C Programming Basic – week 8

Nội dung

1. Sử dụng công cụ debug (gdb)
2. Cây nhị phân
3. Cây nhị phân tìm kiếm
4. Xử lý đệ quy trên cây

1. gdb

- **gdb**: the Gnu DeBugger
- http://www.cs.caltech.edu/courses/cs11/material/c/mike/misc/gdb.html
- Dùng khi có lỗi core dumps
- Hoặc chạy từng dòng lệnh

gdb (2)

- Để dùng **gdb**:
  - Biên dịch với cờ **-g**
  - Cho mã nguồn vào tệp thực thi
- Chạy dòng lệnh: **gdb myprogram**
- Thiết lập môi trường thông dịch

gdb (3)

gdb> run

- Chương trình chạy
- Nếu thực hiện thành công, chương trình thoát ra
- Nếu có lỗi core dump, **gdb** sẽ thông báo và dừng chương trình

gdb – các lệnh cơ bản

- Truy vết ngăn xếp ("**`where`**")
  - core dumps
  - Dòng lệnh cuối cùng được thực thi

gdb – các lệnh cơ bản (2)

```
gdb> where
#0 0x4006cb26 in free () from /lib/libc.so.6
#1 0x4006ca0d in free () from /lib/libc.so.6
#2 0x8048951 in board_updater (array=0x8049bd0,
   ncells=2) at 1dCA2.c:148
#3 0x80486be in main (argc=3, argv=0xbffff7b4) at
   1dCA2.c:44
#4 0x40035a52 in __libc_start_main () from
   /lib/libc.so.6
```

last call

last call in your code

stack backtrace

gdb – các lệnh cơ bản (3)

- Tìm vị trí trên cùng của ngăn xếp
- Các lỗi hay xảy ra:
  - giải phóng vùng bộ nhớ không được cấp phát
  - truy cập ra ngoài phạm vi của mảng

gdb – các lệnh cơ bản (4)

- **`break, continue, next, step`**
- **`break`** dừng chương trình ở một dòng xác định

  `gdb> break foo.c: 100` (tạo breakpoint)
- **`continue`** tái thực thi tại breakpoint
- **`next`** thực hiện dòng tiếp theo và dừng lại
- **`step`** thực thi dòng tiếp theo
  - đi vào trong hàm nếu có (**`next`** không đi vào)

gdb – các lệnh cơ bản (5)

- print và display
- print in giá trị của biểu thức

```
gdb> print i
$1 = 100
```

- display in giá trị của biểu thức mỗi khi chương trình dừng

```
gdb> display i
```

gdb – in mảng

- `print` có thể in mảng

```
int arr[] = { 1, 2, 3 };
```

```
gdb> print arr
$1 = {1, 2, 3}
```

- `$1` chỉ là tên biến chứa kết quả

```
print $1
$2 = {1, 2, 3}
```

gdb – in mảng (2)

- `print` có vấn đề với mảng cấp phát động

```
int *arr;
arr = (int *)malloc(3 * sizeof(int));
arr[0] = 1; arr[1] = 2; arr[2] = 3;
gdb> print arr
$1 = (int *) 0x8094610
```

gdb – in mảng (3)

- Có thể in sử dụng @

```
int *arr;
arr = (int *)malloc(3 * sizeof(int));
arr[0] = 1; arr[1] = 2; arr[2] = 3;

gdb> print *arr@3
$2 = {1, 2, 3}
```

gdb – Từ viết tắt

- Các hàm **gdb** phổ biến có từ viết tắt

p (**print**)

c (**continue**)

n (**next**)

s (**step**)

- Tiện lợi hơn khi debug

Các thao tác khác

- clear : xóa các breakpoint
- delete [break position]: xóa breakpoint tại vị trí cụ thể
- Dừng có điều kiện

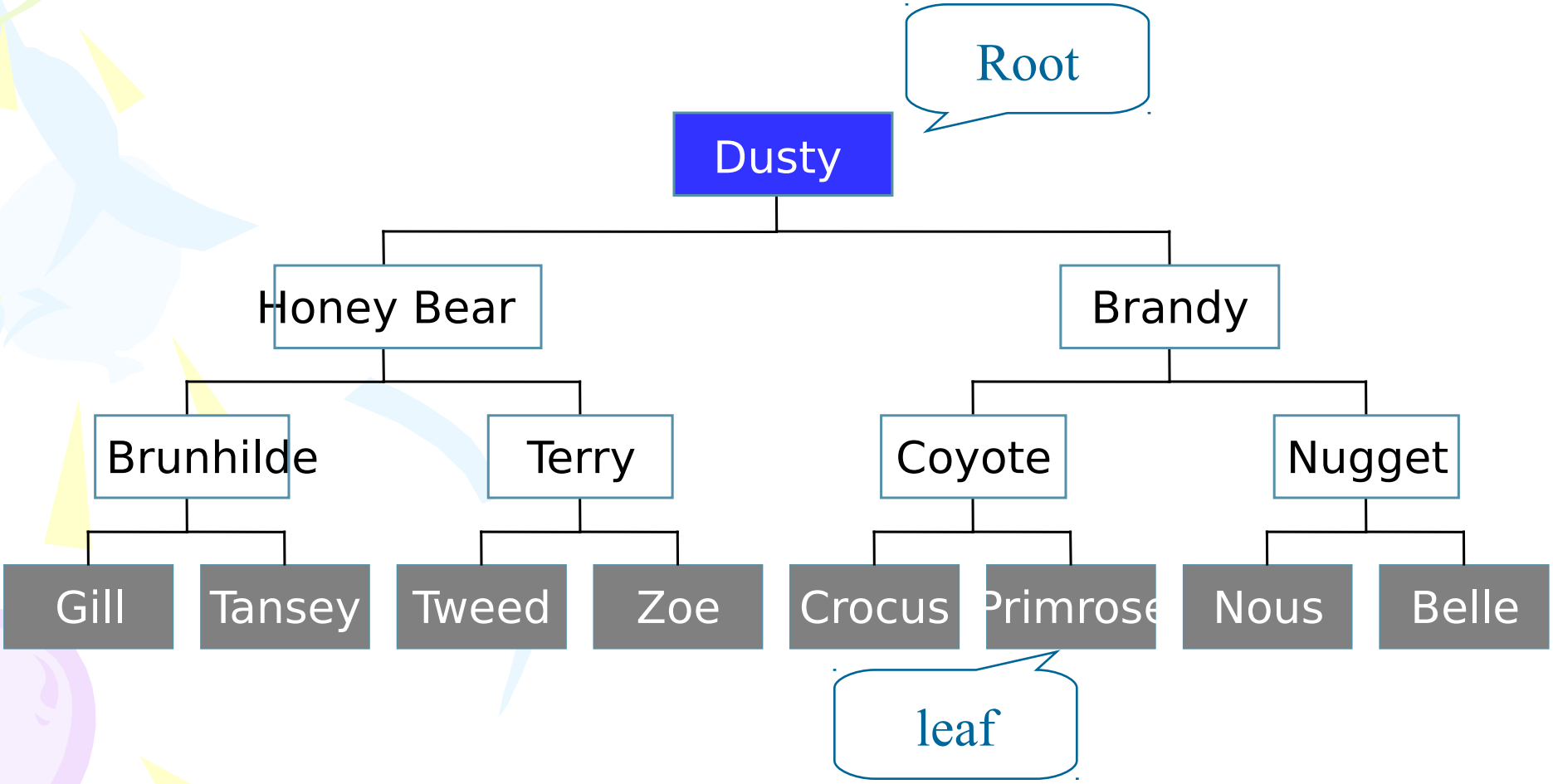
```
gdb> break foo.c: 100 if i==-1
```

- quit
- run: tái khởi động từ đầu chương trình

2. Cây nhị phân

- Danh sách liên kết là cấu trúc tuyến tính; khó có thể sử dụng để biểu diễn cấu trúc dạng phân cấp
- Ngăn xếp và hàng đợi có thể thể hiện thứ tự nhưng chỉ giới hạn trong một chiều
- Cây chứa các nút và các cạnh, có gốc ở trên cùng và các lá ở dưới cùng (không giống cây tự nhiên)

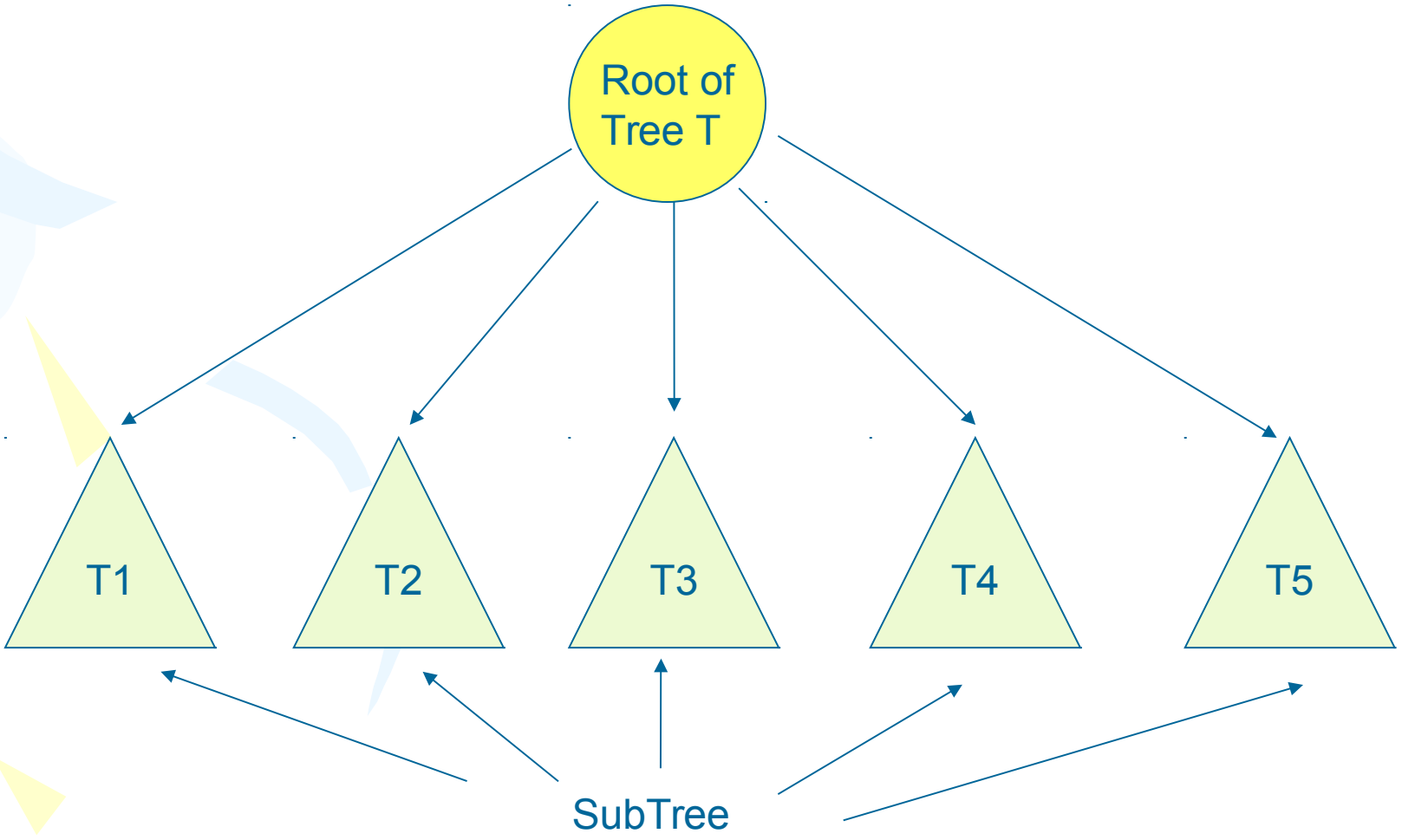
Family Tree


Root
Dusty
Honey Bear
Brandy
Brunhilde
Terry
Coyote
Nugget
Gill
Tansey
Tweed
Zoe
Crocus
Primrose
Nous
Belle
leaf

Định nghĩa cây

- Cây là một tập hữu hạn của một hay nhiều nút:
- Một nút đặc biệt là nút gốc (root)
- Các nút còn lại được chia thành $n>=0$ tập không giao nhau T_1, … , T_n, sao cho mỗi tập là một cây (con)
- Chúng ta gọi T_1, ..., T_n là cây con của root

Định nghĩa đệ quy



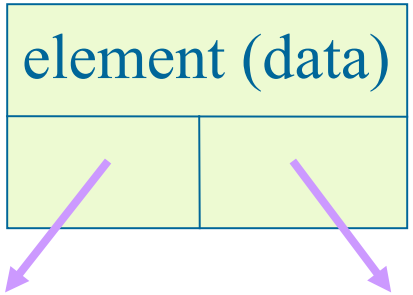
Cây nhị phân

- Cây nhị phân là cây mà mỗi nút có không quá 2 con
- Mỗi nút có thể có 0, 1, hoặc 2 con

Biểu diễn liên kết

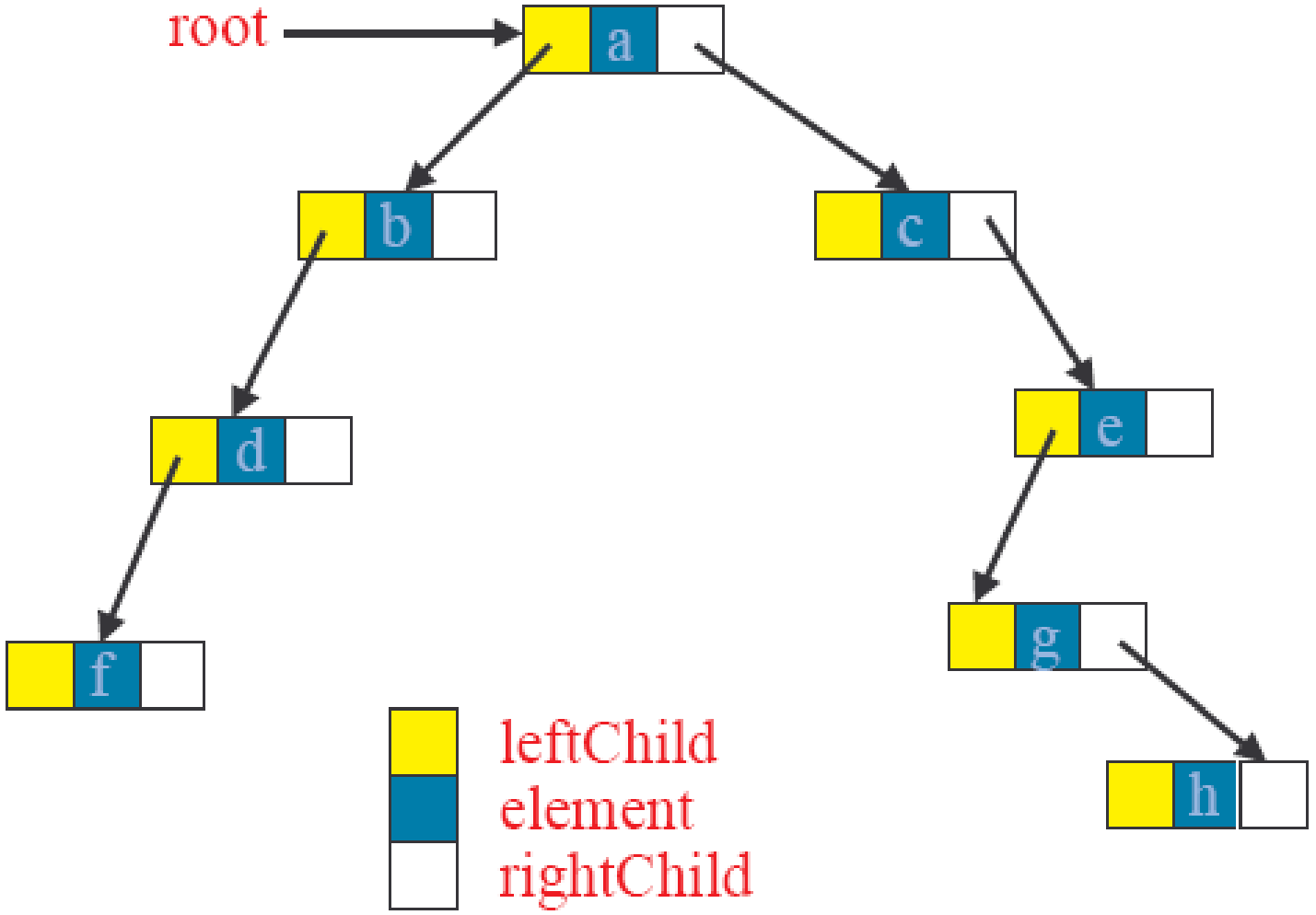
- Mỗi nút được biểu diễn bởi một cấu trúc
- Bộ nhớ cho một cây nhị phân có n noode là n * (bộ nhớ cho một nút)

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



left child right child

VD



Khai báo BST

- makeNullTree(treeType *t)
- creatNewNode()
- isEmpty()

Khởi tạo/kiểm tra

```
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

```
nodeType *createNode(elmType NewData)
{
   nodeType *N;
   N=(nodeType*)malloc(sizeof(nodeType));
   if (N != NULL)
   {
      N->left = NULL;
      N->right = NULL;
      N->element = NewData;
   }
   return N;
}
```

Kiểm tra nút lá

```
int isLeaf(treeType n){
  if(n!=NULL)
    return(leftChild(n)==NULL)&&
          (rightChild(n)==NULL);
   else return -1;
}
```

Tính đệ quy số nút

- Cây là cấu trúc đệ quy, do đó các thuật toán đệ quy phù hợp để xử lý cây

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

Tạo cây từ hai cây con

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

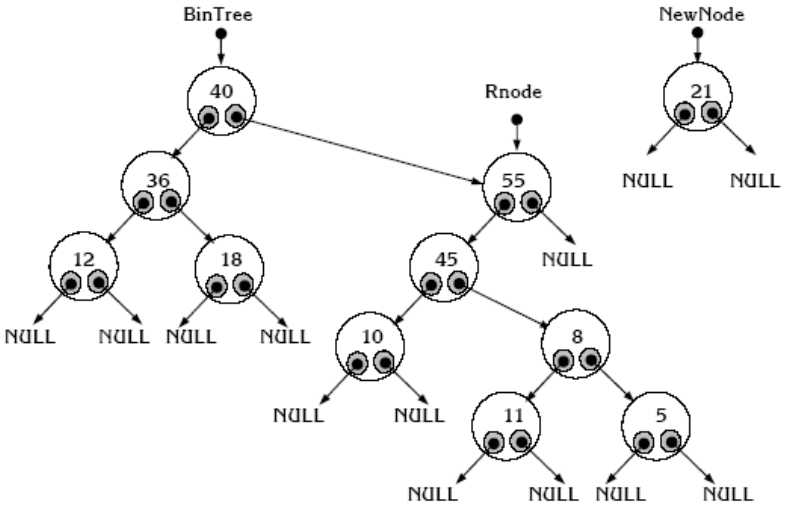
Thêm nút vào bên trái nhất

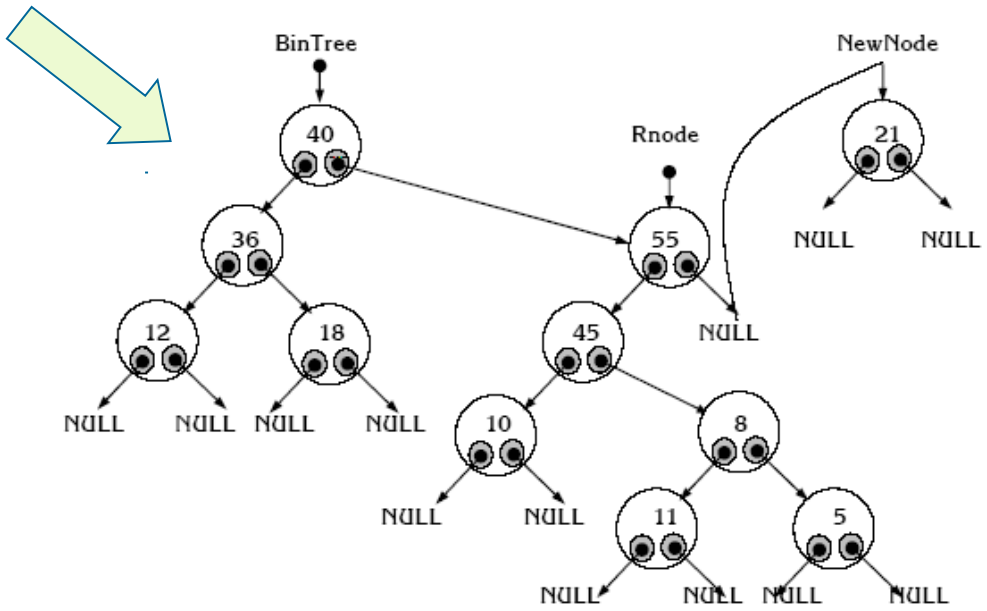
```
treetype Add_Left(treetype *Tree, elmtype NewData)
   { node_type *NewNode = Create_Node(NewData);
   if (NewNode == NULL) return (NewNode);
   if (*Tree == NULL)
      *Tree = NewNode;
   else{
      node_type *Lnode = *Tree;
      while (Lnode->left != NULL)
         Lnode = Lnode->left;
     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

Thêm nút vào bên phải nhất

```
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 *Rnode = *Tree;
      while (Rnode->right != NULL)
         Rnode = Rnode->right;
      Rnode->right = NewNode;
   }
   return (NewNode);
}
```

Minh họa


BinTree
NewNode
Rnode
40
21
36
55
12
18
45
NULL
10
8
11
5


BinTree
NewNode
Rnode
40
21
36
55
12
18
45
NULL
10
8
11
5

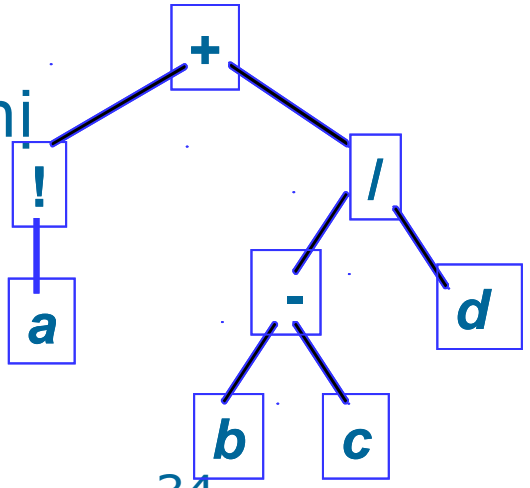
Homework 1

- Xây dựng các hàm sau cho một cây nhị phân
  - Tính chiều cao của cây
  - Đếm số nút lá
  - Đếm số nút trong
  - Đếm số con phải

Exercise 8.1

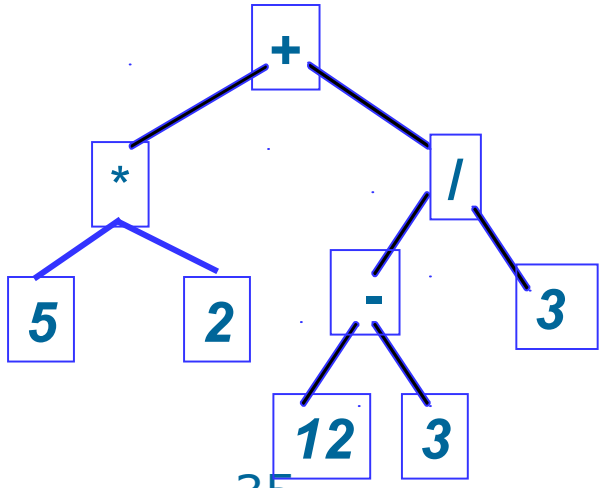
- Cây nhị phân có thể biểu diễn một biểu thức toán học: Lá là toán hạng và nút trong là toán tử
- Cây con trái và phải của một nút biểu diễn các biểu thức con cần phải tính trước khi áp dụng toán tử ở nút đó
- VD:

  !a + (b – c)/d
- Viết chương trình sử dụng cây nhị phân biểu diễn biểu thức sau:



Homework 2

- Viết chương trình nhận vào một biểu thức toán học và:
  - Lưu trữ và biểu diễn trong cây nhị phân
  - Tính giá trị biểu thức

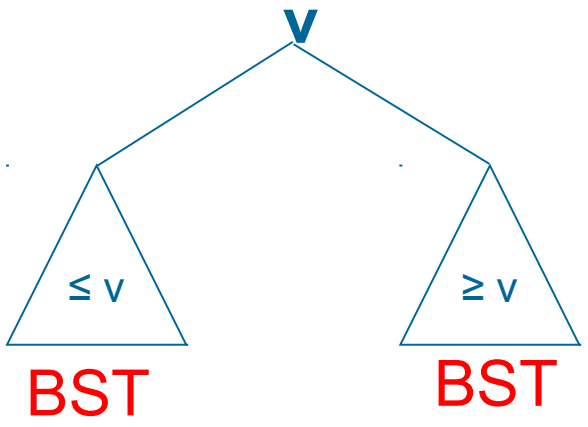


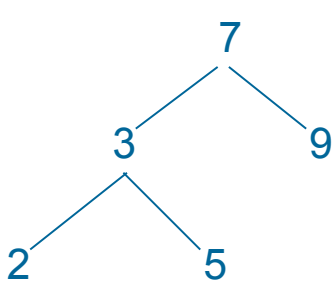
Homework 3

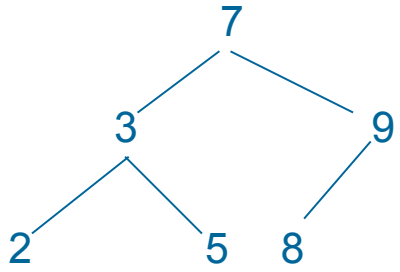
- Tạo tệp USopen.txt với nội dung như sau:
  - Dòng 1 chứa 16 tay vợt
- Xây dựng cây nhị phân biểu diễn kết quả thi đấu cho tới trận chung kết. Đầu tiên, 16 tay vợt là các nút lá
- In kết quả ra tệp treegame.txt

3. Binary Search Tree

- Mỗi nút có một khóa duy nhất
- Khóa trên cây con trái (phải) nhỏ hơn (lớn hơn) khóa ở gốc
- Các cây con trái/phải cũng là các BST







Cài đặt BST

```
#include <stdio.h>
#include <stdlib.h>
typedef . . . KeyType; // specify a type
  for the data
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);
  }
}
```

Thêm nút vào BST

- Nút có khóa là duy nhất trên cây

```
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);
}
```

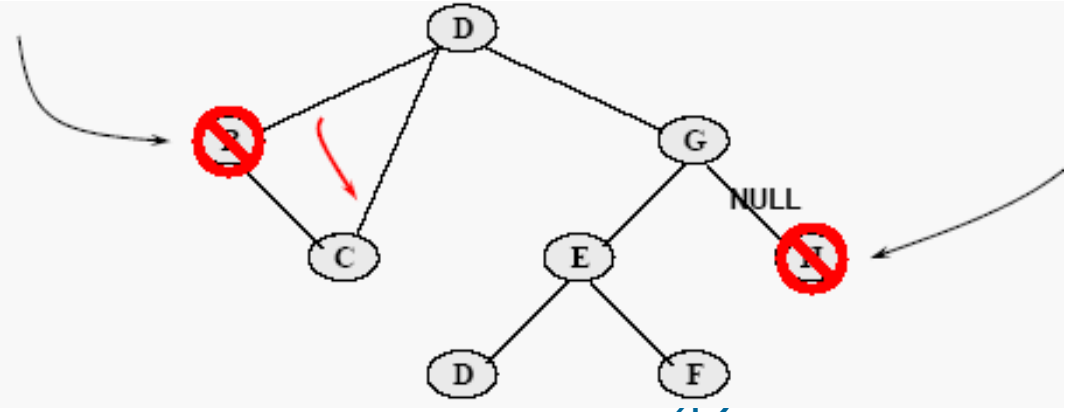
Thêm nút vào BST (2)

- Phiên bản với kiểu trả về

```
TreeType 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;
    Return Root;
}
else if (x < Root->key) return InsertNode(x, Root->left);
else if (x> Root->key) return InsertNode(x, Root->right);
}
```

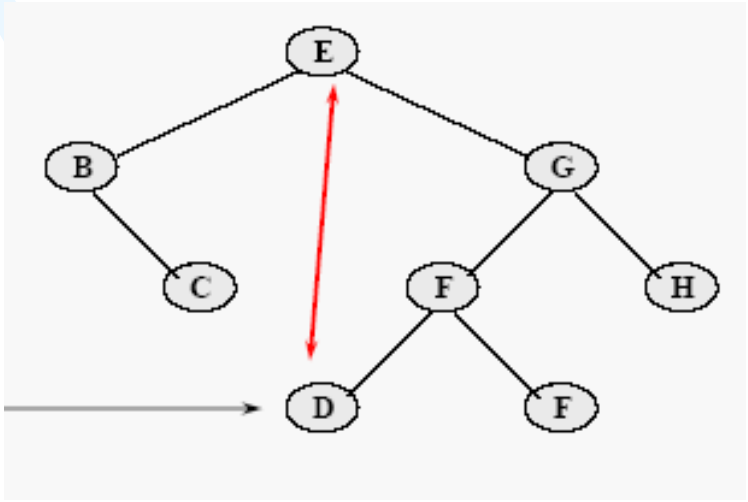
Xóa nút trên BST

- Xóa nút lá: thiết lập con trỏ tương ứng tới NULL
- Xóa nút trong có một cây con: thiết lập con trỏ tương ứng của nút cha tới root của cây con



Xóa nút trên BST (2)

- Xóa nút trong có 2 cây con:
  - Tìm nút trái nhất của cây con phải, đổi giá trị của nút này và nút cần xóa
  - Xóa nút trái nhất của cây con phải khỏi cây



Thuật toán

- Tìm nút trái nhất của cây con phải

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

Thuật toán (2)

```
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);
}
```

In BST theo định dạng

```
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 8.2

- Viết hàm xóa cây. Yêu cầu xóa tất cả các nút. Cần gọi trước khi kết thúc chương trình

Exercise 8.3

- Tạo BST với 10 nút. Mỗi nút chứa một số ngẫu nhiên
- Yêu cầu người dùng nhập một số và tìm kiếm số đó trong cây
- In ra nội dung của cây tới vị trí tìm kiếm

Exercise 8.4

- Viết các hàm FindMin và FindMax cho thư viện BST
- Tham số: con trỏ root
- Trả về: con trỏ tới nút min/max

Exercise 8.5

- Xây dựng danh bạ điện thoại
- Khai báo cấu trúc có chứa các trường "name", "telephone number", "e-mail address.”.
- Đọc 10 bản ghi từ tệp vào BST theo thứ tự tăng dần của email
- (1) Tìm một địa chỉ email
- (2) In ra thông tin các bản ghi từ BST theo thứ tự tăng dần của email

Homework 4

- Viết chương trình tra từ điển kĩ thuật Anh-Việt. Tiếng Việt được viết không dấu

- Hướng dẫn: Mỗi nút chứa từ tiếng Anh và từ tiếng Việt tương ứng
- Sử dụng BST để lưu trữ
- Các hàm cơ bản: tìm kiếm từ, thêm từ, xóa từ, và ghi ra tệp
- Nâng cao: Mỗi từ có danh sách các từ đồng nghĩa

Homework 5

- Kiểm tra tốc độ tìm kiếm trên BST.
- Sinh ra 1 triệu số ngẫu nhiên và đưa vào BST
- In ra chiều cao của cây
- Chương trình cho phép:
  - Tạo cây mới (giải phóng bộ nhớ của cây cũ).
  - Tìm kiếm và in ra số phép so sánh

Homework 6

- Tiếp tục bài tập NokiaDB.
- Tìm kiếm model sử dụng BST.
- Các chức năng: Import, Insert, Delete, Update, Search, Print