

CONFIDENTIAL

C Programming Basic – week 8

Gdb – Make

Tree

Lecturer :

Do Quoc Huy

Dept of Computer Science

Hanoi University of Technology

A decorative graphic on the left side of the slide featuring three balloons in light green, light blue, and light purple, with yellow streamers and triangular flags trailing behind them.

Các chủ đề của tuần này

- Cách sử dụng chương trình gỡ lỗi (debugger) (gdb)
- Cấu trúc cây dữ liệu
 - Cây nhị phân
 - Cây nhị phân tìm kiếm
- Xử lý đệ qui trên cây



Gỡ lỗi với **gdb** (1)

- **gdb**: the Gnu DeBugger
- <http://www.cs.caltech.edu/courses/cs11/material/c/mike/misc/gdb.html>
- Mục đích:
 - Là chương trình debug (gỡ lỗi) chương trình trong Linux
 - Debug: tiến hành kiểm tra, theo dõi sự thực thi của chương trình => tìm ra lỗi



`gdb` for debugging (2)

- Trước khi sử dụng `gdb`:
 - Dịch mã nguồn C với cờ: `-g`
 - Tất cả mã nguồn sẽ được đặt vào file nhị phân thực thi được
- Thực thi: `gdb myprogram`
- Kích hoạt môi trường được thông dịch



`gdb` for debugging (3)

`gdb> run`

- Chương trình chạy...
- Nếu chương trình hoạt động tốt, nó sẽ thoát ra bình thường, trả về dấu nhắc
- Nếu như có lỗi, `gdb` sẽ thông báo và ngừng chương trình



Cách sử dụng GDB

- Tạo điểm dừng (break point): khi đến điểm này, chương trình tạm dừng lại
 - `gdb break` số_dòng hoặc
 - `gdb break` tên_hàm
- Để xóa điểm dừng
 - `gdb break` số_thứ_tự_break_point



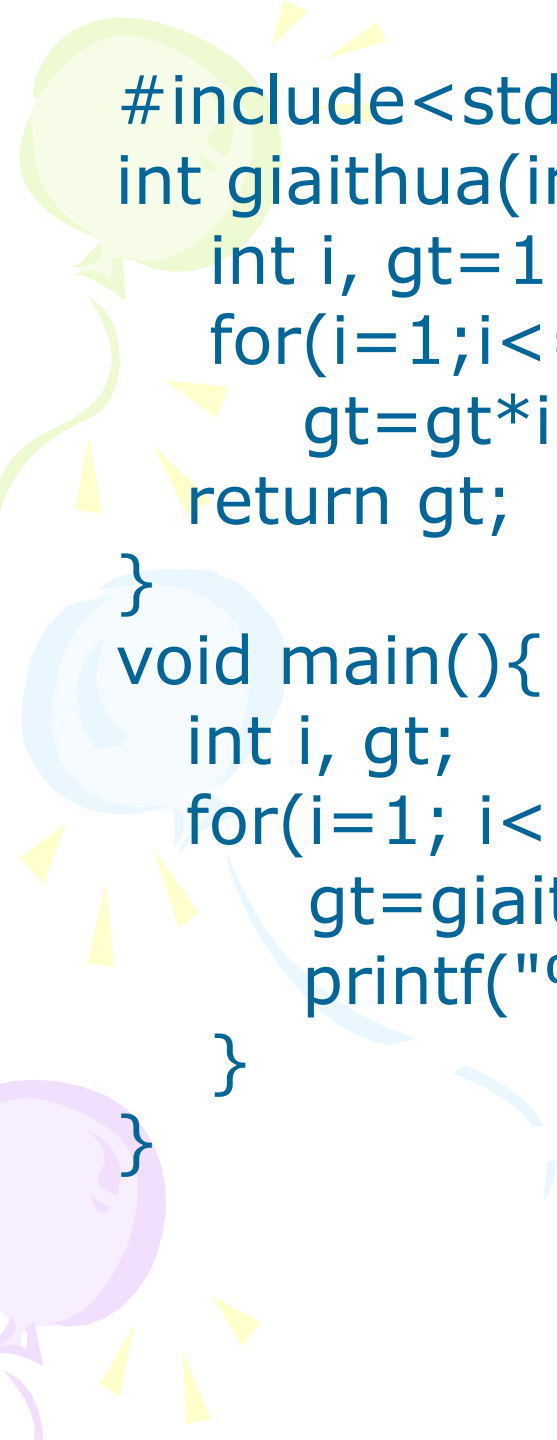
Cách sử dụng GDB

- Tại điểm dừng
 - `gdb next` [số_dòng]
 - nếu không có số_dòng thì lệnh kế tiếp được thực thi
 - ngược lại, chương trình chạy từ lệnh hiện tại tới dòng lệnh [số_dòng]
 - chạy tiếp đến điểm dừng tiếp theo hoặc tới hết chương trình: `gdb continue`
 - chạy vào trong thân hàm `gdb step`



Cách sử dụng GDB

- Chạy chương trình bằng lệnh: **gdb run**
 - chương trình không có lỗi => thực thi bình thường
 - ngược lại => thông báo lỗi. Sử dụng **gdb where** để xác định vị trí lỗi
 - Thoát: **gdb quit**
- Để xem giá trị của một biến
 - **gdb display** tên_biến (in giá trị biến mỗi lần thực hiện lệnh) hoặc
 - **gdb print** tên_biến



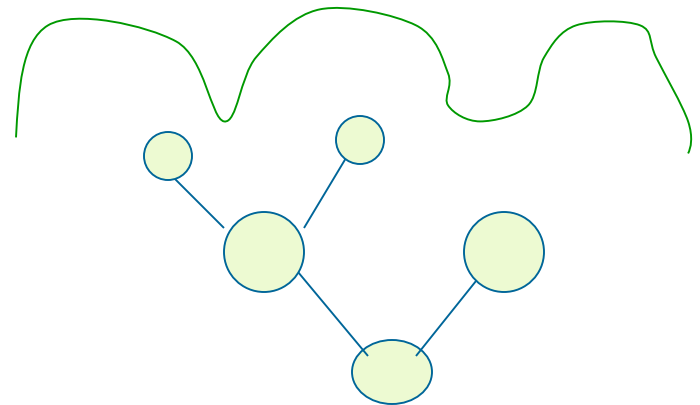
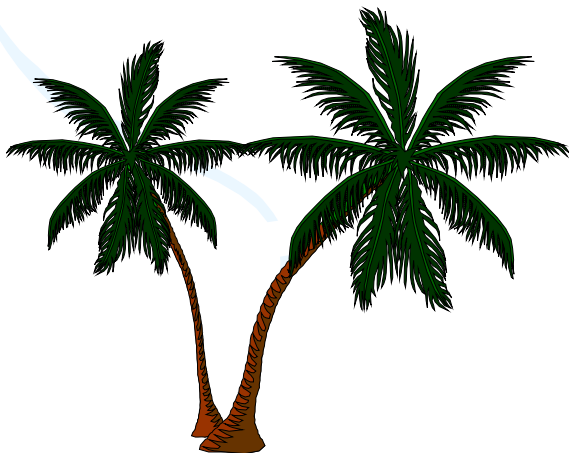
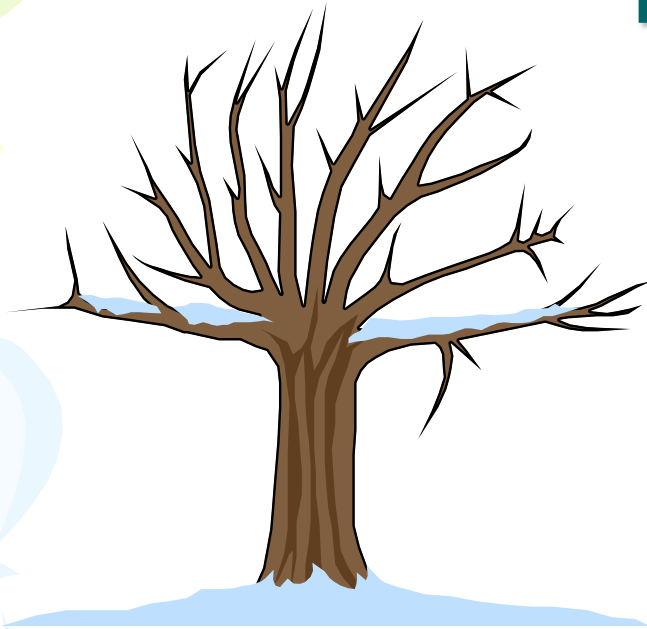
```
#include<stdio.h>
int giaithua(int n){
    int i, gt=1;
    for(i=1;i<=n;i++)
        gt=gt*i;
    return gt;
}
void main(){
    int i, gt;
    for(i=1; i<=5; i++){
        gt=giaithua(i);
        printf("%3d\n",gt);
    }
}
```



`gdb` – các từ viết tắt


- Các lệnh `gdb` phổ biến thường có từ viết tắt
 - `p` (tương tự `print`)
 - `c` (tương tự `continue`)
 - `n` (tương tự `next`)
 - `s` (tương tự `step`)
- Giúp việc tương tác với trình gỡ lỗi hiệu quả hơn

Tree

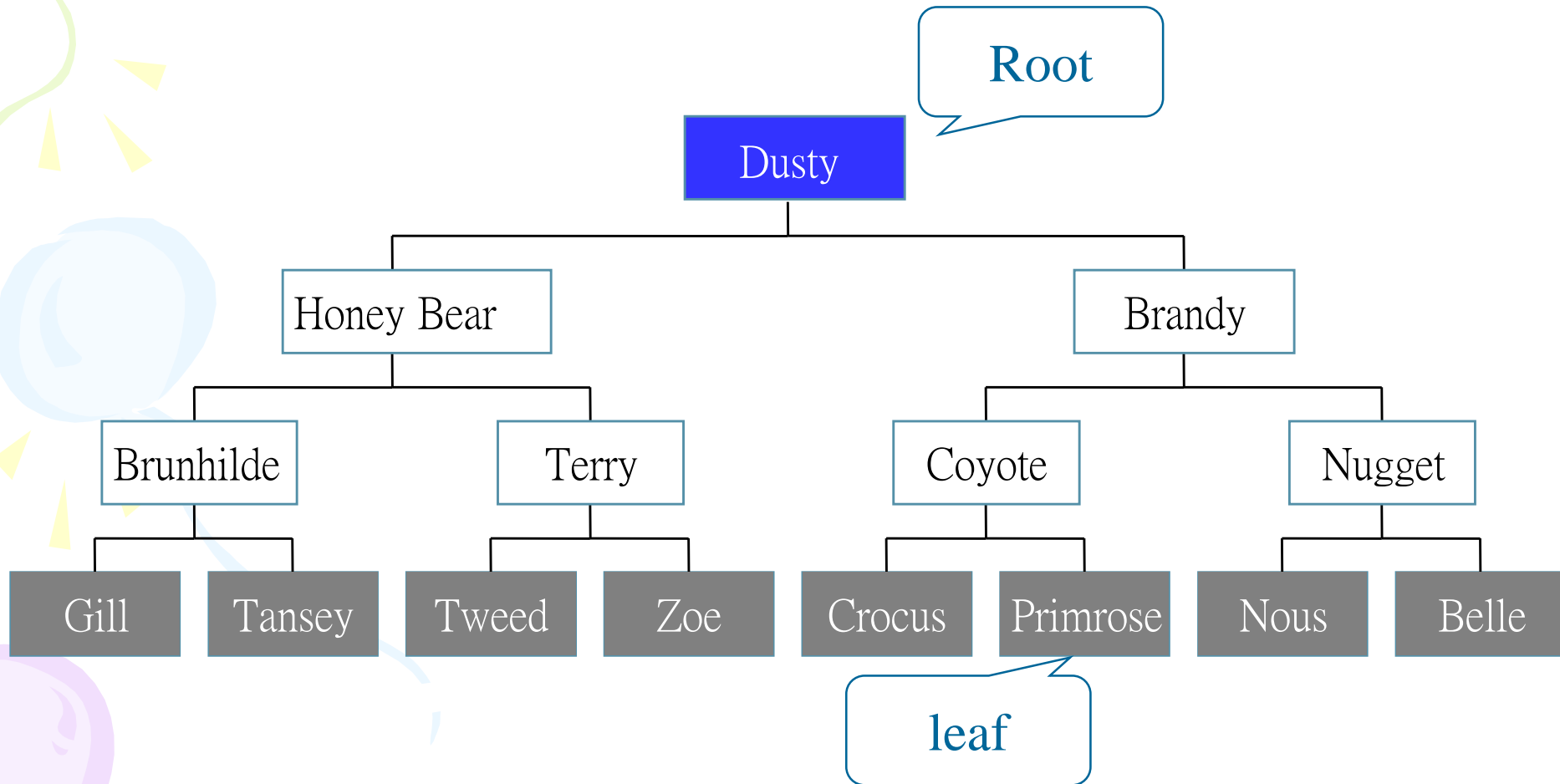




Trees, Binary Trees, and Binary Search Trees

- **Linked list**: cấu trúc tuyến tính, khó thể hiện được sự thứ bậc (hierarchy)
 - **Stack, Queue**: thể hiện được một phần thứ bậc nhưng chỉ 1 chiều
 - **Tree**: khắc phục những hạn chế trên.
 - bao gồm các nút và cạnh.
 - Ngược với cây tự nhiên: gốc ở trên và các lá ở dưới
- 

Family Tree

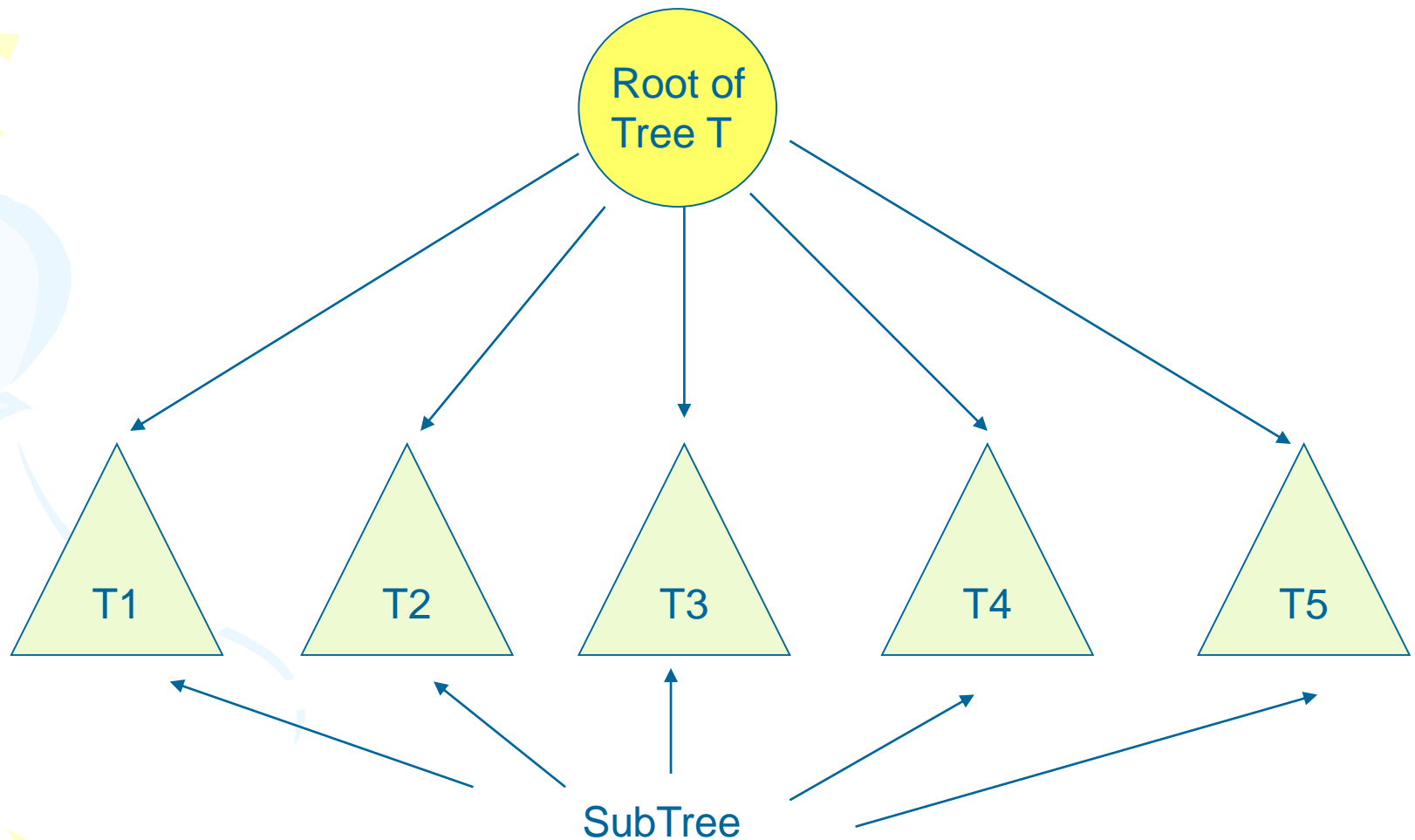




Định nghĩa cây

- Cây là tập hợp hữu hạn của một hoặc nhiều nút trong đó:
- Có 1 nút đặc biệt gọi là nút gốc: root
- Các nút còn lại được phân chi thành các cây con không giao nhau T_1, T_2, \dots, T_n

Định nghĩa đệ quy





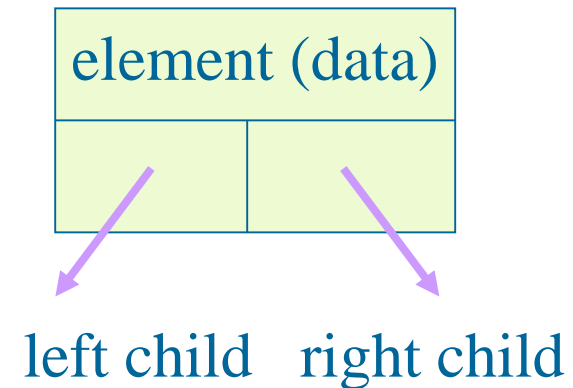
Binary Tree

- Binary tree (cây nhị phân): là một cây trong đó mỗi nút không có quá 2 nút con
=> Mỗi nút chỉ có 0, 1, hoặc 2 nút con

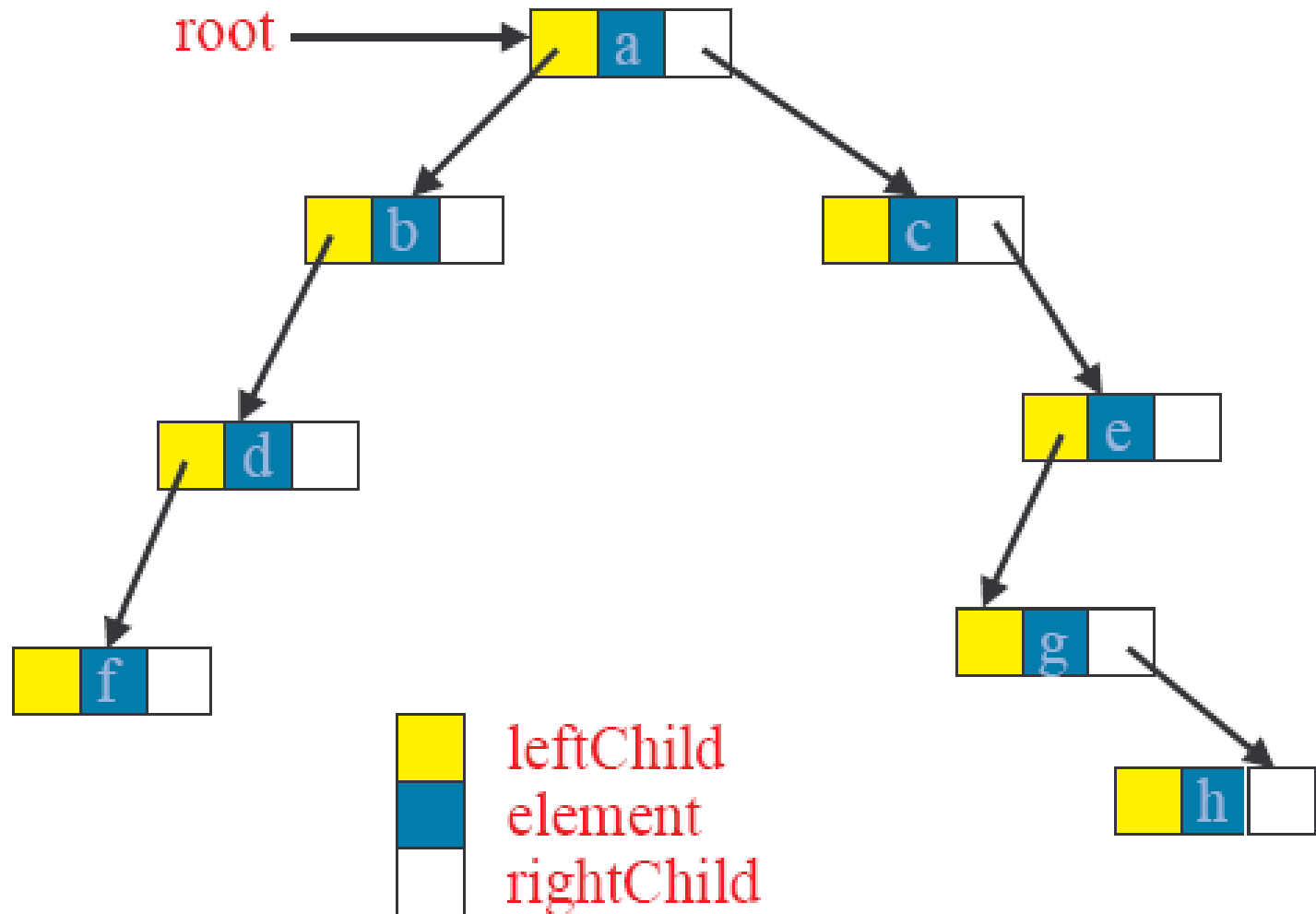
Biểu diễn liên kết

- Mỗi nút gồm có: dữ liệu, liên kết đến các con của nó (tối đa 2 con) => gồm các trường: dữ liệu, con trái và con phải
-)

```
typedef ... elmType;  
//whatever type of element  
typedef struct nodeType {  
    elmType element;  
    struct nodeType *left, *right;  
};  
typedef struct nodeType *treetype;
```



A linked binary tree



Three balloons (green, blue, and purple) with yellow streamers are positioned on the left side of the slide.

Một số hàm

- `makenullTree(treetype *t)`
- `creatnewNode()`
- `isEmpty()`



Khởi tạo và kiểm tra cây

```
typedef ... elmType;
typedef struct nodeType {
    elmType element;
    struct nodeType *left, *right;
} node_Type;

typedef struct nodeType *treetype;

void MakeNullTree(treetype *T) {
    (*T)=NULL;
}

int EmptyTree(treetype T) {
    return T==NULL;
}
```



Truy cập con trái, phải

```
treetype LeftChild(treetype n)
```

```
{  
    if (n!=NULL) return n->left;  
    else return NULL;  
}
```

```
treetype RightChild(treetype n)
```

```
{  
    if (n!=NULL) return n->right;  
    else return NULL;  
}
```

Tạo nút mới

```
node_type *create_node (elmtype NewData)
{
    N=(node_type*)malloc (sizeof (node_type) );
    if (N != NULL)
    {
        N->left = NULL;
        N->right = NULL;
        N->element = NewData;
    }
    return N;
}
```

Kiểm tra một nút có phải lá?

```
int IsLeaf(treetype n) {  
    if (n!=NULL)  
        return (LeftChild(n)==NULL) && (Right  
            Child(n)==NULL) ;  
    else return -1;  
}
```



Xử lý đệ quy: tìm số nút trên cây

- Vì cây là một cấu trúc dữ liệu đệ quy (cây gồm các cây con) => có thể áp dụng giải thuật đệ quy
- Số nút = 1 (nút gốc) + Số nút cây con trái + Số nút cây con phải

```
int nb_nodes(treetype T) {  
    if (EmptyTree(T)) return 0;  
    else return 1+nb_nodes(LeftChild(T)) +  
        nb_nodes(RightChild(T));  
}
```




Tạo 1 cây từ 2 cây con

```
treetype createfrom2 (elmttype v,  
    treetype l, treetype r) {  
    treetype N;  
    N=(node_type*)malloc(sizeof(node_type  
e));  
    N->element=v;  
    N->left=l;  
    N->right=r;  
    return N;  
}
```

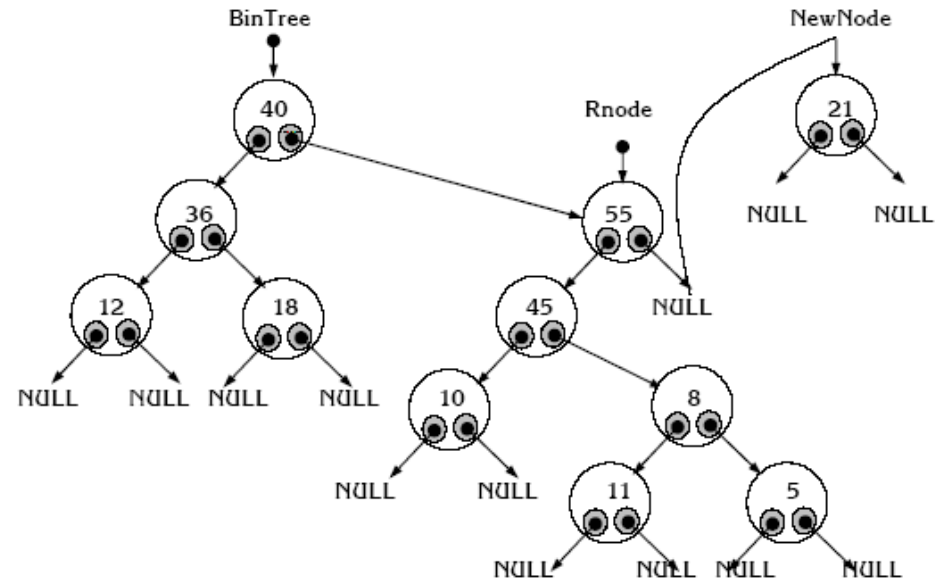
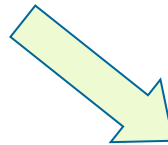
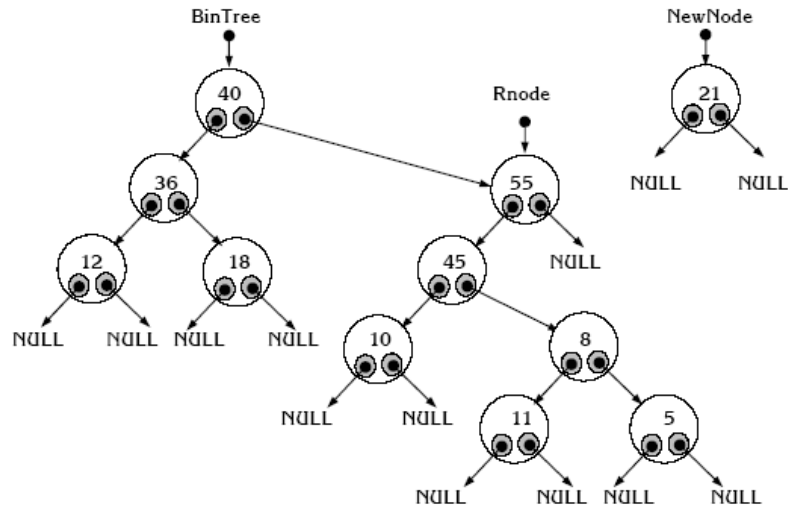
Thêm một nút vào vị trí trái nhất trên cây

```
treetype Add_Left(treetype *Tree, elmtype NewData) {
    node_type *NewNode = Create_Node(NewData);
    if (NewNode == NULL) return (NewNode);
    if (*Tree == NULL)
        *Tree = NewNode;
    else{
        node_type *Lnode = *Tree;
        while (Lnode->left != NULL)
            Lnode = Lnode->left;
        Lnode->left = NewNode;
    }
    return (NewNode);
}
```

Thêm một nút vào vị trí phải nhất trên cây

```
treetype Add_Left(treetype *Tree, elmttype NewData) {
    node_type *NewNode = Create_Node(NewData);
    if (NewNode == NULL) return (NewNode);
    if (*Tree == NULL)
        *Tree = NewNode;
    else{
        node_type *Rnode = *Tree;
        while (Rnode->right != NULL)
            Rnode = Rnode->right;
        Rnode->right = NewNode;
    }
    return (NewNode);
}
```

Illustration



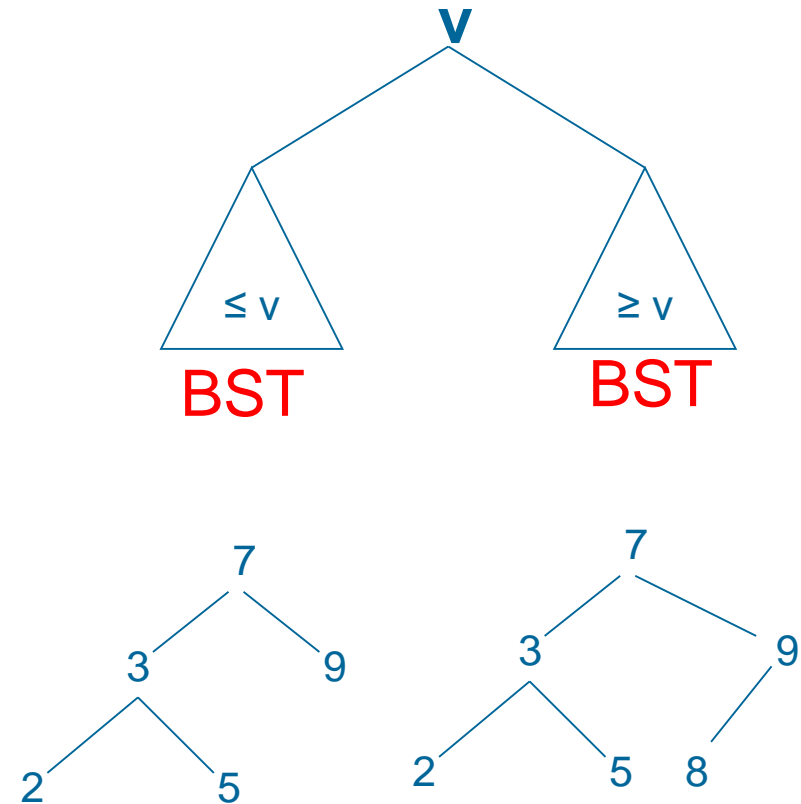


Exercise

- Viết chương trình thực hiện
 - Cài đặt một cấu trúc cây với kiểu dữ liệu `elemType` là `int`
 - Nhập từ bàn phím số nguyên n và m . Sau đó nhập lần lượt n nút trái nhất và m nút phải nhất trên cây
 - Cho biết số lượng nút
 - Cho biết số lượng nút lá
 - Cho biết độ cao của cây

Binary Search Tree

- Mỗi nút có một khóa (key) duy nhất
- Mọi key ở nút con trái (phải) thì nhỏ hơn (lớn hơn) key ở nút gốc.
- Các cây con trái, phải cũng là các cây nhị phân tìm kiếm



Cài đặt Binary Search Tree

```
#include <stdio.h>
#include <stdlib.h>
typedef . . . KeyType; // Loại dữ liệu của Key
typedef struct Node{
    KeyType key;
    struct Node* left, right;
} NodeType;
typedef Node* TreeType;
```



Tìm kiếm trên BST

```
TreeType Search(KeyType x, TreeType Root) {  
    if (Root == NULL) return NULL; // not found  
    else if (Root->key == x) /* found x */  
        return Root;  
    else if (Root->key < x)  
        //continue searching in the right sub tree  
        return Search(x, Root->right);  
    else {  
        // continue searching in the left sub tree  
        return Search(x, Root->left);  
    }  
}
```

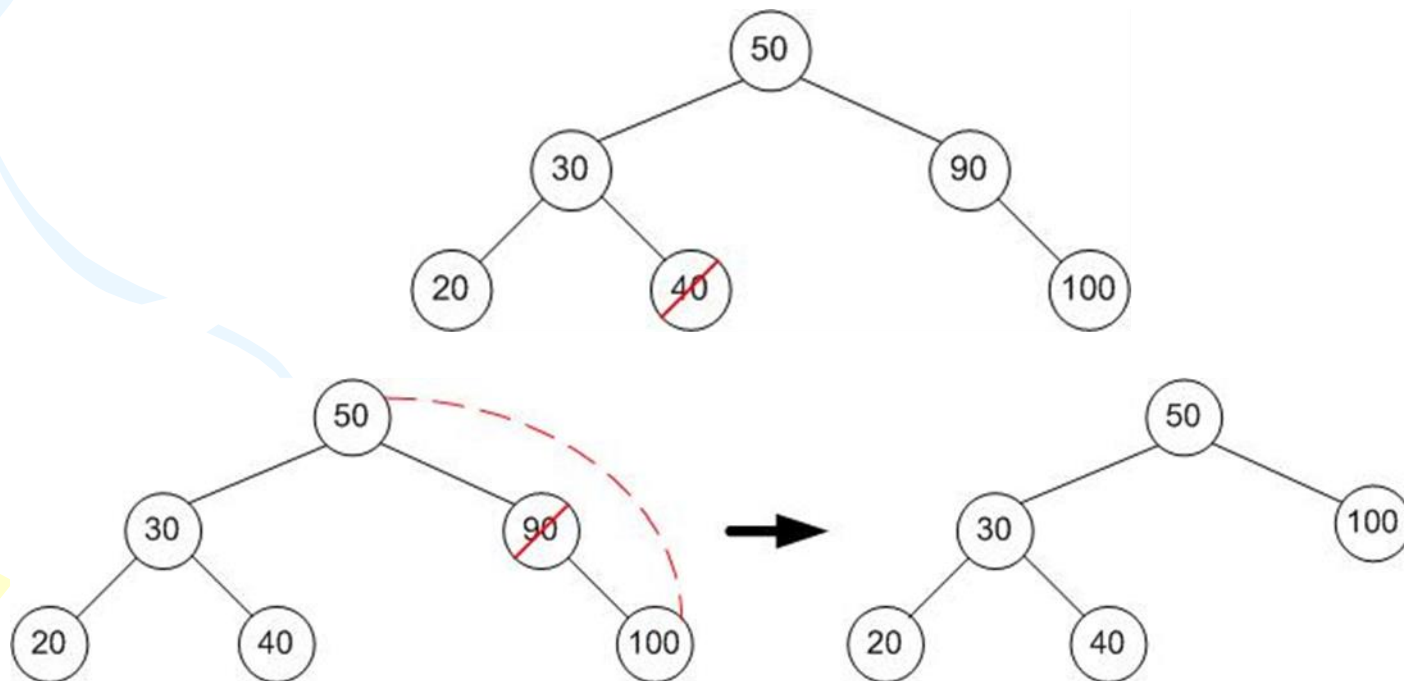

Insert a node from a BST

- Lưu ý: Trong BST, không có 2 nút nào có cùng key

```
void InsertNode(KeyType x, TreeType *Root ) {  
    if (*Root == NULL) {  
        /* Create a new node for key x */  
        *Root = (NodeType*) malloc(sizeof(NodeType));  
        (*Root) -> key = x;  
        (*Root) -> left = NULL;  
        (*Root) -> right = NULL;  
    }  
    else if (x < (*Root) -> key) InsertNode(x, &(*Root) -  
        > left);  
    else if (x > Root -> key) InsertNode(x, &(*Root) -  
        > right);  
}
```

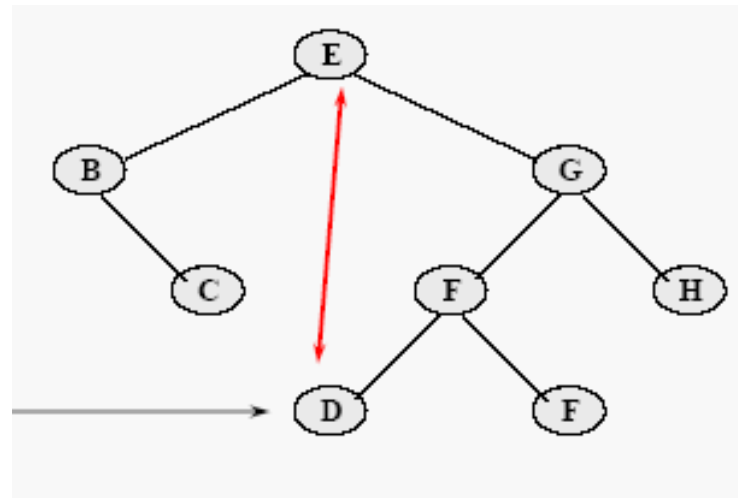
Xóa một nút khỏi BST

- Xóa một nút lá là công việc đơn giản: thiết lập con trỏ tương ứng của nút cha là NULL
- Xóa một nút trong chỉ có duy nhất 1 cây con cũng đơn giản: thiết lập con trỏ tương ứng của nút cha tới con của nút này



Xóa một nút khỏi BST

- Xóa một nút con có 2 cây con: khó khăn hơn
 - Tìm nút trái nhất của cây con phải => đổi chỗ giá trị của nút này với nút cần xóa
 - Xóa nút này (nút trái nhất của cây con phải)



Tìm nút trái nhất của cây con phải

- Tìm nút trái nhất của cây con phải và xóa

```
KeyType DeleteMin (TreeType *Root ) {  
    KeyType k;  
    if ( (*Root)->left == NULL) {  
        k= (*Root)->key;  
        (*Root) = (*Root)->right;  
        return k;  
    }  
    else return DeleteMin(&(*Root)->left);  
}
```

Xóa một nút từ BST

```
void DeleteNode(key X, TreeType *Root) {
    if (*Root != NULL)
        if (x < (*Root)->Key) DeleteNode(x, &(*Root)->left)
        else if (x > (*Root)->Key)
            DeleteNode(x, &(*Root)->right)
        else if
            ((*Root)->left == NULL) && ((*Root)->right == NULL)
            *Root = NULL;
        else if ((*Root)->left == NULL)
            *Root = (*Root)->right
        else if ((*Root)->right == NULL)
            *Root = (*Root)->left
        else (*Root)->Key = DeleteMin(&(*Root)->right);
}
```

Pretty print a BST

```
void prettyprint(TreeType tree, char *prefix) {
    char *prefixend = prefix + strlen(prefix);
    if (tree != NULL) {
        printf("%04d", tree->key);
        if (tree->left != NULL) if (tree->right == NULL) {
            printf("\304"); strcat(prefix, "    ");
        }
        else {
            printf("\302"); strcat(prefix, "\263    ");
        }
        prettyprint(tree->left, prefix);
        *prefixend = '\0';
        if (tree->right != NULL) if (tree->left != NULL) {
            printf("\n%s", prefix); printf("\300");
        } else printf("\304");
        strcat(prefix, "    ");
        prettyprint(tree->right, prefix);
    }
}
```



Exercise

- Viết hàm xóa toàn bộ nút trên cây.
Hàm này được gọi trước khi kết thúc chương trình



Solution

```
void freetree(TreeType tree)
{
    if (tree!=NULL)
    {
        freetree(tree->left);
        freetree(tree->right);
        free((void *) tree);
    }
}
```




Exercise

- Tạo cây nhị phân tìm kiếm có 10 nút.
Mỗi nút là một số nguyên được khởi tạo ngẫu nhiên
- Nhập từ bàn phím một số nguyên => tìm kiếm số nguyên này
- Gợi ý: tạo số ngẫu nhiên
`srand (time (NULL)) ;`
`rand () % MAX ;`



Exercise

- Khai báo một cấu trúc cây nhị phân để lưu trữ một danh bạ điện thoại.
- Đọc dữ liệu của 10 danh bạ từ file đầu vào, lưu vào cây nhị phân theo quy tắc sau:
 - Địa chỉ email nhỏ hơn (theo trật tự từ điển) lưu ở bên trái node.
 - Địa chỉ email nhỏ hơn (theo trật tự từ điển) lưu ở bên phải node
- (1) Xác nhận dữ liệu địa chỉ được tổ chức trong cây nhị phân (printing, debugger, .v.v.).
- (2) Tìm một địa chỉ email cụ thể trong cây và xuất nó ra file nếu tìm thấy.
- (3) Xuất dữ liệu trên cây theo chiều tăng lên của địa chỉ.
(Lưu nó lại cho tuần kế tiếp)