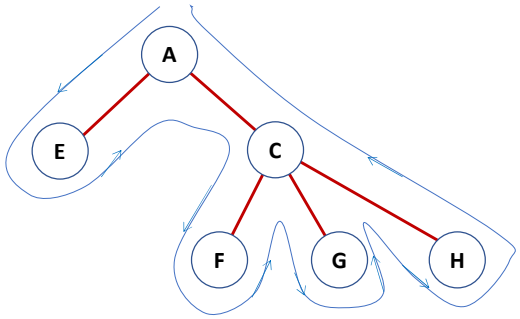


In ra các đỉnh khi duyệt theo thứ tự trước: A, E, C, F, G, H

Khái niệm

- Duyệt cây theo thứ tự sau (post-order traversal): xử lý dữ liệu trên các nút con trước sau đó mới xử lý dữ liệu trên nút hiện tại

```
postOrder(root){
   if(root == NULL) return;
   for each p = r1, r2, .., rk {
      postOrder(p);
   }
   process(root);
}
```



In ra các đỉnh khi duyệt theo thứ tự sau: E, F, G, H, C, A

Thao tác với cây tổng quát

- find(r, label): tìm và trả về nút có nhãn label trên cây có gốc là r
- insert(r, label): tạo một nút có nhãn label, chèn vào cuối danh sách nút con của nút r
- height(r): trả về độ cao của nút r
- depth(r): trả về độ sâu của nút r
- parent(r): trả về nút cha của nút r
- count(r): trả về số nút trên cây có gốc là r
- countLeaves(r): trả về số nút lá trên cây có gốc là r

```
#include <stdio.h>
#include <stdlib.h>
#include <queue>

using namespace std;
typedef struct TREENODE
{
        char label;
        struct TREENODE* firstChild;
        struct TREENODE* nextSibling;
} TreeNode;
```



```
// hàm tạo nút mới và cấp phát động bộ nhớ
TreeNode* makeNode(char label)
{
        TreeNode* newNode =
            (TreeNode*)malloc(sizeof(TreeNode));
        newNode->label = label;
        newNode->firstChild = NULL;
        newNode->nextSibling = NULL;
        return newNode;
}
```

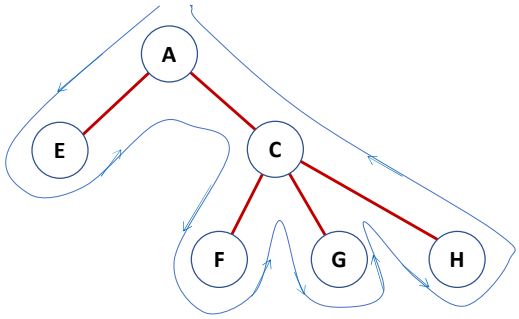
```
// thêm nút mới vào cây, nút mới thêm là newNode
// nút mới sẽ được thêm vào con tiếp theo của nút gốc root
TreeNode* insertNode(TreeNode* root, TreeNode* newNode)
{
        // TH1. nút gốc hiện tại rỗng --> nút mới là root
        if (root == NULL) {
                return root  = newNode;
        }
        // TH2. Gốc hiện tại KHông có con, nút mới sẽ là con đầu tiên
        else if (root->firstChild == NULL) {
                root->firstChild = newNode;
        }
        else // TH3. đã có con, nút mới sẽ là con phải nhất
        {
                // tìm con phải nhất hiện tịa của gốc
                TreeNode* rightmost = root->firstChild;
                while (rightmost->nextSibling != NULL){
                        rightmost = rightmost->nextSibling;
                }
                // nút mới sẽ được thêm vào sau con phải nhất
                rightmost->nextSibling = newNode;
        }
        return root;
}
```

```
// chen nut da duoc tao newNode vao nut con moi nhat (phai nhat)
// cua nut root, ham khong can tra ve gia tri
void insertNode2(TreeNode** root, TreeNode* newNode)
{
	if (*root == NULL) *root = newNode;
	else if ((*root)->firstChild == NULL) (*root)->firstChild = newNode;
	else {
		TreeNode* rightmost = (*root)->firstChild;
		while (rightmost->nextSibling != NULL)rightmost = rightmost->nextSibling;
		rightmost->nextSibling = newNode;
	}
}
```

```
// tao va chen nut moi thanh cay con phai nhat cua nut root
// Chu ý: root phai khac NULL
void insertNode_v2(TreeNode* root, char label)
{
	if (root == NULL) return;
	TreeNode* newNode = makeNode(label);
	if (root->firstChild == NULL) root->firstChild = newNode;
	else {
		TreeNode* rightmost = root->firstChild;
		while (rightmost->nextSibling != NULL) rightmost = rightmost->nextSibling;
		rightmost->nextSibling = newNode;
	}
}
```



```
// duyet cay theo muc
void levelTraversal(TreeNode* root)
{
	if (NULL == root) return;
	queue<TreeNode*> Q;
	Q.push(root);

	while (Q.size() > 0)
	{
		TreeNode* p = Q.front();
		Q.pop();
		printf("%c, ", p->label);
		TreeNode* q = p->firstChild;
		while (NULL != q)
		{
			Q.push(q);
			q = q->nextSibling;
		}
	}
	printf("\n");
}
```




```
int main()
{
	TreeNode* root = makeNode('A');
	insertNode_v2(root, 'E');
	insertNode_v2(root, 'C');
	insertNode_v2(root->firstChild->nextSibling, 'F');
	insertNode_v2(root->firstChild->nextSibling, ‘G');
	insertNode_v2(root->firstChild->nextSibling, ‘H');
	levelTraversal(root);
	return 0;
}
```

```
// duyet cay theo thu tu truoc
void preOrderTravesal(TreeNode* root)
{
        if (NULL == root) return;
        printf("%c, ", root->label);
        TreeNode* p = root->firstChild;
        while (NULL != p)    {
                preOrderTravesal(p);
                p = p->nextSibling;
        }
}
```

```
// duyet cay theo thu tu sau
void postOrderTravesal(TreeNode* root)
{
        if (NULL == root) return;
        TreeNode* p = root->firstChild;
        while (NULL != p)    {
                postOrderTravesal(p);
                p = p->nextSibling;
        }
        printf("%c, ", root->label);
}
```

```
// dem so luong nut tren cay
int countNodes(TreeNode* root)
{
        if (NULL == root) return 0;
        TreeNode* p = root->firstChild;
        int sum = 0;
        while (NULL != p)    {
                sum += countNodes(p);
                p = p->nextSibling;
        }
        return sum + 1;
}
```

```
int countLeaves(TreeNode* root) {
        if (root == NULL) return 0;
        int sum = 0;
        TreeNode* p = root->firstChild;
        if (p == NULL) sum = 1;
        while (p != NULL) {
                sum += countLeaves(p);
                p = p->nextSibling;
        }
        return sum;
}
```



```
// tim nut cha cua nut p tren cay co goc la root
TreeNode* parent(TreeNode* root, TreeNode* p) {
        if (root == NULL) return NULL;
        TreeNode* q = root->firstChild;
        while (q != NULL) {
                if (p == q) return root;
                TreeNode* pp = parent(q, p);
                if (pp != NULL) return pp;
                q = q->nextSibling;
        }
        return NULL;
}
```

```
// tinh chieu cao cua nut (chiều cao la la 1)
int height(TreeNode* root) {
        if (root == NULL) return 0;
        int maxh = 0;
        TreeNode* q = root->firstChild;
        while (q != NULL) {
                int h = height(q);
                if (h > maxh) maxh = h;
                q = q->nextSibling;
        }
        return maxh + 1;
}
```

```
// tinh chieu cao cua nut
// coi nut la chieu cao la 0
int height(TreeNode* root) {
	if (root == NULL) return -1; // cay rong
	TreeNode* q = root->firstChild;
	if (NULL == q) return 0; // nut hien tai la la
	int maxh = 0;
	while (q != NULL) {
		int h = height(q);
		if (h > maxh) maxh = h;
		q = q->nextSibling;
	}
	return maxh + 1;
}
```

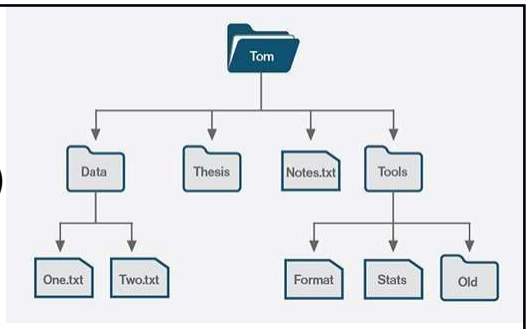


```
// Tim do sau nut co nhan label tren cay goc la root
// dung de quy
int depth(TreeNode* root, char label, int d) {
	// d la do sau cua nut root
	if (root == NULL) return -1;
	if (root->label == label) return d;
	TreeNode* p = root->firstChild;
	while (p != NULL) {
		if (p->label == label) return d + 1;
		int dv = depth(p, label, d + 1);
		if (dv > 0) return dv;
		p = p->nextSibling;
	}
	return -1;
}

int calDepth(TreeNode* root, char v) {
	return depth(root, v, 0);
}
```

```
// Tim do sau nut p tren cay goc la root dung de quy
// d la do sau cua nut root

int depth(TreeNode* root, TreeNode* p, int d) {
	if (root == NULL) return -1;
	if (root == p) return d;
	TreeNode* q = root->firstChild;
	while (q != NULL) {
		if (q == p) return d + 1;
		int dv = depth(q, p, d + 1);
		if (dv > 0) return dv;
		q = q->nextSibling;
	}
	return -1;
}

int calDepth(TreeNode* root, TreeNode* v) {
	return depth(root, v, 0);
}
```

```
// tinh do sau cua nut co nhan label, không de quy
int depth(TreeNode* root, char label) {
        if (root == NULL) return -1;
        queue<pair<TreeNode*,int>> Q;
        Q.push(make_pair(root,0));

        int currentDepth;
        TreeNode* p;
        while (Q.size()>0) {
                p = Q.front().first;
                currentDepth = Q.front().second;
                Q.pop();
                if (p->label == label) return currentDepth;
                p = p->firstChild;
                while (p != NULL) {
                        Q.push(make_pair(p, currentDepth+1));
                        p = p->nextSibling;
                }
        }
        return -1;
}
```


Ứng dụng



- Quản lý hệ thống Hệ thống tệp tin (File System)
  - Tổ chức
    - Nút gốc: thư mục gốc (/ trong Linux hoặc C:\ trong Windows)
    - Nút trong: thư mục hoặc tệp tin con
    - Nút lá: là tệp tin
  - Thao tác
    - Duyệt BFS hoặc DFS
    - Thêm/xóa, tìm kiếm
    - Sao chép và di chuyển
    - Đường dẫn từ thư mục cha đến con
    - Quản lý quyền truy cập dựa trên cấu trúc phân cấp: Quyền truy cập có thể được kế thừa từ thư mục cha xuống các thư mục và tệp tin con
    - Dùng cây cân bằng giúp truy cập nhanh hơn:
      - NTFS (Windows): Sử dụng cấu trúc cây B+ (B+ Tree)
      - EXT4 (Linux): Sử dụng cấu trúc hash tree và B+ Tree

Ứng dụng

- hệ thống quản lý nội dung (CMS - Content Manageme
  - Quản lý cấu trúc trang web phân cấp
  - Tổ chức và phân loại nội dung: phân loại nội dung thành các danh mục hoặc chuyên mục, giúp dễ dàng truy cập
  - Quản lý quyền truy cập phân cấp
  - Quản lý phiên bản nội dung (Content Versioning): Khi nội dung thay đổi, mỗi phiên bản có thể được coi như một nhánh mới trong cây
  - Tìm kiếm và sắp xếp nội dung: hệ thống có thể duyệt theo từng nhánh, giúp người dùng tìm kiếm bài viết hoặc nội dung liên quan nhanh chóng
  - Tổ chức nội dung cho đa ngôn ngữ: Trang chủ (Tiếng Anh) và Trang chủ (Tiếng Việt) có các mục tương ứng trong mỗi nhánh
  - Phân phối nội dung động: hệ thống có thể hiển thị nội dung cho người dùng dựa trên vị trí của họ trong cây nội dung

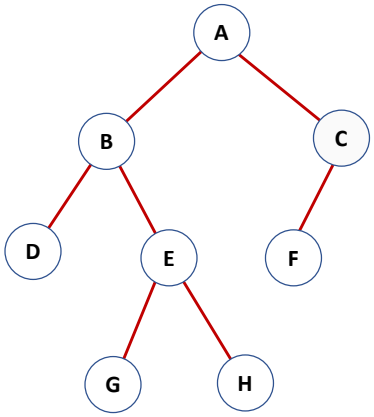
Ứng dụng

- Một số ứng dụng khác
  - Biểu diễn biểu thức toán học hoặc logic (thường là cây nhị phân)
  - Biểu diễn tài liệu XML hoặc JSON: mỗi nút là 1 tag
  - Hệ thống phân loại và tổ chức dữ liệu (Ontology và Taxonomy): VD. Trong sinh học, chúng ta có các cấp bậc như giới, ngành, lớp, bộ, họ, chi, loài. Trong việc làm có phân loại theo lĩnh vực, nghề nghiệp
  - Cây quyết định (Decision Tree) trong học máy (Machine Learning)
  - Hệ thống trò chơi và AI (Game Trees)
  - Biểu diễn ngữ pháp trong trình biên dịch (Syntax Tree): phân tích cú pháp của mã nguồn
  - Mạng xã hội và các mối quan hệ phức tạp

Cây nhị phân

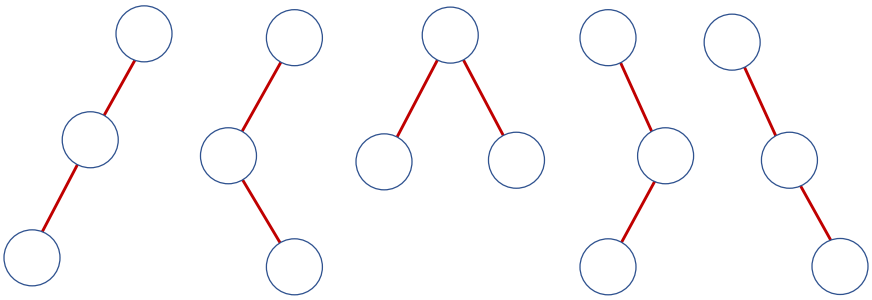
Cây nhị phân

- Cây nhị phân – binary tree:
  - Là một tập rỗng, hoặc
  - Có một nút gốc với hai cây nhị phân con của gốc gọi là cây con trái (left subtree) và cây con phải (right subtree)



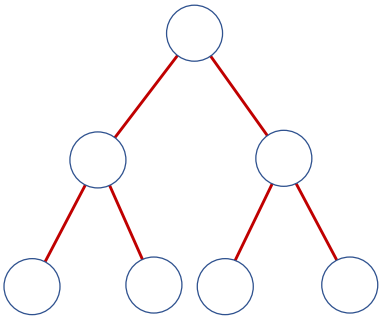
Cây nhị phân

- Một số cây nhị phân gồm 3 nút khác nhau



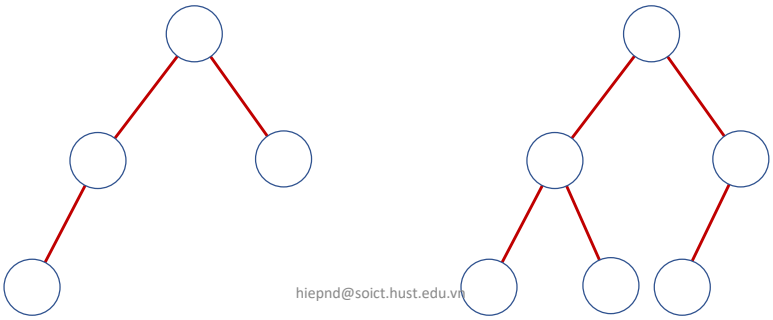
Cây nhị phân

- Cây nhị phân đầy đủ (full binary tree):
  - Các nút không phải là lá đều có 2 nút con
  - Các nút lá có độ sâu bằng nhau
- Cây nhị phân đầy đủ chiều cao h có bao nhiêu nút lá?
- Cây nhị phân đầy đủ có bao nhiêu nút?



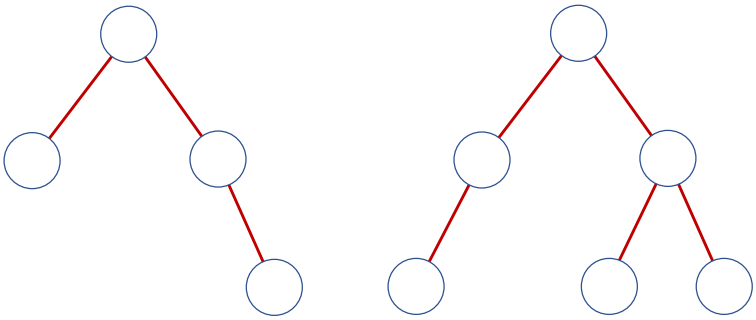
Cây nhị phân

- Cây nhị phân hoàn chỉnh (complete binary tree) chiều cao h
  - Các nút từ mức 0 tới $h-1$ là cây nhị phân đầy đủ
  - Một số nút ở mức $h-1$ có thể có 0 hoặc 1 con
  - Nếu i, j là các nút ở mức $h-1$, thì i có nhiều con hơn j nếu i nằm ở bên trái j



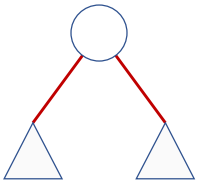
Cây nhị phân

- Cây nhị phân cân bằng (balanced binary tree): là cây nhị phân thỏa mãn
  - Với mọi nút thì chênh lệch chiều cao của cây con trái và cây con phải không quá 1
  - Mỗi cây nhị phân hoàn chỉnh là cây cân bằng



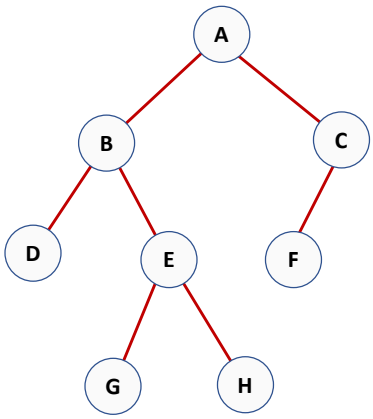
Duyệt cây – tree traversal

- Mỗi nút trên cây nhị phân gồm :
  - Giá trị chứa tại nút
  - Cây con trái
  - Cây con phải

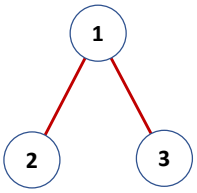


- Duyệt cây theo thứ tự
  - Thứ tự trước – preorder: Giá trị → con trái → con phải
  - Thứ tự giữa – inorder : con trái → giá trị → con phải
  - Thứ tự sau – postorder : con trái → con phải → giá trị

Duyệt cây – tree traversal



- Thứ tự trước : 1, 2, 3
- Thứ tự giữa : 2, 1, 3
- Thứ tự sau : 2, 3, 1



Thứ tự trước : A, B, D, E, G, H, C, F

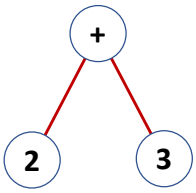
Thứ tự giữa : D, B, G, E, H, A, F, C

Thứ tự sau : D, G, H, E, B, F, C, A

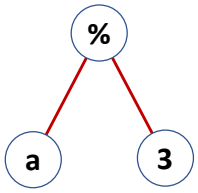
Cây biểu thức – expression tree

- *Cây biểu thức – expression tree*: xây dựng từ các toán tử và toán hạng
  - Toán tử 2 ngôi: gốc là toán tử, 2 nút con lần lượt là toán hạng trái và phải

2+3



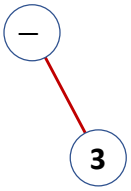
a%3



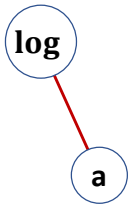
Cây biểu thức – expression tree

- Toán tử 1 ngôi: nút gốc biểu diễn toán tử, chỉ có 1 nút con biểu diễn toán hạng

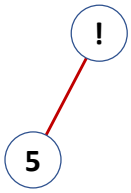
-3



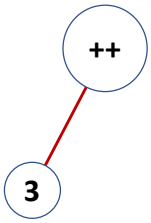
$\log a$



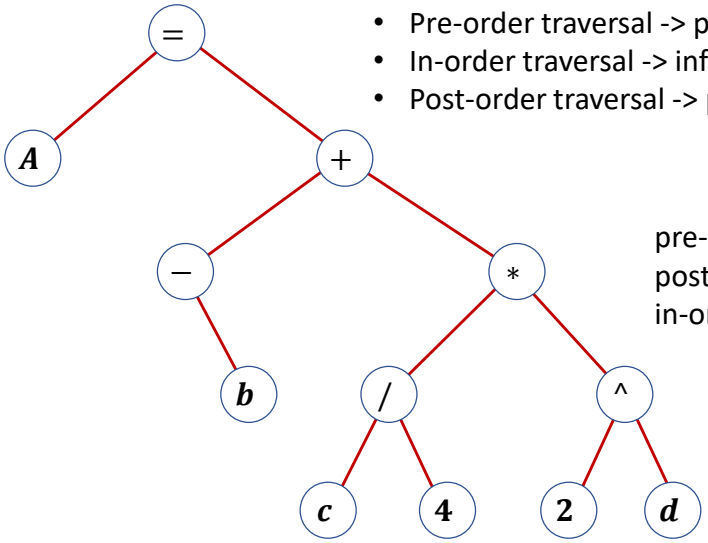
5!



3++



$$A = -b + c/4 * 2\text{^}d$$



- Pre-order traversal -> prefix notation
- In-order traversal -> infix notation
- Post-order traversal -> postfix notation

pre-order : = A + - b * / c 4 ^ 2 d
post-order : A b - c 4 / 2 d ^ * + =
in-order : A = - b + c / 4 * 2 ^ d

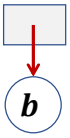
Cây biểu thức – expression tree

- Thuật toán xây dựng cây biểu thức từ biểu thức dạng hậu tố
- Đọc lần lượt các phần tử của biểu thức hậu tố
- Gặp toán hạng: tạo ra cây có 1 nút là toán hạng và đẩy vào stack
- Gặp toán tử:
  - Nếu là toán tử 1 ngôi: lấy ra 1 cây T trong stack, tạo ra 1 cây trong đó toán tử là nút gốc và cây T là nút con, sau đó đẩy cây này vào stack
  - Nếu là toán tử 2 ngôi: lấy ra 2 cây trong stack T_1, T_2 (T_2 được lấy ra trước), tạo cây trong đó toán tử này là gốc và T_1, T_2 lần lượt là con trái và con phải của nó, sau đó đẩy cây này vào stack
- Sau khi duyệt xong biểu thức hậu tố, cây biểu thức là cây có gốc nằm ở đỉnh của stack

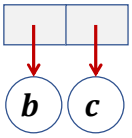
$b\ c + a * D -$

Ban đầu stack rỗng

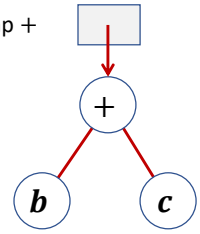
Gặp b



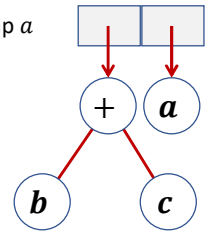
Gặp c



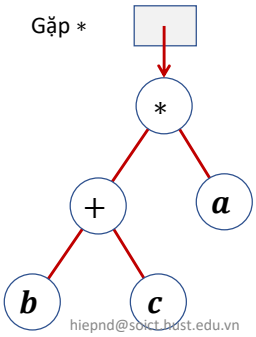
Gặp $+$



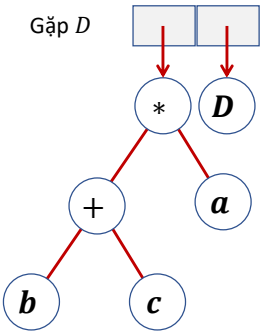
Gặp a



Gặp $*$

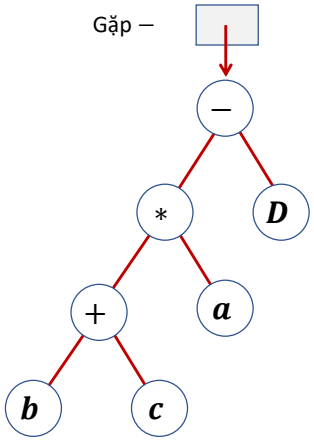


Gặp D

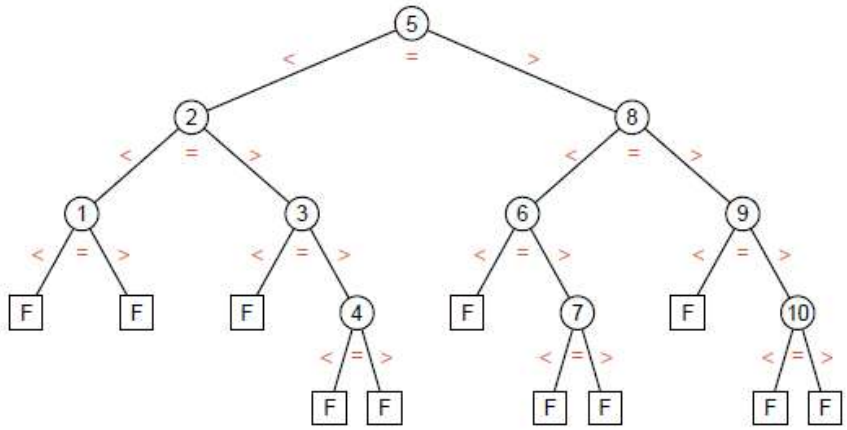


$b\ c + a * D -$

Gặp $-$



Cây quyết định, cây tìm kiếm nhị phân

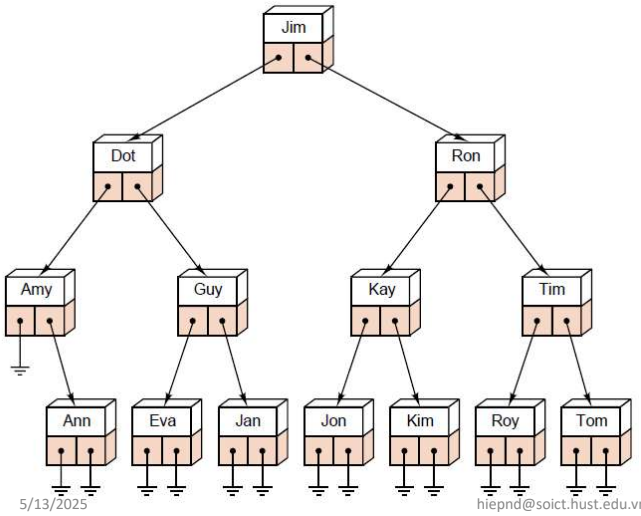


Cây quyết định cho thuật toán tìm kiếm nhị phân trên dãy số 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Biểu diễn cây nhị phân

- Biểu diễn cây nhị phân
- Dùng cấu trúc liên tiếp (mảng): số lượng nút trên cây chiều cao h là giới hạn ($\sum 2^i$)
  - Truy nhập nhanh $O(1)$
  - Lãng phí bộ nhớ lớn nếu cây chưa phải là dạng đầy đủ hoặc hoàn chỉnh
- Dùng cấu trúc liên kết: thường dùng hơn
  - Truy cập 1 nút chậm hơn
  - Tiết kiệm bộ nhớ hơn (trong trường hợp tổng quát)

Biểu diễn cây nhị phân




```
struct TreeNode
{
    DATA_TYPE info;
    struct TreeNode *leftChild;
    struct TreeNode *rightChild;
}
```

- Biểu diễn bằng cấu trúc liên kết

Biểu diễn cây nhị phân

- Duyệt cây theo thứ tự trước/giữa/sau

```
struct TreeNode
{
  int data;
  struct TreeNode *leftChild;
  struct TreeNode *rightChild;
}
```

```
void preorderTraversal(TreeNode* root)
{
    if(root==NULL) return;
    printf("%d ",root->data);
    preorderTraversal(root->leftChild);
    preorderTraversal(root->rightChild);
}
```

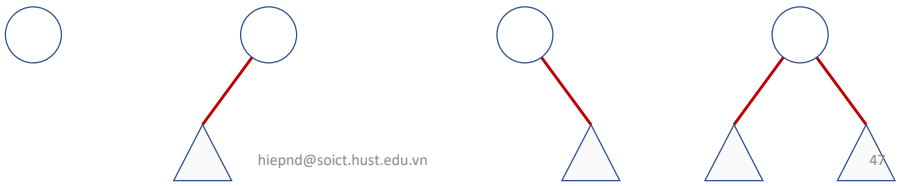
```
void inorderTraversal(TreeNode* root)
{
    if(root==NULL) return;
    inorderTraversal(root->leftChild);
    printf("%d ",root->data);
    inorderTraversal(root->rightChild);
}
```

```
void postorderTraversal(TreeNode* root)
{
    if(root==NULL) return;
    postorderTraversal(root->leftChild);
    postorderTraversal(root->rightChild);
    printf("%d ",root->data);
}
```

Cài đặt thuật toán trên cây nhị phân

- Cây nhị phân có 5 trường hợp khi xử lý đệ quy
  - Cây rỗng
  - Nút gốc là nút lá
  - Nút gốc chỉ có con trái
  - Nút gốc chỉ có con phải
  - Nút gốc có đủ 2 con
- Tùy các bài toán mà có thể gộp các trường hợp lại với nhau

NULL



Ứng dụng

- Cây nhị phân tìm kiếm (Binary Search Tree - BST): O(logn)
  - Cây AVL và Cây Đỏ-Đen (Balanced Binary Trees)
- Cây biểu thức (Expression Tree)
- Cây Huffman trong mã hóa dữ liệu:
  - Hay dùng trong các hệ thống nén như ZIP, JPEG, MP3, và PNG
- Cây quyết định (Decision Tree)
- Cây nhị phân tìm kiếm trong các hệ quản trị cơ sở dữ liệu
  - cây B+ (một dạng mở rộng cây nhị tìm kiếm phân thành cây đa cấp – nhiều nhánh)
- Tổ chức bộ nhớ và phân bổ bộ nhớ (Memory Allocation)
  - Hệ thống Buddy Memory Allocation phân chia bộ nhớ thành các khối có kích thước lũy thừa của 2, và các khối được tổ chức dưới dạng cây nhị phân để phân bổ và giải phóng bộ nhớ hiệu quả.

Ứng dụng

- Các hệ thống phân cấp và quản lý quyền truy cập:
  - Tra cứu quyền dựa trên cây nhị phân
- Hệ thống đánh chỉ mục văn bản (Text Indexing)
  - cây nhị phân tìm kiếm tiền tố giúp tăng tốc độ tìm kiếm các từ khóa trong văn bản
- Cây Khung Nhỏ Nhất (Minimum Spanning Tree - MST)
  - Cây tìm kiếm nhị phân (Binary Search Tree - BST) hoặc Heap để sắp xếp và truy xuất các cạnh
  - Cấu trúc Union-Find với cây nhị phân trong thuật toán Kruskal

Cây nhị phân

- Bài 1. Tìm nút có giá trị lớn nhất/ nhỏ nhất trên cây nhị phân
- Bài 2. Thêm/xóa nút trên cây nhị phân
- Bài 3. Kiểm tra 2 cây nhị phân đồng dạng
- Bài 4. Kiểm tra 2 cây nhị phân đối xứng
- Bài 5. Thêm nút mới vào cây theo mức lần lượt từ trái sang phải (tương tự trên cây tổng quát)
- Bài 6. In ra đường đi dài nhất trên cây từ gốc tới lá
- Bài 7. In ra nút lá nông nhất/sâu nhất trên cây
- Bài 8. Cho cây nhị phân mà giá trị tại nút là 0 hoặc 1 (gốc luôn là 1), in ra đường đi chứa toàn số 1 dài nhất từ gốc
- Bài 9. Định giá cây biểu thức

Cây nhị phân

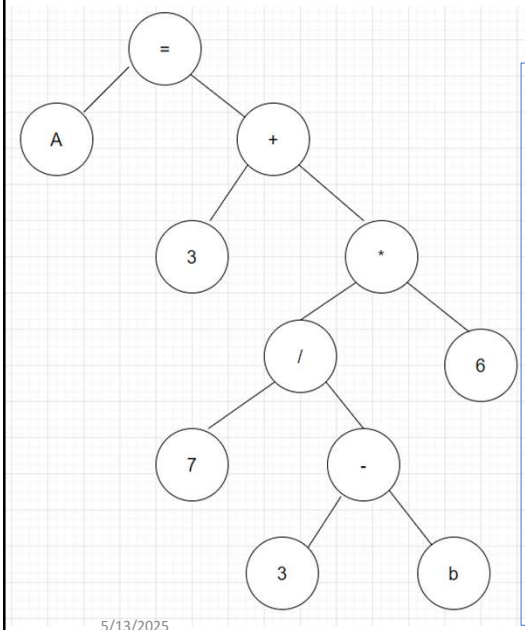
- Bài 10. Cho cây nhị phân trong đó mỗi nút trong có 2 con và nút lá, cho thứ tự duyệt cây trước là B C D E A F G và thứ tự giữa C B A E F D G, hãy đưa ra thứ tự duyệt sau
- Bài 11. Tìm nút tổ tiên gần nhất của 2 nút trên cây
- Bài 12. kiểm tra xem các nút lá trên cây có phải cùng mức
- Bài 13. Cho 2 cây T1 và T2, kiểm tra xem T2 có phải là cây con của T1
- Bài 14. Cho cây nhị phân với nút là số nguyên, in ra các đường đi từ gốc có tổng giá trị ≤k
- Bài 15. Loại các nút bị trùng lặp trên cây nhị phân


```
#include <stdio.h>
#include <stdlib.h>
typedef struct TREENode
{
        char label;
        struct TREENode* left;
        struct TREENode* right;
} TreeNode;

TreeNode* makeNode(char label)
{
        TreeNode* newNode =
           (TreeNode*)malloc(sizeof(TreeNode));
        newNode->label = label;
        newNode->left = NULL;
        newNode->right = NULL;
        return newNode;
}
```

```
// tao va chen nut moi thanh cay con phai nhat cua nut root
// Chu y: root phai khac NULL
// leftorRight = 0==> la con trai, 1 la con phai
void insertNode_v2(TreeNode* root, char label, int leftorRight)
{
        if (root == NULL) return;
        TreeNode* newNode = makeNode(label);
        if (leftorRight == 0) root->left = newNode;
        else root->right = newNode;
}


void preorderTraversal(TreeNode* root)
{
        if (root == NULL) return;
        printf("%c ", root->label);
        preorderTraversal(root->left);
        preorderTraversal(root->right);
}
```

```
void inorderTraversal(TreeNode* root)
{
	if (root == NULL) return;
	inorderTraversal(root->left);
	printf("%c ", root->label);
	inorderTraversal(root->right);
}

void postorderTraversal(TreeNode* root)
{
	if (root == NULL) return;
	postorderTraversal(root->left);
	postorderTraversal(root->right);
	printf("%c ", root->label);
}
```

```
int countNode(TreeNode* root)
{
	if (NULL == root) return 0;
	return 1 + countNode(root->left) + countNode(root->right);
}

int countLeaves(TreeNode* root)
{
	if (NULL == root) return 0;
	if (NULL == root->left && NULL == root->right) return 1;
	return countLeaves(root->left) + countLeaves(root->right);
}

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


```
int main()
{
	TreeNode* root = makeNode('=');
	insertNode_v2(root, 'A', 0);
	insertNode_v2(root, '+', 1);
	insertNode_v2(root->right, '3', 0);
	insertNode_v2(root->right, '*', 1);
	insertNode_v2(root->right->right, '/', 0);
	insertNode_v2(root->right->right, '6', 1);
	insertNode_v2(root->right->right->left, '7', 0);
	insertNode_v2(root->right->right->left, '-', 1);
	insertNode_v2(root->right->right->left->right, '3', 0);
	insertNode_v2(root->right->right->left->right, 'b', 1);
	inorderTraversal(root);
	printf("\nSo luong nut: %d\n", countNode(root));
	printf("\nSo luong nut la: %d\n", countLeaves(root));
	printf("\nSo luong nut trong: %d\n", countInternalNodes(root));
	return 0;
}
```

```
// duyet cay theo muc
void levelTraversal(TreeNode* root)
{
	if (NULL == root) return;
	queue<TreeNode*> Q;
	Q.push(root);

	while (Q.size() > 0)
	{
		TreeNode* p = Q.front();
		Q.pop();
		printf("%c, ", p->label);
		if (NULL != p->left) Q.push(p->left);
		if (NULL != p->right) Q.push(p->right);
	}
	printf("\n");
}
```

```
// duyet cay theo muc
void printNodesBylevel(TreeNode* root, int depthLevel)
{
	if (NULL == root) return;

	queue<pair<TreeNode*, int>> Q;
	Q.push(make_pair(root, 0));
	int currDepth = 0;
	while (Q.size() > 0)
	{
		TreeNode* p = Q.front().first;
		currDepth = Q.front().second;
		Q.pop();
		if (currDepth == depthLevel) printf("%c, ",
							p->label);
		if (NULL != p->left) Q.push(make_pair(p->left,
							currDepth + 1));
		if (NULL != p->right) Q.push(make_pair(p->right,
							currDepth + 1));
	}
	printf("\n");
}
```


Cây mã Huffman

Cây Huffman là một cây nhị phân dùng để **mã hóa dữ liệu nén không mất thông tin**.

- Xây dựng dựa trên **tần suất xuất hiện** của các ký tự trong dữ liệu
- **mã nhị phân ngắn nhất cho ký tự xuất hiện nhiều nhất**, và dài hơn cho các ký tự ít gặp.

Ưu điểm của cây Huffman

- Mã hóa hiệu quả hơn so với mã ASCII (cố định 8 bit).
- Phù hợp để nén văn bản, file nhị phân, ảnh…

Ứng dụng

- Nén dữ liệu trong ZIP, GZIP, PNG.
- Mã hóa truyền dữ liệu.
- Giao thức truyền tin có băng thông hạn chế.

Cây mã Huffman

Ví dụ. Mã hóa từ khóa CAPYBARA MEME

Ký tự	Tần số
A	3
B	1
C	1
E	2
M	2
P	1
R	1
Y	1
<space>	1

Tạo nút lá cho từng ký tự và đưa vào danh sách
Lặp:
- Lấy lần lấy ra 2 nút có tần số nhỏ nhất
- Tạo nút gộp (tần số bằng tổng tần số của 2 nút)
- Đưa nút gộp trở lại danh sách
- Lặp cho đến khi chỉ còn 1 nút

Trường hợp có nhiều nút có cùng tần số, chọn ngẫu nhiên hoặc theo thứ tự ABC (cho dễ kiểm soát và debug)

Cây mã Huffman

- Ví dụ. Mã hóa từ khóa CAPYBARA MEME

Ký tự	Tần số
A	3
B	1
C	1
E	2
M	2
P	1
R	1
Y	1
<space>	1

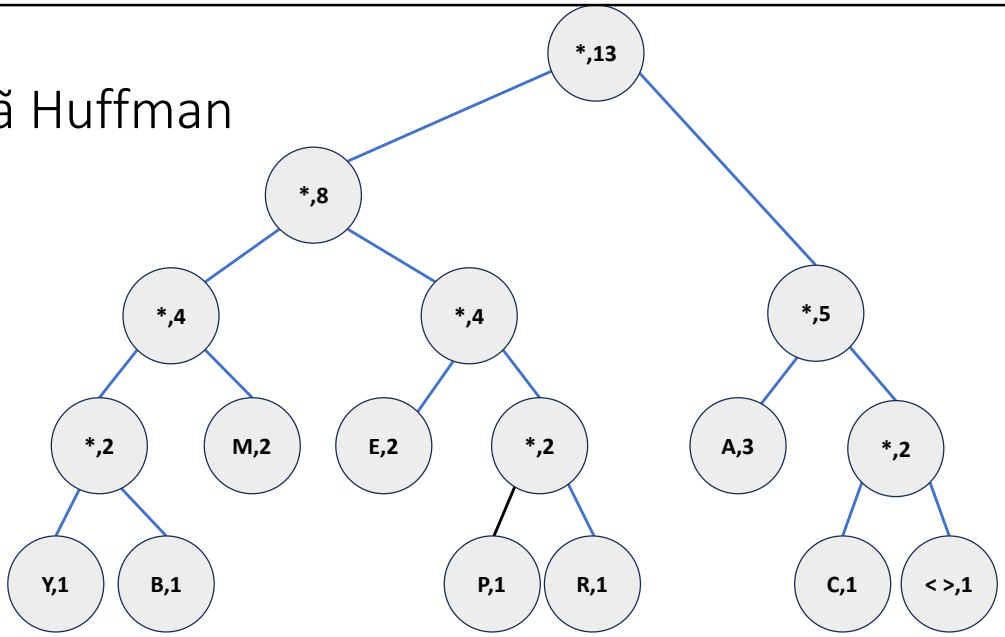
Trong cài đặt thực tế, dùng minheap để biểu diễn danh sách (hàng đợi ưu tiên tần số nhỏ nhất)

Bước 1: Tạo nút lá cho từng ký tự, đưa vào hàng đợi ưu tiên (min-heap) dựa trên tần suất.
Bước 2: Duyệt theo vòng lặp:
- Lấy ra 2 nút có **tần suất nhỏ nhất**.
- Tạo **nút cha mới** với tần suất bằng tổng 2 nút vừa lấy.
- Gán nút con trái và phải cho nút cha.
- Đưa nút cha trở lại hàng đợi.

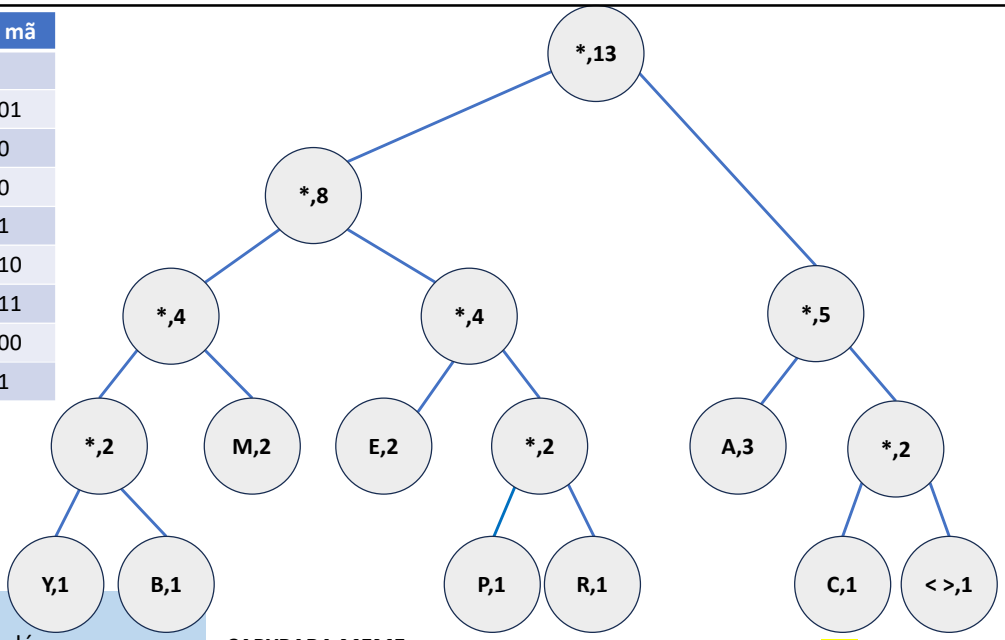
Bước 3: Khi còn 1 nút duy nhất → đó là gốc cây Huffman.

Cây mã Huffman

Ký tự	Tần số
A	3
B	1
C	1
E	2
M	2
P	1
R	1
Y	1
<space>	1



Ký tự	Tần số	Từ mã
A	3	10
B	1	0001
C	1	110
E	2	010
M	2	001
P	1	0110
R	1	0111
Y	1	0000
<space>	1	111



Sinh mã Huffman
Ta duyệt từ gốc xuống lá:
•Gán 0 khi đi về trái, 1 khi đi về phải.

CAPYBARA MEME

11010011000000001100111101110010100010 10

Văn bản gốc : 13*8 = 104 bit

Văn mã hóa : 40 bit

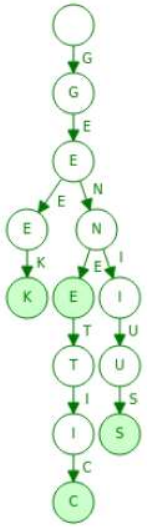
Mở rộng của cây tổng quát

Prefix-tree

- Tên gọi khác: Trie, radix tree (compressed Trie)
- Tries được giới thiệu bởi René de la Briandais trong *File searching using variable length keys*. **1959**

Thao tác với cây tổng quát

- Cây tiền tố (Prefix Tree), còn được gọi là Trie (được phát âm là "try"),
  - là một cấu trúc dữ liệu cây đặc biệt được sử dụng để lưu trữ và tra cứu các chuỗi kí tự có thể chia thành các từ hoặc tiền tố (prefix).
  - Cây tiền tố thường được sử dụng để tìm kiếm, tìm kiếm từ điển, và kiểm tra tính tồn tại của các chuỗi.



"geek", "genius", "gene" và "genetic",

63

Thao tác với cây tổng quát

- Hãy xây dựng chương trình
  - Đọc vào danh sách các từ (chỉ gồm chữ cái tiếng anh, viết thường) và xây dựng cây prefix
  - Nhập vào 1 từ, kiểm tra từ đó có trong danh sách các từ ban đầu hay không
  - Nhập vào 1 cụm prefix, in ra các từ có cùng bắt đầu bởi cụm đó trong cây vừa dựng
  - Xóa 1 từ trên cây prefix vừa dựng
  - Giải phóng bộ nhớ cấp phát cho cây

64

```
/*Minh họa các thao tác cơ bản trên prefix tree*/

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define ALPHABET_SIZE 26

// Định nghĩa cấu trúc dữ liệu nút cho cây prefix
// cây chỉ chứa các từ tiếng anh chữ thường
// Dùng mảng số 0-25 cho chữ cái từ a-z (trừ độ lệch -'a')
struct TrieNode {
    struct TrieNode* children[ALPHABET_SIZE];
    bool isEndOfWord; // đánh dấu trạng thái kết thúc của từ
};
```

```
// Hàm tạo nút mới bằng cách cấp phát động nút mới
// và khởi tạo giá trị các nút con của nút mới là NULL
struct TrieNode* createNode() {
    struct TrieNode* pNode = (struct
TrieNode*)malloc(sizeof(struct TrieNode));
    pNode->isEndOfWord = false;
    for (int i = 0; i < ALPHABET_SIZE; i++) {
        pNode->children[i] = NULL;
    }
    return pNode;
}
```

65


```
// Hàm tạo nút mới bằng cách cấp phát động nút mới
// và khởi tạo giá trị các nút con của nút mới là NULL
struct TrieNode* createNode() {
    struct TrieNode* pNode = (struct
TrieNode*)malloc(sizeof(struct TrieNode));
    pNode->isEndOfWord = false;
    for (int i = 0; i < ALPHABET_SIZE; i++) {
        pNode->children[i] = NULL;
    }
    return pNode;
}
```


```
// Hàm thêm từ vào cây prefix
void insertWord(struct TrieNode* root, const char* word) {
    struct TrieNode* currentNode = root;
    for (int i = 0; word[i] != '\0'; i++) {
        int index = word[i] - 'a';
        if (!currentNode->children[index]) {
            currentNode->children[index] = createNode();
        }
        currentNode = currentNode->children[index];
    }
    currentNode->isEndOfWord = true;
}
```

66

```
// Hàm kiểm tra từ có trong cây prefix hay không
bool searchWord(struct TrieNode* root, const char* word) {
    struct TrieNode* currentNode = root;
    for (int i = 0; word[i] != '\0'; i++) {
        int index = word[i] - 'a';
        if (!currentNode->children[index]) {
            return false;
        }
        currentNode = currentNode->children[index];
    }
    return (currentNode != NULL && currentNode->isEndOfWord);
}
```

67

```
// Hàm duyệt và in ra tất cả các từ trong cây prefix
// word là mảng chứa các ký tự ở phía trước
// level là số lượng ký tự trong mảng word tới thời điểm hiện tại
void traverseTrie(struct TrieNode* root, char word[], int level) {
    if (root->isEndOfWord) {
        word[level] = '\0';
        printf("%s\n", word);
    }

    for (int i = 0; i < ALPHABET_SIZE; i++) {
        if (root->children[i]) {
            word[level] = 'a' + i;
            traverseTrie(root->children[i], word, level + 1);
        }
    }
}
```

68

```
// Hàm kiểm tra từ có trong cây prefix hay không
void printPrefixWords(struct TrieNode* root, const char* word) {
   struct TrieNode* currentNode = root;
   // đi theo danh sách prefix
            for (int i = 0; word[i] != '\0'; i++) {
      int index = word[i] - 'a';
      if (!currentNode->children[index]) {
         return ;
      }
      currentNode = currentNode->children[index];
   }
   if(currentNode == NULL) return;

   // Duyệt và in ra tất cả các từ trong cây prefix
   char prefixWord[50];
   strcpy(prefixWord, word);
   int level = strlen(prefixWord);

   traverseTrie(currentNode,prefixWord,level);
}
```

69

```
// Hàm main để thử nghiệm cây prefix
int main() {
   struct TrieNode* root = createNode();

   // Thêm các từ vào cây prefix
   insertWord(root, "hello");
   insertWord(root, "world");
   insertWord(root, "hi");
   insertWord(root, "hey");

   // Kiểm tra từ có trong cây prefix hay không
   printf("Does 'hello' exist in the trie? %s\n", searchWord(root, "hello") ? "Yes" : "No");
   printf("Does 'world' exist in the trie? %s\n", searchWord(root, "world") ? "Yes" : "No");
   printf("Does 'hey' exist in the trie? %s\n", searchWord(root, "hey") ? "Yes" : "No");
   printf("Does 'how' exist in the trie? %s\n", searchWord(root, "how") ? "Yes" : "No");

   // in ra các từ có cùng prefix là "he" --> hello và hey
   printPrefixWords(root,"he");

   return 0;
}
```

70

Cây prefix

- Yêu cầu thêm
  - Sửa lại hàm đọc để có thể đọc vào danh sách các từ lấy từ 1 đoạn văn (chuyển hết các ký tự in hoa về thường)
  - Sửa lại cấu trúc để tìm xem các tổ hợp prefix có k ký tự nào có tần số xuất hiện lớn nhất trên đoạn văn vừa nhập
  - Thêm hàm xóa 1 từ trên cây prefix
  - Hoàn thiện hàm xóa toàn bộ cây (giải phóng bộ nhớ cấp phát động của từng phần tử)

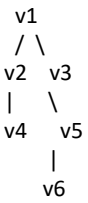
71

Quản lý phiên bản (version control)

Vì sao dùng cây tổng quát để quản lý phiên bản?

- Có thể có nhiều **nhánh phát triển** (branches).
- Một phiên bản có thể sinh ra nhiều phiên bản con.
- Các phiên bản có thể được **merge** trở lại.
- Cần lưu lại lịch sử **cha – con** giữa các phiên bản.

Quản lý phiên bản (version control)



- v1: phiên bản đầu tiên.
- v2, v3: nhánh phát triển song song từ v1.
- v5: phát triển từ v3, rồi sinh v6.

```
typedef struct VersionNode {
    char id[20];                      // mã phiên bản, ví dụ "v1", "v2"
    char message[100];                // ghi chú (commit message)
    struct VersionNode* parent;       // con trỏ đến phiên bản cha
    struct VersionNode* firstChild;   // con trỏ đến phiên bản con đầu tiên
    struct VersionNode* nextSibling; // con trỏ đến phiên bản cùng cấp tiếp theo
} VersionNode;
```



```
// Tạo 1 phiên bản mới
VersionNode* createVersion(const char* id, const char* message) {
    VersionNode* node = (VersionNode*)malloc(sizeof(VersionNode));
    strcpy(node->id, id);
    strcpy(node->message, message);
    node->parent = NULL;
    node->firstChild = NULL;
    node->nextSibling = NULL;
    return node;
}
```

```
// Thêm 1 phiên bản con vào phiên bản cha
void addChild(VersionNode* parent, VersionNode* child) {
    child->parent = parent;
    if (parent->firstChild == NULL) {
        parent->firstChild = child;
    }
    else {
        VersionNode* temp = parent->firstChild;
        while (temp->nextSibling != NULL)
            temp = temp->nextSibling;
        temp->nextSibling = child;
    }
}
```

Suffix tree

Kd-tree