



CONFIDENTIAL

# C Programming Basic – week 8

*Gdb – Make*

*Tree*

**Lecturers :**

**Cao Tuan Dung**

**Le Duc Trung**

**Dept of Software Engineering  
Hanoi University of Technology**

1



## Topics of this week

- How to use debugger tool(gdb)
- Tree data structure
  - Binary Tree
  - Binary Search Tree
- Recursive processing on Tree

2



## **gdb** for debugging (1)

- **gdb**: the Gnu DeBugger
- <http://www.cs.caltech.edu/courses/cs11/material/c/mike/misc/gdb.html>
- Use when program core dumps
- or when want to walk through execution of program line-by-line

3



## **gdb** for debugging (2)

- Before using **gdb**:
  - Must compile C code with additional flag:  
**-g**
  - This puts all the source code into the binary executable
- Then can execute as: **gdb myprogram**
- Brings up an interpreted environment


4



## **gdb** for debugging (3)



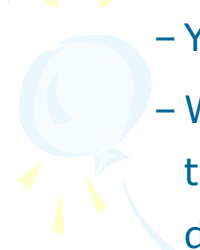

**gdb**> run

- Program runs...
  - If all is well, program exits successfully, returning you to prompt
  - If there is (e.g.) a core dump, **gdb** will tell you and abort the program
- 

5



## **gdb** – basic commands (1)

- Stack backtrace ("**where**")
    - Your program core dumps
    - Where was the last line in the program that was executed before the core dump?
    - That's what the **where** command tells you
- 
- 

6

## gdb – basic commands (2)

`gdb> where`      last call      last call in your code

```
#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
```

stack backtrace

7

## gdb – basic commands (3)

- Look for topmost location in stack backtrace that corresponds to your code
- Watch out for
  - freeing memory you didn't allocate
  - accessing arrays beyond their maximum elements
  - dereferencing pointers that don't point to part of a `malloc()`ed block

8



## **gdb – basic commands (4)**

- **break**, **continue**, **next**, **step** commands
- **break** causes execution to stop on a given line  

```
gdb> break foo.c: 100
```

 (setting a breakpoint)
- **continue** resumes execution from that point
- **next** executes the next line, then stops
- **step** executes the next statement
  - goes into functions if necessary (**next** doesn't)

9



## **gdb – basic commands (5)**

- **print** and **display** commands
- **print** prints the value of any program expression  

```
gdb> print i
```

```
$1 = 100
```
- **display** prints a particular value every time execution stops  

```
gdb> display i
```


10



## gdb – printing arrays (1)

- `print` will print arrays as well

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



```
gdb> print arr
```

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

- N.B. the `$1` is just a name for the result



```
print $1
```

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

11



## gdb – printing arrays (2)

- `print` has problems with dynamically-allocated arrays



```
int *arr;
```

```
arr = (int *)malloc(3 * sizeof(int));
```

```
arr[0] = 1; arr[1] = 2; arr[2] = 3;
```

```
gdb> print arr
```



```
$1 = (int *) 0x8094610
```

- Not very useful...

12




## gdb – printing arrays (3)

- Can print this array by using `@` (gdb special syntax)

```
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}
```



13



## gdb – abbreviations


- Common gdb commands have abbreviations

`p` (same as `print`)

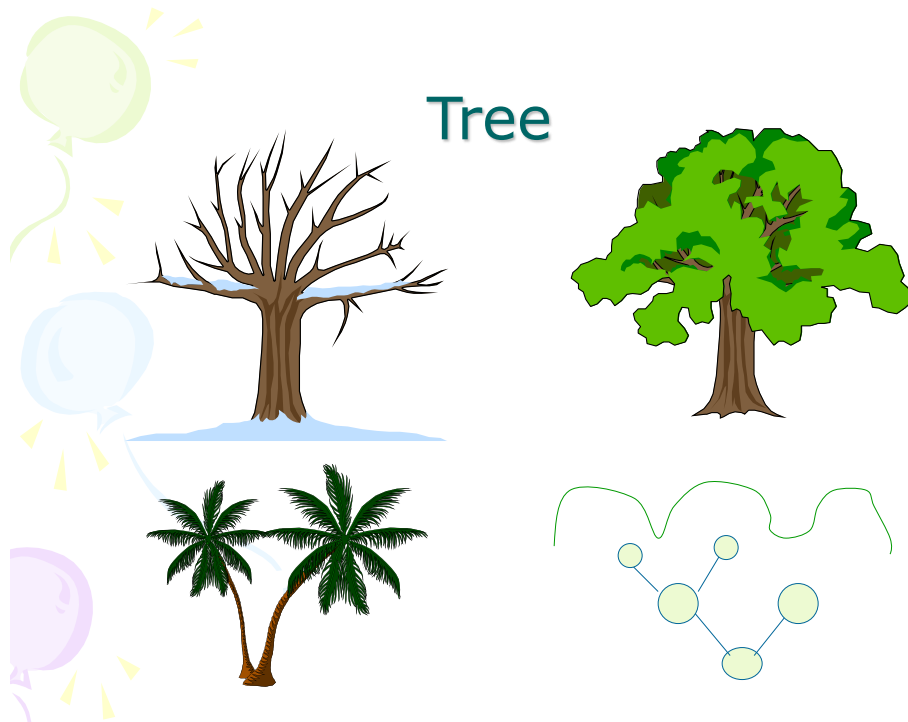
`c` (same as `continue`)

`n` (same as `next`)

`s` (same as `step`)

- More convenient to use when interactively debugging
- 

14



15

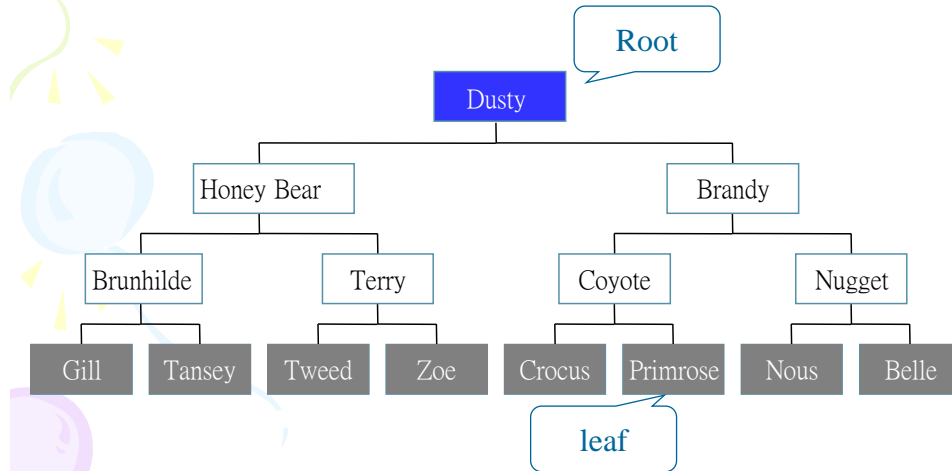
## Trees, Binary Trees, and Binary Search Trees

- Linked lists are **linear structures** and it is difficult to use them to organize a **hierarchical** representation of objects.
- Although stacks and queues reflect some hierarchy, they are limited to only **one dimension**.
- To overcome this limitation, we create a new data type called a **tree** that consists of **nodes** and **arcs**. Unlike natural trees, these trees are **depicted upside down** with the **root** at the top and the **leaves** at the bottom.

16



## Family Tree



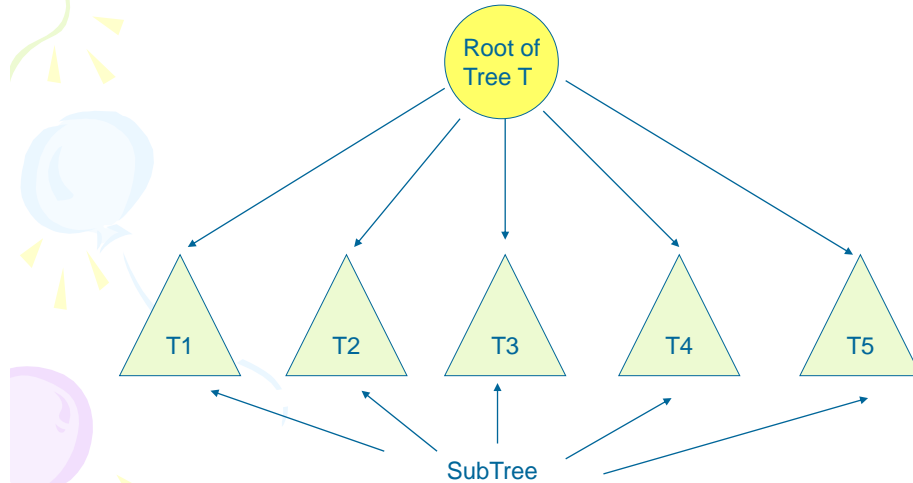
17

## Definition of tree

- A tree is a finite set of one or more nodes such that:
  - There is a specially designated node called the root.
  - The remaining nodes are partitioned into  $n \geq 0$  disjoint sets  $T_1, \dots, T_n$ , where each of these sets is a tree.
  - We call  $T_1, \dots, T_n$  the subtrees of the root.

18

## Recursive definition



19

## Binary Tree

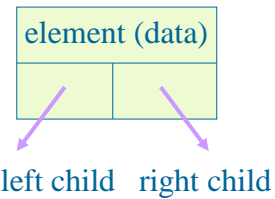
- A binary tree is a tree in which **no** node can have **more than** two children.
- Each node has 0, 1, or 2 children

20

## Linked Representation

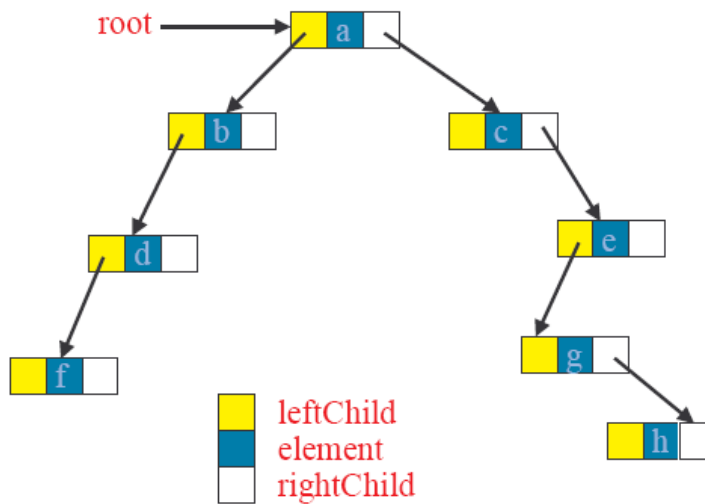
- Each tree node is represented as an object whose data type is
- The space required by an  $n$  node binary tree is  $n *$  (space required by one node)

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

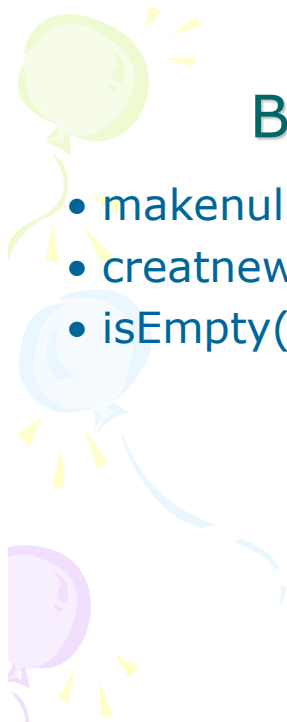


21

## A linked binary tree



22



## Binary Tree ADT

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

23



## Tree initialization and verification

```
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;
}
```

24

## Access left and right child

```

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;
}

```

25

## create a new node

```

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;
}

```

26

## check if a node is a leaf

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

27

## Recursive processing: Number of nodes

- As tree is a recursive data structure, recursive algorithms are useful when they are applied on tree.

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

28

## Create a tree from two sub-trees

```

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

```

29

## Adding a new node to the left most position

```

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 *Lnode = *Tree;
        while (Lnode->left != NULL)
            Lnode = Lnode->left;
        Lnode->left = NewNode;
    }
    return (NewNode);
}

```

30

## Adding a new node to the right most position

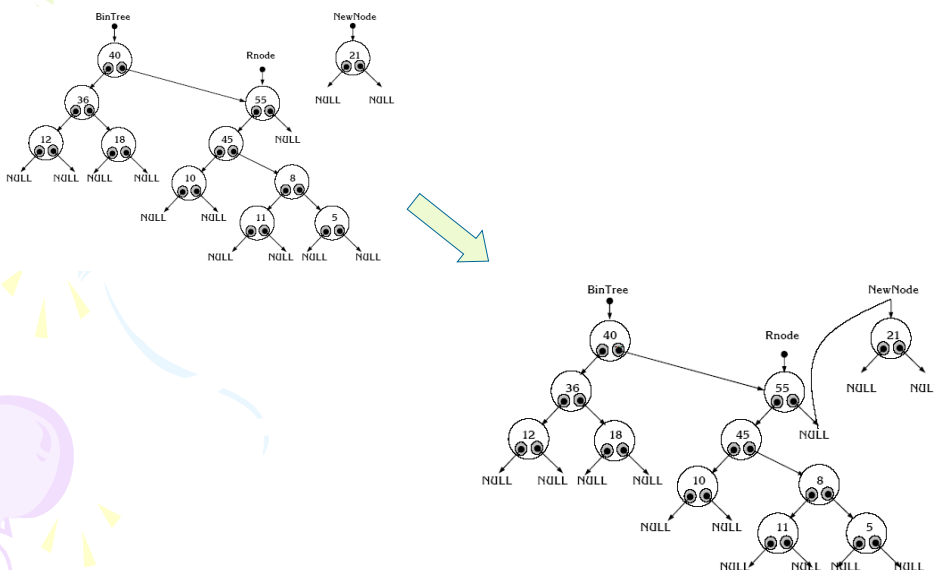
```

treetype Add_Right(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);
}

```

31

## Illustration



32



## Exercise

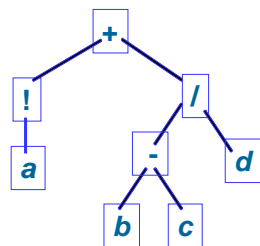
- Develop the following helper functions for a tree:
  - return the height of a binary tree.
  - return the number of leafs
  - return the number of internal nodes
  - count the number of right children.

33

## Exercise

- A binary tree can represent an arithmetic expression: The leaves are operands and the other nodes are operators.
- The left and right subtrees of an operator node represent **subexpressions** that must be evaluated **before** applying the operator at the root of the subtree.
- For example  

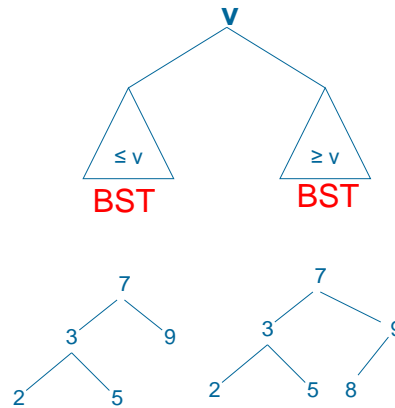
$$!a + (b - c)/d$$
- Write a program create a tree representing **this expression**



34

## Binary Search Tree

- Every element has a unique key.
- The keys in a nonempty left subtree (right subtree) are smaller (larger) than the key in the root of subtree.
- The left and right subtrees are also binary search trees.



35

## Binary Search Tree Implementation

```

4  typedef int key_t; // specify a type for the data
5
6  typedef struct node_s {
7      key_t key;
8      struct node_s *left;
9      struct node_s *right;
10 } node_t;
11
12 typedef node_t *tree_t;

```

36



## Search on BST

```

9  tree_t search(key_t x, tree_t root) {
10     if (root == NULL)
11         return NULL;           // not found
12     else if (root->key == x) /* found x */
13         return root;
14     else if (root->key < x)
15         // continue searching in the right sub tree
16         return search(x, root->right);
17     else {
18         // continue searching in the left sub tree
19         return search(x, root->left);
20     }
21 }

```

37



## Insert a node from a BST

- In a binary, there are not two nodes with the same key.

```

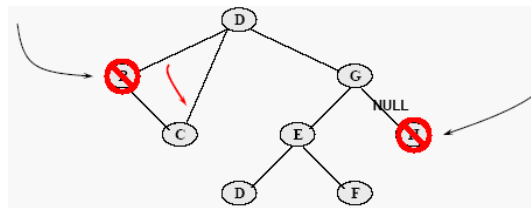
23 void insertNode(key_t x, tree_t *pRoot) {
24     if (*pRoot == NULL) {
25         /* Create a new node for key x */
26         *pRoot = (node_t *)malloc(sizeof(node_t));
27         (*pRoot)->key = x;
28         (*pRoot)->left = NULL;
29         (*pRoot)->right = NULL;
30     } else if (x < (*pRoot)->key)
31         insertNode(x, &((*pRoot)->left));
32     else if (x > (*pRoot)->key)
33         insertNode(x, &((*pRoot)->right));
34 }

```

38

## Delete a node from a BST

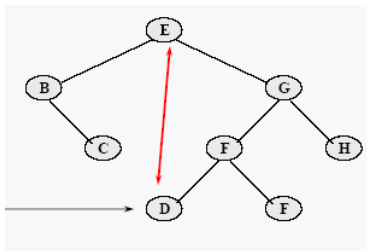
- Removing a leaf node is trivial, just set the relevant child pointer in the parent node to NULL.
- Removing an internal node which has only one subtree is also trivial, just set the relevant child pointer in the parent node to target the root of the subtree.



39

## Delete a node from a BST

- Removing an internal node which has two subtrees is more complex
  - Find the left-most node of the right subtree, and then swap data values between it and the targeted node.
  - Delete the swapped value from the right subtree.



40

## Find the left-most node of right sub tree

- This function find the leftmost node then delete it.

```

37 key_t deleteMin(tree_t *root) {
38     if ((*root)->left == NULL) {
39         key_t k = (*root)->key;
40         (*root) = (*root)->right;
41         return k;
42     } else
43         return deleteMin(&(*root)->left);
44 }

```

41

## Delete a node from a BST

```

46 void deleteNode(key_t x, tree_t *root) {
47     if (*root != NULL)
48         if (x < (*root)->key)
49             deleteNode(x, &(*root)->left);
50         else if (x > (*root)->key)
51             deleteNode(x, &(*root)->right);
52         else if ((*root)->left == NULL && ((*root)->right == NULL))
53             *root = NULL;
54         else if ((*root)->left == NULL)
55             *root = (*root)->right;
56         else if ((*root)->right == NULL)
57             *root = (*root)->left;
58         else
59             (*root)->key = deleteMin(&(*root)->right);
60 }

```

42

```

64 void prettyPrint(tree_t tree) {
65     static char prefix[200] = " ";
66     char *prefixend = prefix + strlen(prefix);
67     if (tree != NULL) {
68         printf("%04d", tree->key);
69         if (tree->left != NULL)
70             if (tree->right == NULL) {
71                 printf("\304");
72                 strcat(prefix, " ");
73             } else {
74                 printf("\302");
75                 strcat(prefix, "\263 ");
76             }
77         prettyPrint(tree->left);
78         *prefixend = '\0';
79         if (tree->right != NULL)
80             if (tree->left != NULL) {
81                 printf("\n%s", prefix);
82                 printf("\300");
83             } else
84                 printf("\304");
85         strcat(prefix, " ");
86         prettyPrint(tree->right);
87     }
88 }

```

Pretty print a BST

43

## Exercise

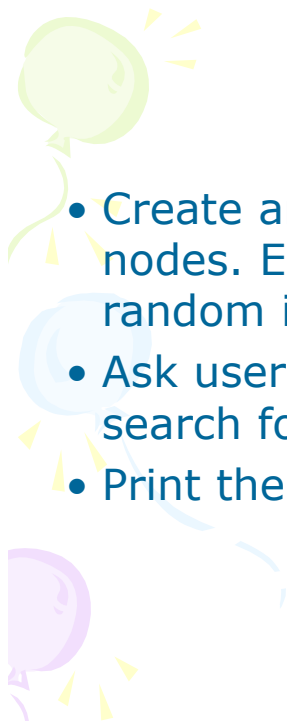
- Write a function to delete all node of a tree. This function must be called before terminating program.

44



## Solution

45



## Exercise

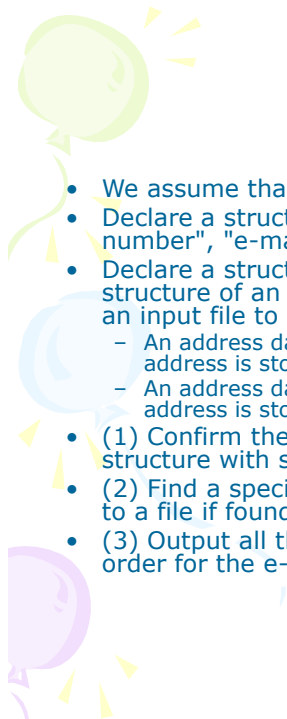
- Create an binary search tree with 10 nodes. Each node contains an random integer.
- Ask user to input an number and search for it.
- Print the content of the trees.

46



## Solution

47



## Exercise

- We assume that you make a mobile phone's address book.
- Declare a structure which can store at least "name", "telephone number", "e-mail address".
- Declare a structure for a binary tree which can stores the structure of an address book inside. Read data of about 10 from an input file to this binary tree as the following rules.
  - An address data which is smaller in the dictionary order for the e-mail address is stored to the left side of a node.
  - An address data which is larger in the dictionary order for the e-mail address is stored to the right side of a node.
- (1) Confirm the address data is organized in the binary tree structure with some methods (printing, debugger, etc).
- (2) Find a specified e-mail address in the binary tree and output it to a file if found.
- (3) Output all the data stored in the binary tree in ascending order for the e-mail address. (Reserve it for the next week)

48



## Solution

```

4 typedef struct phoneAddr_s {
5     char name[20];
6     char tel[12];
7     union {
8         char email[28];
9         char key[28];
10    };
11 } phoneAddr_t;
12 typedef char *key_t;
13
14 typedef phoneAddr_t data_t; // specify a type for the data
15
16 typedef struct node_s {
17     data_t data;
18     struct node_s *left;
19     struct node_s *right;
20 } node_t;
21
22 typedef node_t *tree_t;

```

49

## Search function



50



51

## Insert a node



52

## Solution

## Solution



53