Leibniz Universität Hannover

Programmieren 1

Lists and Trees

Human-Computer Interaction Group

Prof. Dr. Michael Rohs
michael.rohs@hci.uni-hannover.de

Lectures

#	Date	Topic	HÜ→	HÜ←
1	14.10.	Organization, computers, programming, algorithms, PostFix introduction (execution model, IDE, basic operators, booleans, naming)	1	20.10. 23:59
2	21.10.	PostFix (primitive types, functions, parameters, local variables, tests), recipe for atomic data	2	27.10. 23:59
3	28.10.	PostFix (operators, array operations, string operations), recipes for enumerations and intervals	3	3.11. 23:59
4	4.11.	Recipes for compound and variant data, iteration and recursion, PostFix (loops, association arrays, data definitions)	4	10.11. 23:59
5	11.11.	C introduction (if, variables, functions, loops), Programming I C library	5	17.11. 23:59
6	18.11.	Data types, infix expressions, C language (enum, switch)	6	24.11. 23:59
7	25.11.	Compound and variant data, C language (formatted output, struct, union)	7	1.12. 23:59
8	2.12.	C language (arrays, pointers) arrays: fixed-size collections, linear and binary search	8	8.12. 23:59
9	9.12.	Dynamic memory (malloc, free), recursion (recursive data, recursive algorithms)	9	15.12. 23:59
10	16.12.	Linked lists, binary trees, search trees	10	22.12. 23:59
11	13.1.	C language (program structure, scope, lifetime, linkage), function pointers, pointer lists	11	12.1. 23:59
12	20.1.	List and tree operations (filter, map, reduce), objects, object lists	12	19.1. 23:59
13	27.1.	Dynamic data structures (stacks, queues, maps, sets), iterators, documentation tools	(13)	

Review

- Pointers and arrays
  - Strong relationship between pointers and arrays, pointer arithmetic
- Command line arguments
  - Input to a program on the command line
- String and character functions
  - typedef char* String;
- Dynamic memory allocation
  - Automatic storage (call stack), static storage, dynamic heap
  - malloc/calloc, xmalloc/xcalloc, free
  - Memory allocation errors
- Recursive types and algorithms
  - Operations on lists

Preview

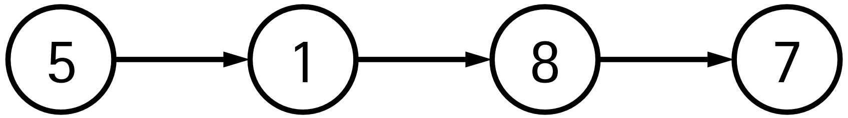
- Linked lists
  - Basic list operations
  - Basic list implementation
  - Ordered insertion
- Binary trees
  - Self-referential, hierarchical data structure
  - Either (1) empty or (2) a node with a value, a left binary tree, and a right binary tree
- Search trees
  - Ordered elements allow for efficient search
- Balanced search trees (optional topic)
  - In each search step exclude about half of the elements

introduced last time:
- dynamic memory allocation and
- recursion / recursive types

needed now

LINKED LISTS

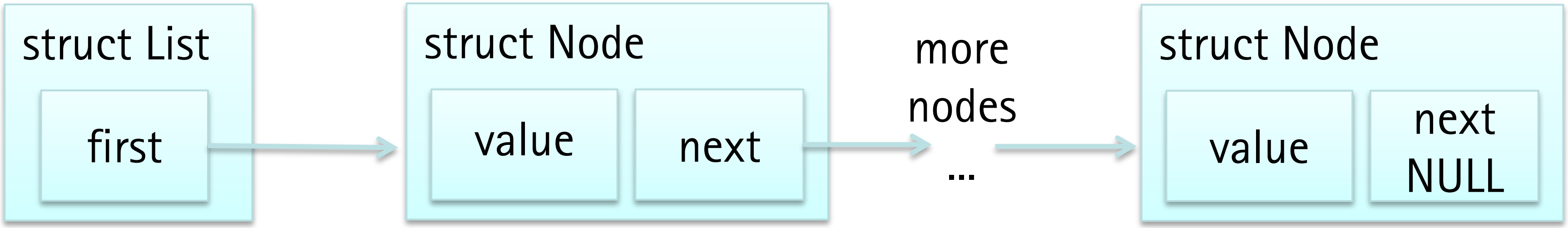
Linked Lists



- Often the size of a collection is initially unknown
  - Example: From a sentence only keep only those words that contain a particular letter
- Linked list: A chain of linked nodes
  - List grows and shrinks as elements are added and removed
  - Example: Word that contains particular letter is added to list
- Recursive list definition
  - A list is either empty or an element followed by a list

Linked Lists

- A lists is a chain of linked nodes
  - Store pointer to first node
  - Each node explicitly points to the next node (except the last one)
  - Direct access to i^{th} element impossible, need to follow the chain
- Each node is a structure
  - Contains (or points to) the actual data
  - Points to next node (and therefore the rest of the list) or is NULL

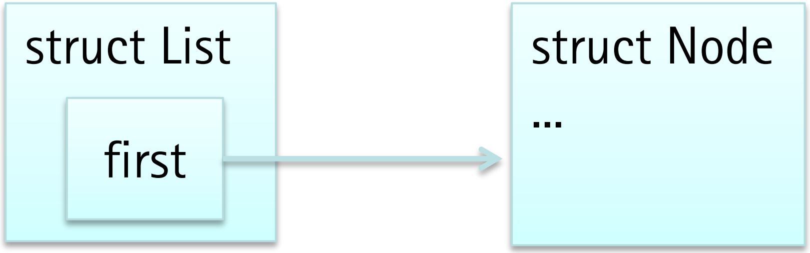


A Linked List of Integers

- Structure for list (head)

not strictly necessary, ignore for now

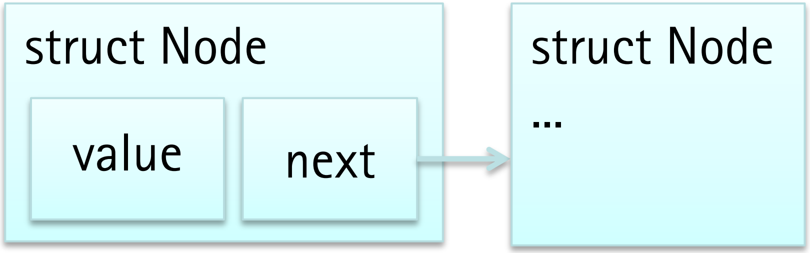
```
typedef struct List {
    Node *first;
} List;
```



- Structure for nodes

```
typedef struct Node {
    int value;
    struct Node *next;
} Node;
```

structural recursion



Typical List Operations (for a Singly Linked List)

new_node: create a list node (heap allocation)	1 step
free_list: release dynamic memory	n steps
print_list: print contents	n steps
length_list: number of elements	n steps
prepend_list: add element to front of list	1 step
append_list: add element to end of list	n steps[1]
insert_list: insert an element at a certain position	i steps
remove_list: remove the element at a certain position	i steps
copy_list: copy each node to get two independent lists	n steps

[1] can be improved to 1 step

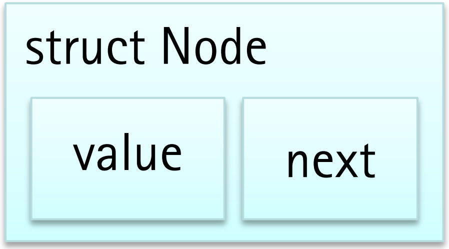
Creating New Nodes

- Dynamic memory allocation
  - for the list head
  - for each node
- Creating the list head

```
List* new_list(void) {
    return xcalloc(1, sizeof(List));
}
```

- Creating and initializing a node

```
Node* new_node(int value, Node* next) {
    Node* node = xcalloc(1, sizeof(Node));
    node->value = value;
    node->next = next;
    return node;
}
```



Creating a List (a Chain of Nodes)

```
// empty list
Node* list = NULL;
print_list(list);
```

list → null

```
// three-element list: 5 1 8
list = new_node(5, new_node(1, new_node(8, NULL)));
print_list(list);
```

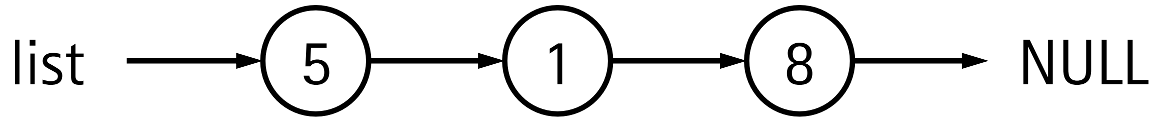
list → 5 → 1 → 8 → NULL

Adding an Element to the Front of a List

```
// Extends the list by adding value at the front.
Node* prepend_list(int value, Node* list) {
    return new_node(value, list);
}
```

- create a new node
- set its element value
- the successor is the old first element
- the new node is the new first element

```
// three-element list: 5 1 8
list = prepend_list(5, prepend_list(1, prepend_list(8, NULL)));
print_list(list);
```

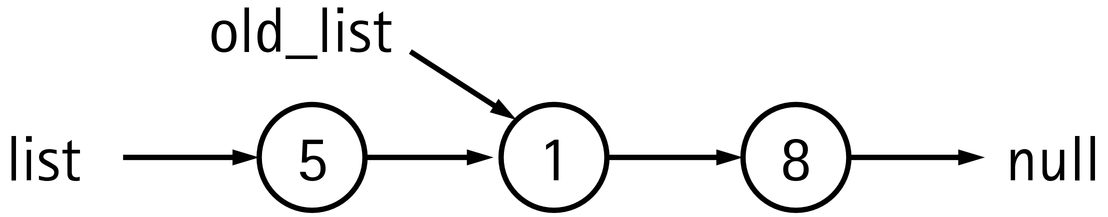


Adding an Element to the Front of a List

```
// Extends the list by adding value at the front.
Node* prepend_list(int value, Node* list) {
    return new_node(value, list);
}
```

- create a new node
- set its element value
- the successor is the old first element
- the new node is the new first element

```
Node* list = prepend_list(5, old_list);
```



prepending 5 does not change old_list

Number of Elements – Recursively and Iteratively

```
int length_list_rec(Node* list) {
    if (list == NULL) {
        return 0; // base case
    } else {
        return 1 + length_list_rec(list->next); // recursive call
    }
}
```

recursive

```
int length_list_iter(Node* list) {
    int result = 0;
    for (Node* node = list; node != NULL; node = node->next) {
        result++; // count this node
    }
    return result;
}
```

iterative

Printing a List – Recursively

```c
// Prints the components of the given list.
void print_list(Node* list) {
    if (list == NULL) {
        printf("\n");
    } else if (list->next == NULL) {
        printf("%d\n", list->value);
    } else {
        printf("%d, ", list->value);
        print_list(list->next);
    }
}
```

`print_list(new_node(10, new_node(20, NULL)));` → `10, 20\n`

Printing a List with Brackets – Recursively

```
print_list2(new_node(10, new_node(20, NULL)));    →  [10, 20]\n
```

```
void print_elements(Node* list) {
    if (list == NULL) return;
    if (list->next == NULL) {
        printf("%d", list->value);
    } else {
        printf("%d, ", list->value);
        print_elements(list->next);
    }
}
```

```
void print_list2(Node* list) {
    printf("[");
    print_elements(list);
    printf("]\n");
}
```

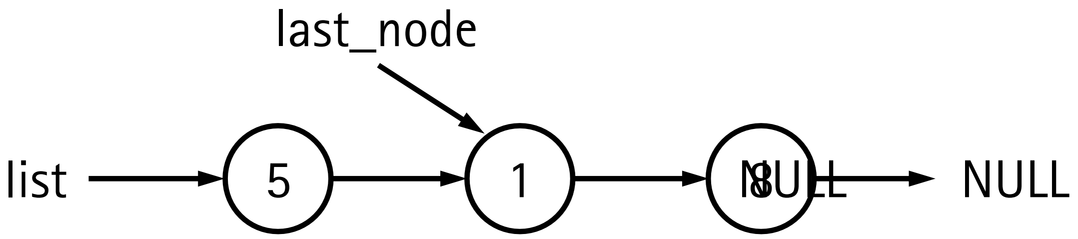
Printing a List with Brackets – Iteratively

```c
// Prints the components of the given list.
void print_list_iter(Node* list) {
    if (list == NULL) {
        printf("[]\n");
    } else {
        printf("[%d", list->value);
        for (Node* n = list->next; n != NULL; n = n->next) {
            printf(", %d", n->value);
        }
        printf("]\n");
    }
}
```

Adding an Element to the End of a List

```
// Adds an element to the end of the list. Modifies the existing list.
Node* append_list(Node* list, int value) {
    if (list == NULL) { // empty list
        return new_node(value, NULL);
    } else { // non-empty list
        last_node(list)->next = new_node(value, NULL);
        return list;
    }
}
```

```
list = append_list(list, 8);
```



Finding the Last Node of a List – Recursively

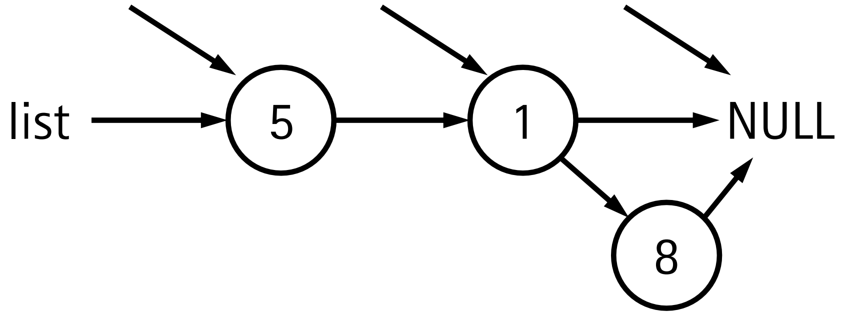
```
// Returns the last node of the list.
// Must only be called on a non-empty list.
Node* last_node(Node* list) {
    require_not_null(list);
    if (list->next == NULL) { // last element?
        return list;
    } else {
        return last_node(list->next); // recursive call
    }
}
```

Node: Compiler option **-O2** performs tail-call optimization, which makes such recursive calls as efficient as loops

Adding an Element to the End of a List – Recursively

```
// Adds an element to the end of the list.
// Modifies the existing list. Recursive version.
Node* append_list2(Node* list, int value) {
    if (list == NULL) {
        return new_node(value, NULL);
    } else {
        list->next = append_list2(list->next, value); // recursive call
        return list;
    }
}
```

```
list = append_list2(list, 8);
```



Adding an Element to the End of a List – Iteratively

```
// Adds an element to the end of the list.
// Modifies the existing list. Iterative version.
Node* append_list_iter(Node* list, int value) {
    if (list == NULL) { // empty list
        return new_node(value, NULL);
    } else { // non-empty list
        Node* n = list;
        while (n->next != NULL) n = n->next; // find last element
        assert("on last element", n != NULL && n->next == NULL);
        n->next = new_node(value, NULL);
        return list;
    }
}
```

Efficiently Adding at the End of a List

- List head with pointers to first and last nodes

```
struct Lst {
    struct Node *first;
    struct Node *last;
};
```

Named Lst to avoid name collision with prog1lib List

```
struct Node {
    double value;
    struct Node *next;
};
```

- Add to end in one step (independent of list length)
  - Without last-pointer requires list traversal (n steps)
- Invariants
  - Empty list: first == NULL && last == NULL
  - One element: first != NULL && last != NULL && first == last
  - Two or more elements: first != NULL && last != NULL && first != last

Efficiently Adding at the End of a List

```
void efficient_append_list(Lst* list, int value) {
    require("list head exists", list != NULL);
    Node* n = new_node(value, NULL);
    if (list->first == NULL) { // empty list, first and last change
        list->first = n;
        list->last = n;
    } else { // non-empty list, only last changes
        list->last->next = n;
        list->last = n;
    }
}
```

```
struct Node {
    double value;
    struct Node *next;
};
struct Lst {
    struct Node *first;
    struct Node *last;
};
```

Inserting an Element in Order

Inserting a value at the right position, such that increasing order is maintained

```
Node* insert_ordered(Node* list, int value) {
    if (list == NULL) { // empty list
        return new_node(value, NULL);
    } else if (value  <  list->value) { // insert before first
        return new_node(value, list);
    } else { // non-empty list, find insertion position after first node
        for (Node* n = list; n != NULL; n = n->next) {
            if (n->next == NULL) { // end of list?
                n->next = new_node(value, n->next);    break;
            } else if (value < n->next->value) { // found position?
                n->next = new_node(value, n->next);    break;
            }
        }
        return list;
    }
}
```

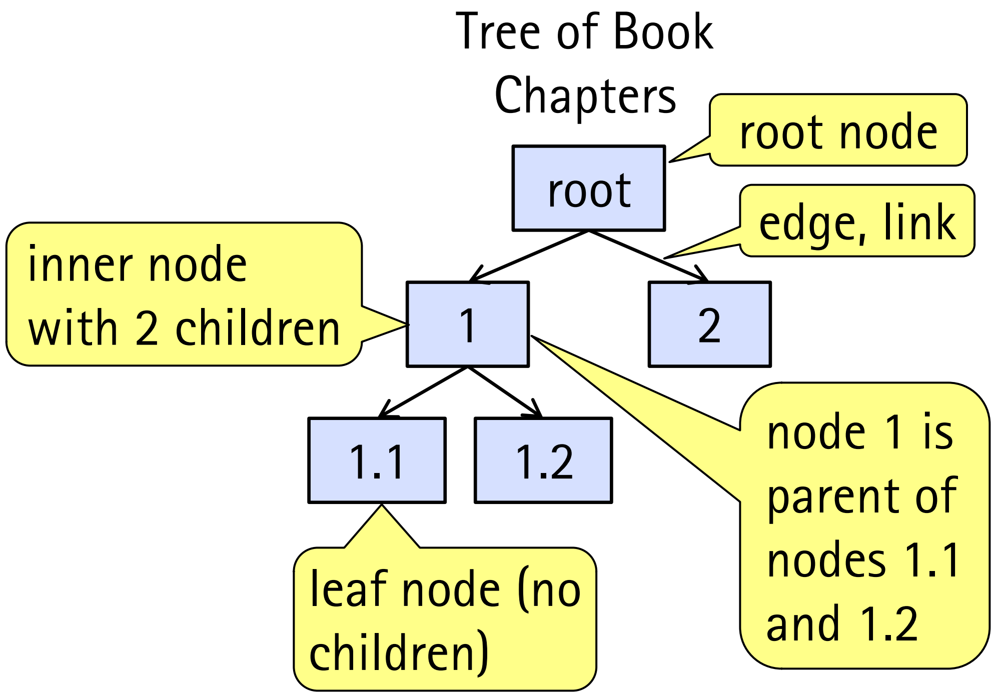
Inserting an Element in Order with Pre- and Postcondition

```
Node* insert_ordered(Node* list, int value) {
    require("sorted in non-decreasing order",
            forall(Node* n = list, n != NULL && n->next != NULL, n = n->next, n->value <= n->next->value));
    Node* result = list;
    if (list == NULL) { // empty list
        result = new_node(value, NULL);
    } else if (value  <  list->value) { // insert before first
        result = new_node(value, list);
    } else { // find insertion position in non-empty list, after first node
        for (Node* n = list; n != NULL; n = n->next) { /* as before... */ }
        result = list;
    }
    ensure("sorted in non-decreasing order",
           forall(Node* n = result, n != NULL && n->next != NULL, n = n->next, n->value <= n->next->value));
    return result;
}
```

BINARY TREES

Trees

- A hierarchical arrangement of elements
  - Chapters, sections, paragraphs in a book
  - Efficient access to sorted data
- Terminology
  - Root is the only node without a parent node
  - Other nodes have exactly one parent node
  - Inner nodes have 1 or more children
  - Leaf nodes do not have children
  - An edge connects a parent to a child
    - No edges between siblings, no cycles



Self-Referential Data: Binary Trees

A binary tree is either empty (variant 1)
or a left tree, a value, and a right tree (variant 2)

```
struct Node {
    int value;
    struct Node* left; // self-reference or NULL
    struct Node* right; // self-reference or NULL
};
typedef struct Node Node;

struct BinTree {
    Node* first; // first element or NULL
};
typedef struct BinTree BinTree;
```

pointers can be NULL, so represent both variants

BinTree structure, not strictly necessary

Data Definition: Binary Tree Constructor Functions

```c
// Create a binary tree node.
Node* new_node(int value, Node* left, Node* right) {
    Node* n = xcalloc(1, sizeof(Node));
    n->value = value;
    n->left = left;
    n->right = right;
    return n;
}

// Create a binary tree.
BinTree* new_bintree(void) {
    BinTree* t = xcalloc(1, sizeof(BinTree));
    t->first = NULL;
    return t;
}
```

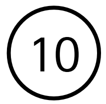
1. Problem Statement

- Write down the problem statement as a comment.
  - What is the relevant information?
  - What should the function do with the data?
- Example

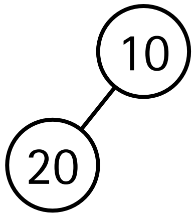
```
/*
Compute the sum of the values of a binary tree of integer numbers.
*/
```

2b. Example Values for Data Definition

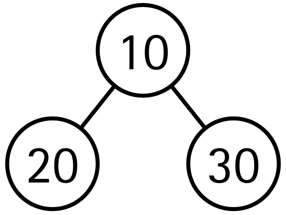
- Create at least one example value per variant in the data definition
- Create examples that use the self-referential variant(s) more than once (i.e., create examples of different lengths)
- Examples
  - `Node* tree = NULL; // variant 1 (base case)`
  - `tree = new_node(10, NULL, NULL); // value, left, right`



  - `tree = new_node(10, new_node(20, NULL, NULL), NULL);`




  - ```
    tree = new_node(10,
                    new_node(20, NULL, NULL),
                    new_node(30, NULL, NULL));
    ```


    ```

3. Function Name and Parameter List

- Find a good function name
  - Short, non-abbreviated, descriptive name that describes what the function does
- Find good parameter names
  - Short, non-abbreviated, descriptive name that describes what the parameter means
- Write function header
- Example

```
int sum(Node* tree);
```

4a. Function Stub

- Function stub returns an arbitrary value from the function's range
- The function stub can be executed
- Example

```
int sum(Node* tree) {
    return 0;
}
```

4b. Purpose Statement

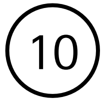
- Briefly describes what the function does (not how!).
  Ideally as a single sentence. Multiple sentences may be necessary.
- Example

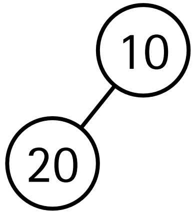
```
// Computes the sum of the values of the tree.
int sum(Node* tree) {
    return 0;
}
```

5b. Examples and Expected Results (Test Function)

- Write several examples with expected results, at least one per variant in the data definition
  - Use the example values created before (in 2b)

```
void sum_test(void) {
    Node* tree = NULL;
    test_equal_i(sum(tree), 0);
    tree = new_node(10, NULL, NULL);
    test_equal_i(sum(tree), 10);
    tree = new_node(10, new_node(20, NULL, NULL), NULL);
    test_equal_i(sum(tree), 30);
}
```





6. Template

- Translate the data definition into a template
- Use if-else to handle the different variants
  - Conditions: Write one condition per variant
  - Actions: Access members relevant for the respective variant
  - Actions: Add one recursive call per self-reference

6. Template

- Data definition

```
struct Node {
    int value;
    struct Node* left; // self-reference or NULL
    struct Node* right; // self-reference or NULL
};
```

self-references

- Translate the data definition into a template

```
int sum(Node* tree) {
    if (tree == NULL) {
        ...
    } else {
        ... tree->value ... sum(tree->left...right) ...
    }
}
```

non-recursive base case

recursive case

self-reference

recursive calls on self-references

6. Function Body

- Combine expressions in template to obtain expected values
- For the recursion: Assume that the function already works (induction hypothesis)

```
// Computes the sum of the values of the tree.
int sum(Node* tree) {
    if (tree== NULL) {
        ...
    } else {
        ... tree->value ...
        ... sum(tree->left) ...
        ... sum(tree->right) ...
    }
}
```

6. Function Body

- Combine expressions in template to obtain expected values
- For the recursion: Assume that the function already works (induction hypothesis)

```
// Computes the sum of the values of the tree.
int sum(Node* tree) {
    if (tree== NULL) {
        return 0;
    } else {
        ... tree->value ...
        ... sum(tree->left) ...
        ... sum(tree->right) ...
    }
}
```

purpose statement

sum of empty list is 0 (base case)

assume that sum already does what the purpose statement says

then do induction step ("leap of faith")

6. Function Body

- Combine expressions in template to obtain expected values
- For the recursion: Assume that the function already works (induction hypothesis)

```
// Computes the sum of the values of the tree.
int sum(Node* tree) {
    if (tree== NULL) {
        return 0;
    } else {
        return tree->value +
            sum(tree->left) +
            sum(tree->right);
    }
}
```

purpose statement

sum of empty list is 0 (base case)

if sum(left) = x and sum(right) = y (induction hypothesis), then sum(node(v, left, right)) = v + x + y (induction step)

7. Testing

- Call test function

```
int main(void) {
    sum_test();
    return 0;
}
```

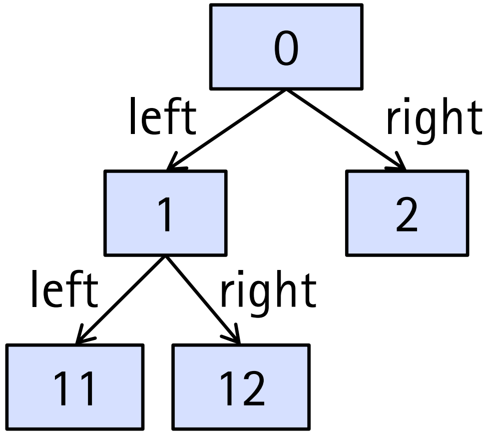
- Test results

```
bintree.c, line 116: check passed
bintree.c, line 118: check passed
bintree.c, line 119: check passed
All 3 tests passed!
```

Printing a Binary Tree

```
void print_tree(BTNode* tree) {
    if (tree == NULL) {
        printf("_");
    } else {
        printf("(");
        print_tree(tree->left);
        printf(", %d, ", tree->value);
        print_tree(tree->right);
        printf(")");
    }
}
```

Binary Tree




```
BTNode* t = new_btnode(0,
    new_btnode(1, new_leaf(11), new_leaf(12)),
    new_leaf(2));
print_tree(t);
```

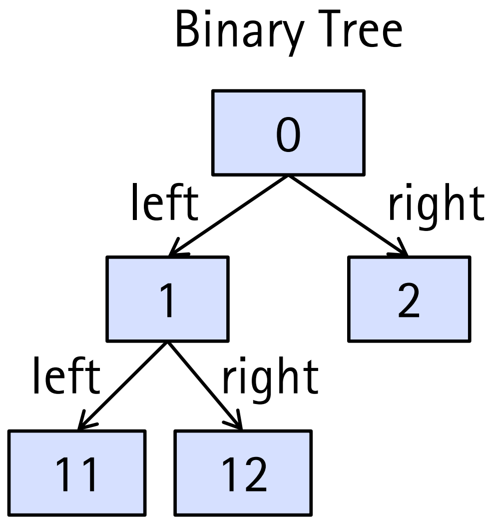
Output:

(((_, 11, _), 1, (_, 12, _)), 0, (_, 2, _))

Counting the Values of a Binary Tree

```
// Counts the number of values of a binary tree.
int count_tree(BTNode* tree) {
    if (tree == NULL) return 0;
    else return 1 + count_tree(tree->left) + count_tree(tree->right);
}
```

count = 5

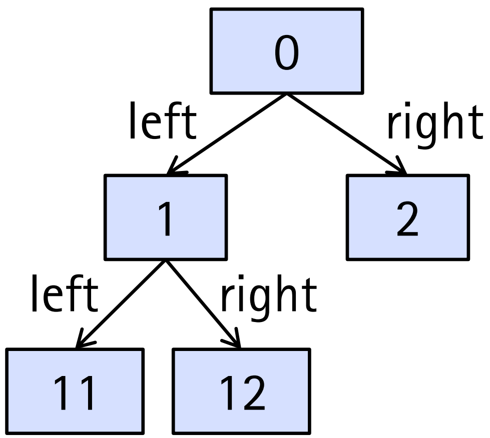


Checking whether a Binary Tree Contains a Given Value

```
// Checks whether the binary tree contains x.
bool contains_tree(BTNode* tree, int x) {
    if (tree == NULL) return false;
    else return (tree->value == x) ||   // we found it
        contains_tree(tree->left, x) ||   // check whether it is in the left subtree
        contains_tree(tree->right, x);   // check whether it is in the right subtree
}
```

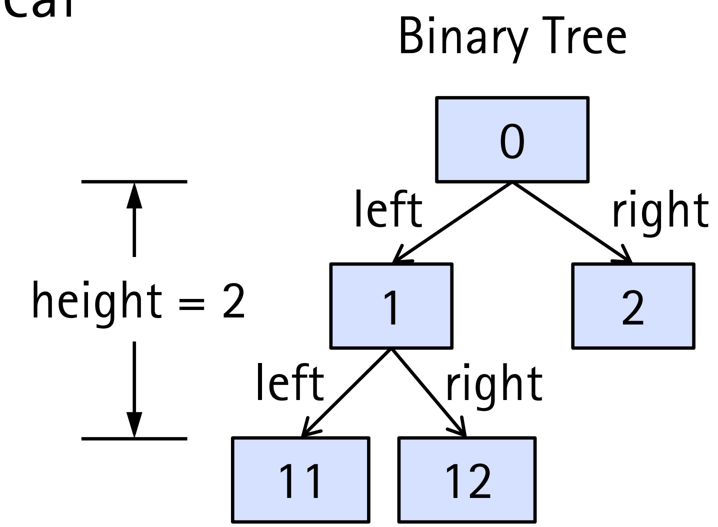
contains_tree(t, 5) → false
contains_tree(t, 12) → true

Binary Tree



Computing the Height of a Binary Tree

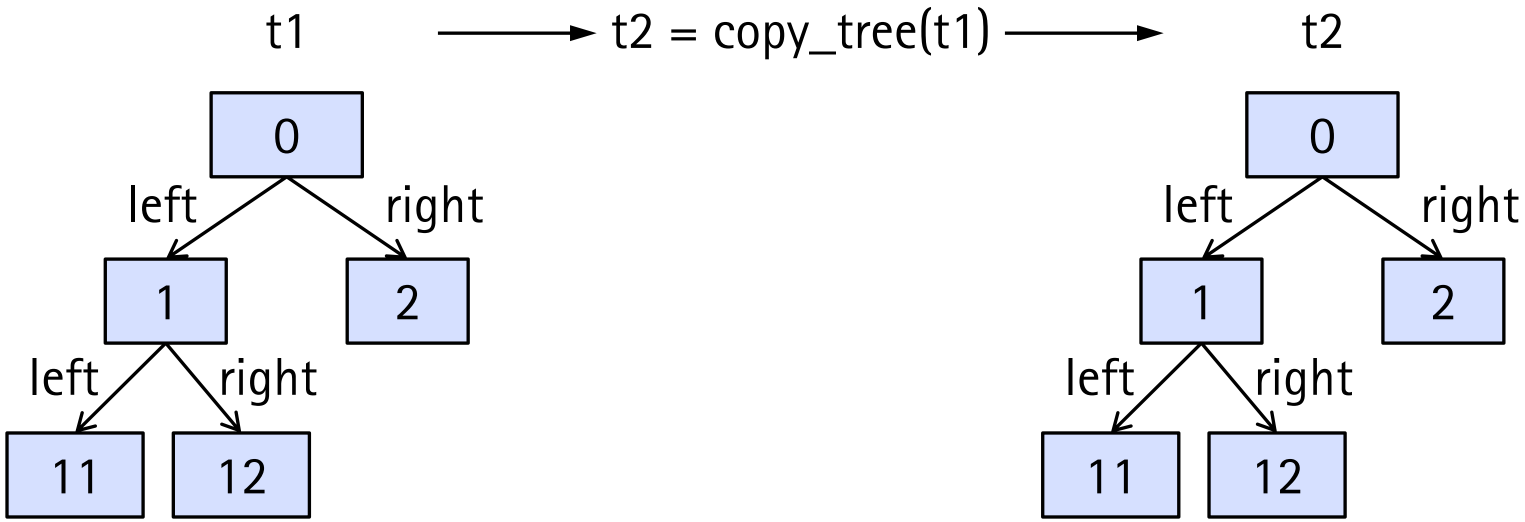
- Height: Number of edges from root to a deepest leaf
- Equivalently: The length of a longest path from the root to a leaf




```c
int height_tree(BTNode* tree) {
    if (tree == NULL) {
        return 0;
    } else {
        if (tree->left == NULL && tree->right == NULL) return 0;
        int left_height = height_tree (tree->left);
        int right_height = height_tree (tree->right);
        return 1 + ((left_height > right_height) ? left_height : right_height);
    }
}
```

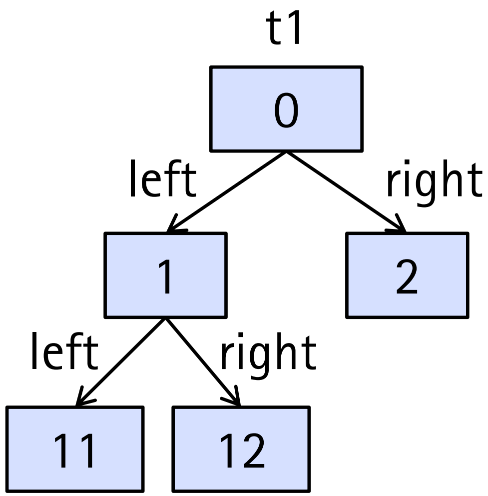
Copying a Binary Tree

```
// Copies each node of a binary tree.
BTNode* copy_tree(BTNode* tree) {
    if (tree == NULL) return NULL;
    return new_btnode(tree->value, copy_tree(tree->left), copy_tree(tree->right));
}
```

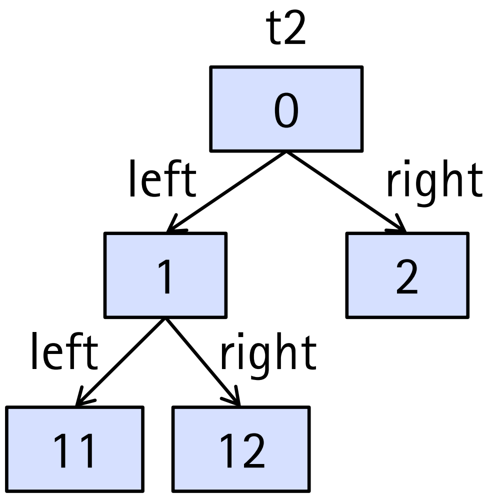


Checking whether two Binary Trees are Equal

```
bool trees_equal(BTNode* t1, BTNode* t2) {
    if (t1 == t2) return true; // same address (or both NULL)
    // assert: t1 != t2
    if (t1 == NULL || t2 == NULL) return false; // one tree is NULL, the other is not
    // assert: t1 != NULL && t2 != NULL
    if (t1->value != t2->value) return false; // trees differ at current node
    // assert: t1->value == t2->value
    return trees_equal(t1->left, t2->left) && trees_equal(t1->right, t2->right);
}
```



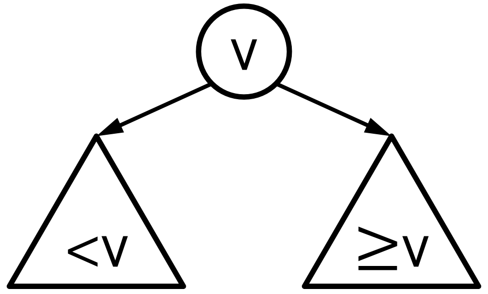
trees_equal(t1, t2)



Ordered Insertion into Binary Trees

```
BTNode* insert_ordered(BTNode* tree, int value) {
    require("is ordered", is_ordered(tree));
    if (tree == NULL) { // empty tree
        tree = new_leaf(value);
    } else if (value < tree->value) { // insert in left subtree
        tree->left = insert_ordered(tree->left, value);
    } else { // insert in right subtree
        tree->right = insert_ordered(tree->right, value);
    }
    ensure("is ordered", is_ordered(tree));
    return tree;
}
```

Insert so that for each node with value v:
left subtree (if not empty) contains values smaller v and right subtree (if not empty) contains values equal to or larger than v



Ordered Insertion into Binary Trees

Example: Insert 10, 3, 5, 15, 20, 12, 1

insert: 10   tree

NULL

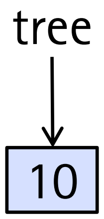
```
BTNode* insert_ordered(BTNode* tree, int value) {
    require("is ordered", is_ordered(tree));
    if (tree == NULL) { // empty tree
        tree = new_leaf(value);
    } else if (value < tree->value) {// insert in left subtree
        tree->left = insert_ordered(tree->left, value);
    } else { // insert in right subtree
        tree->right = insert_ordered(tree->right, value);
    }
    ensure("is ordered", is_ordered(tree));
    return tree;
}
```

Ordered Insertion into Binary Trees

Example: Insert 10, 3, 5, 15, 20, 12, 1

```
BTNode* insert_ordered(BTNode* tree, int value) {
    require("is ordered", is_ordered(tree));
    if (tree == NULL) { // empty tree
        tree = new_leaf(value);
    } else if (value < tree->value) {// insert in left subtree
        tree->left = insert_ordered(tree->left, value);
    } else { // insert in right subtree
        tree->right = insert_ordered(tree->right, value);
    }
    ensure("is ordered", is_ordered(tree));
    return tree;
}
```

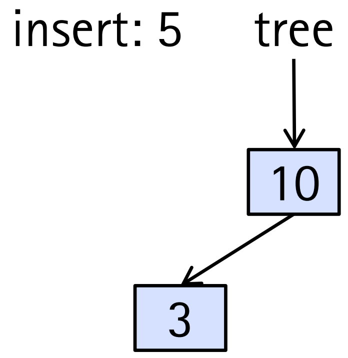
insert: 3



Ordered Insertion into Binary Trees

Example: Insert 10, 3, 5, 15, 20, 12, 1

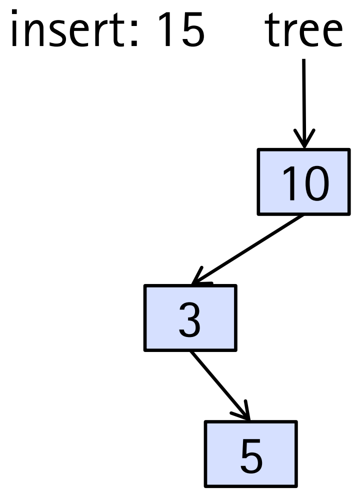
```
BTNode* insert_ordered(BTNode* tree, int value) {
    require("is ordered", is_ordered(tree));
    if (tree == NULL) { // empty tree
        tree = new_leaf(value);
    } else if (value < tree->value) {// insert in left subtree
        tree->left = insert_ordered(tree->left, value);
    } else { // insert in right subtree
        tree->right = insert_ordered(tree->right, value);
    }
    ensure("is ordered", is_ordered(tree));
    return tree;
}
```



Ordered Insertion into Binary Trees

Example: Insert 10, 3, 5, 15, 20, 12, 1

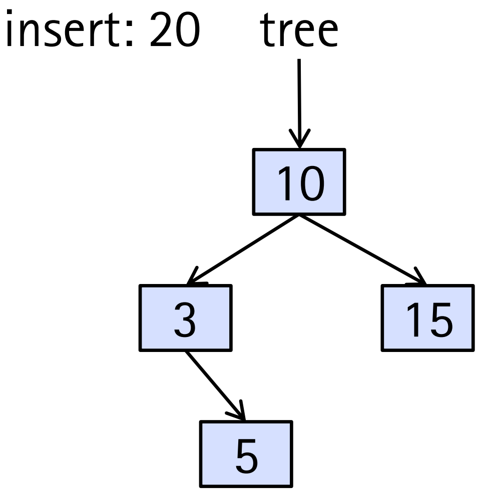
```
BTNode* insert_ordered(BTNode* tree, int value) {
    require("is ordered", is_ordered(tree));
    if (tree == NULL) { // empty tree
        tree = new_leaf(value);
    } else if (value < tree->value) {// insert in left subtree
        tree->left = insert_ordered(tree->left, value);
    } else { // insert in right subtree
        tree->right = insert_ordered(tree->right, value);
    }
    ensure("is ordered", is_ordered(tree));
    return tree;
}
```



Ordered Insertion into Binary Trees

Example: Insert 10, 3, 5, 15, 20, 12, 1

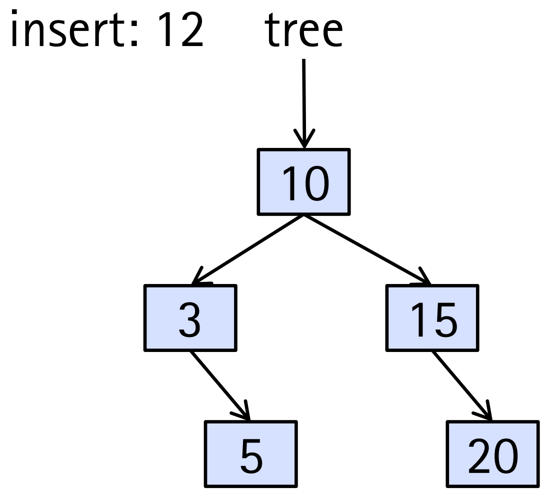
```
BTNode* insert_ordered(BTNode* tree, int value) {
    require("is ordered", is_ordered(tree));
    if (tree == NULL) { // empty tree
        tree = new_leaf(value);
    } else if (value < tree->value) {// insert in left subtree
        tree->left = insert_ordered(tree->left, value);
    } else { // insert in right subtree
        tree->right = insert_ordered(tree->right, value);
    }
    ensure("is ordered", is_ordered(tree));
    return tree;
}
```



Ordered Insertion into Binary Trees

Example: Insert 10, 3, 5, 15, 20, 12, 1

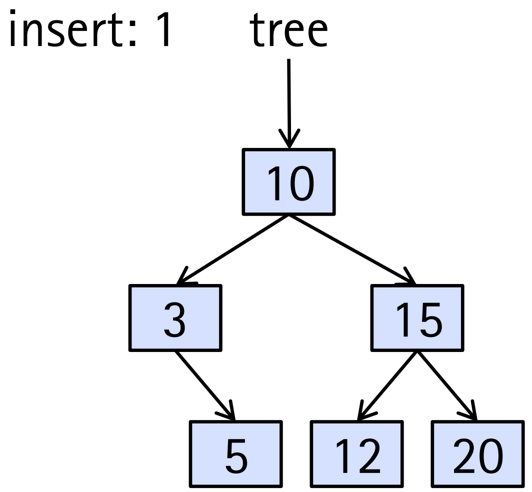
```
BTNode* insert_ordered(BTNode* tree, int value) {
    require("is ordered", is_ordered(tree));
    if (tree == NULL) { // empty tree
        tree = new_leaf(value);
    } else if (value < tree->value) {// insert in left subtree
        tree->left = insert_ordered(tree->left, value);
    } else { // insert in right subtree
        tree->right = insert_ordered(tree->right, value);
    }
    ensure("is ordered", is_ordered(tree));
    return tree;
}
```



Ordered Insertion into Binary Trees

Example: Insert 10, 3, 5, 15, 20, 12, 1

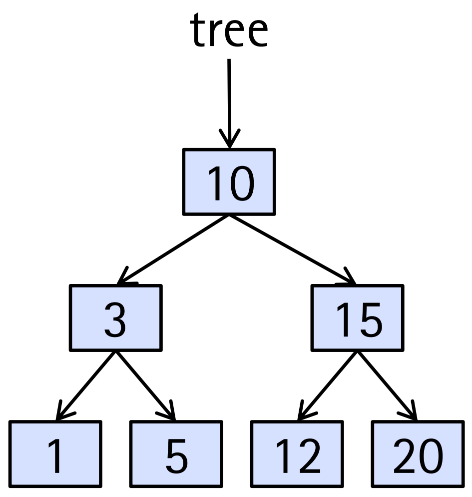
```
BTNode* insert_ordered(BTNode* tree, int value) {
    require("is ordered", is_ordered(tree));
    if (tree == NULL) { // empty tree
        tree = new_leaf(value);
    } else if (value < tree->value) {// insert in left subtree
        tree->left = insert_ordered(tree->left, value);
    } else { // insert in right subtree
        tree->right = insert_ordered(tree->right, value);
    }
    ensure("is ordered", is_ordered(tree));
    return tree;
}
```



Ordered Insertion into Binary Trees

Example: Insert 10, 3, 5, 15, 20, 12, 1

```
BTNode* insert_ordered(BTNode* tree, int value) {
    require("is ordered", is_ordered(tree));
    if (tree == NULL) { // empty tree
        tree = new_leaf(value);
    } else if (value < tree->value) {// insert in left subtree
        tree->left = insert_ordered(tree->left, value);
    } else { // insert in right subtree
        tree->right = insert_ordered(tree->right, value);
    }
    ensure("is ordered", is_ordered(tree));
    return tree;
}
```



Ordered Insertion into Binary Trees – Iterative Version

```
BTNode* insert_ordered_tree(BTNode* tree, int value)
{
    require("ordered", is_ordered_tree(tree));
    if (tree == NULL) { // empty tree
        tree = new_leaf(value);
    } else {
        BTNode* t = tree;
        while (true) {
            if (value < t->value) {
                if (t->left) {
                    t = t->left;
                } else {
                    t->left = new_leaf(value);
                    break;
                }
            } else /* value >= t->value */ {
                if (t->right) {
                    t = t->right;
                } else {
                    t->right = new_leaf(value);
                    break;
                }
            } // else
        } // while
    } // else
    ensure("ordered", is_ordered_tree(tree));
    return tree;
}
```

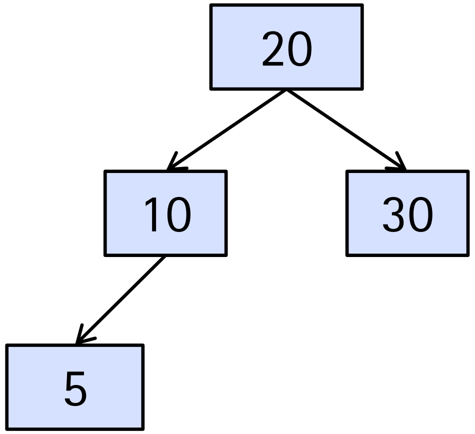
Printing an Ordered Binary Tree as an Ordered Sequence

- Print elements of ordered tree in the right order

```
void print_inorder(BTNode* tree) {
    if (tree != NULL) {
        print_inorder(tree->left);   // first, print values less than tree->value
        printf("%d ", tree->value); // then, print tree->value
        print_inorder(tree->right); // last, print values greater than (or =) tree->value
    }
}
```

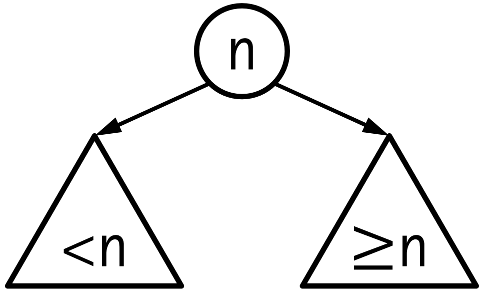
- Example:

```
BTNode* t = NULL;
t = insert_ordered(t, 20); t = insert_ordered(t, 10);
t = insert_ordered(t, 30); t = insert_ordered(t, 5);
println_inorder(t); // 5 10 20 30 ← sorted output
```



Efficient Search in Ordered Binary Trees

- If the values in the tree are ordered, search can be more efficient
- Idea: Either found in root or only search in left/right subtree
- Can exclude a whole subtree in each step
- Typically immense speedup
- Ordered binary trees are sometimes called search trees

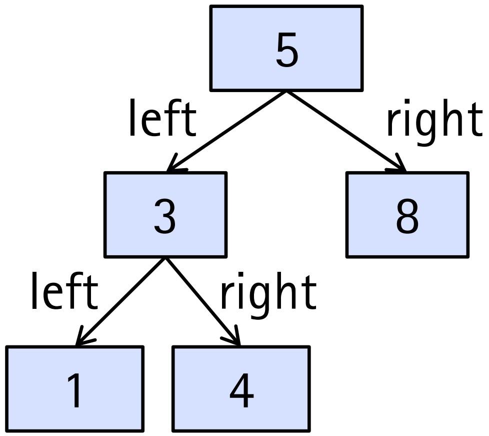


- How many search steps in an ordered binary tree?
  - Best case?
  - Worst case?

Efficient Search in Ordered Binary Trees

```
BTNode* search_ordered(BTNode* tree, int x) {
    if (tree == NULL) return NULL;
    if (x == tree->value) return tree;
    if (x < tree->value) return search_ordered(tree->left, x);
    return search_ordered(tree->right, x);
}
```

Binary Tree (ordered)

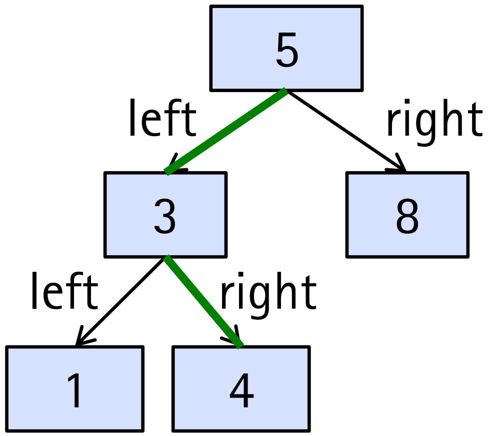


Efficient Search in Ordered Binary Trees

```
BTNode* search_ordered(BTNode* tree, int x) {
    if (tree == NULL) return NULL;
    if (x == tree->value) return tree;
    if (x < tree->value) return search_ordered(tree->left, x);
    return search_ordered(tree->right, x);
}
```

search_ordered(tree, 4)

Binary Tree (ordered)

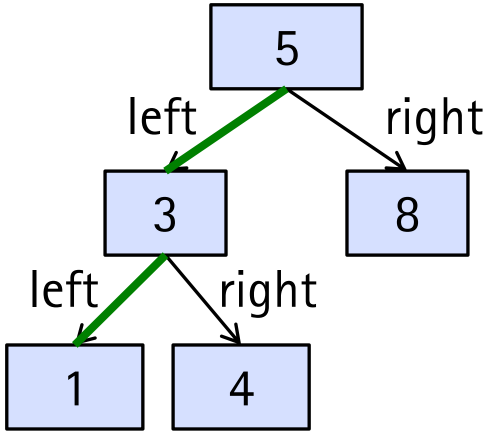


Efficient Search in Ordered Binary Trees

```
BTNode* search_ordered(BTNode* tree, int x) {
    if (tree == NULL) return NULL;
    if (x == tree->value) return tree;
    if (x < tree->value) return search_ordered(tree->left, x);
    return search_ordered(tree->right, x);
}
```

search_ordered(tree, 2)

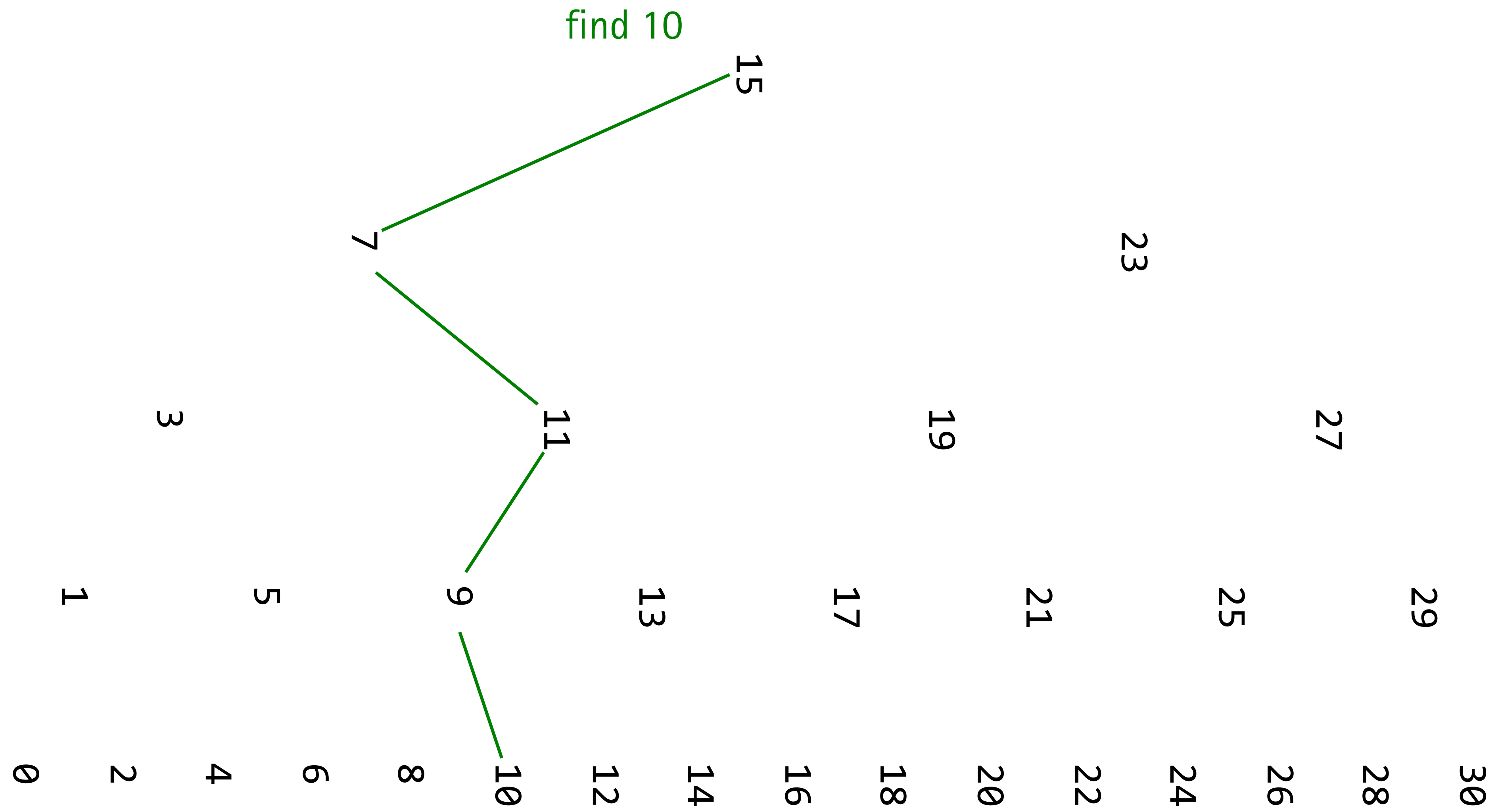
Binary Tree (ordered)



Efficient Search in Ordered Binary Trees – Iterative Version

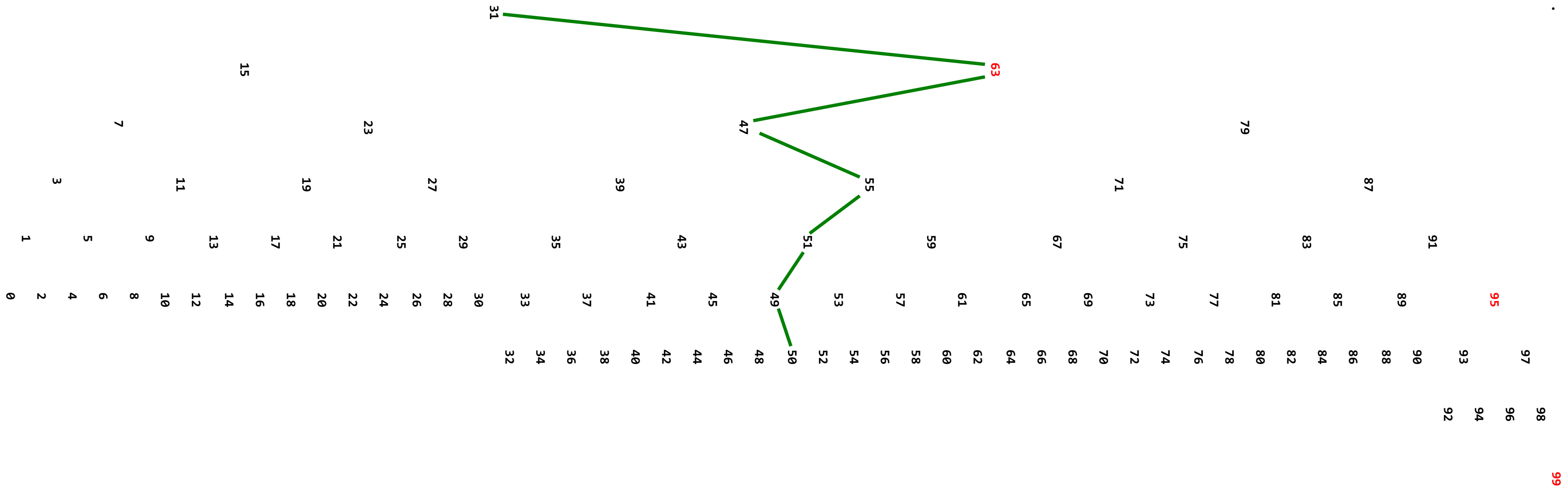
```
BTNode* search_ordered_tree_iter(BTNode* tree, int x) {
    require("ordered", is_ordered_tree(tree));
    while (tree != NULL) {
        if (x < tree->value) tree = tree->left;
        else if (x > tree->value) tree = tree->right;
        else return tree;
    }
    return NULL;
}
```

Example Search Tree: 31 Elements → 5 Comparisons



Example Search Tree: 100 Elements → 7 Comparisons

find 50



Summary

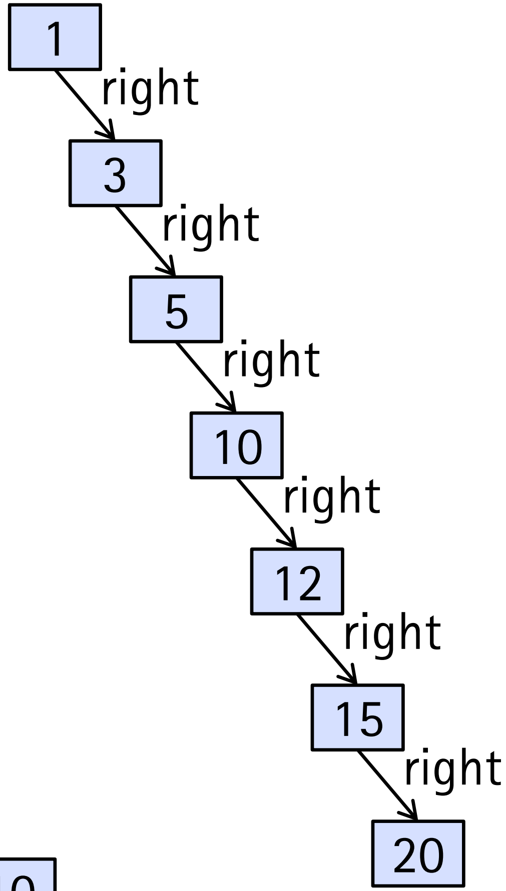
- Linked lists
  - Basic list operations
  - Basic list implementation
  - Ordered insertion
- Binary trees
  - Self-referential, hierarchical data structure
  - Either (1) empty or (2) a node with a value, a left binary tree, and a right binary tree
- Search trees
  - Ordered elements allow for efficient search
- Balanced search trees (optional topic)
  - In each search step exclude about half of the elements

(optional topic)

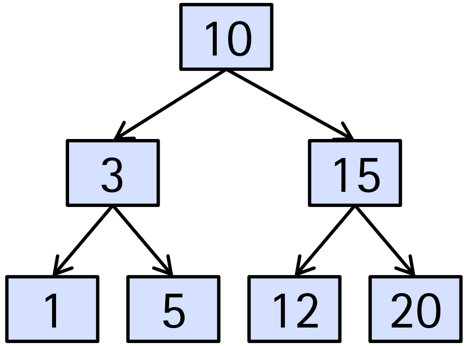
BALANCED SEARCH TREES

Balanced Search Trees

- What happens if inserted elements are already ordered?
  - Search tree will degenerate to a list
  - Inefficient search
  - Example: Inserting 1, 3, 5, 10, 12, 15, 20

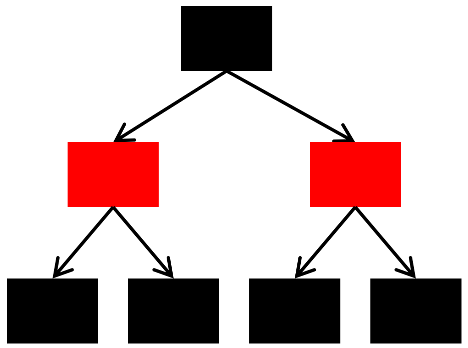


- What to do?
  - Ensure that the tree is always balanced
  - Efficient search
- How?
  - Red-black trees (one possibility)

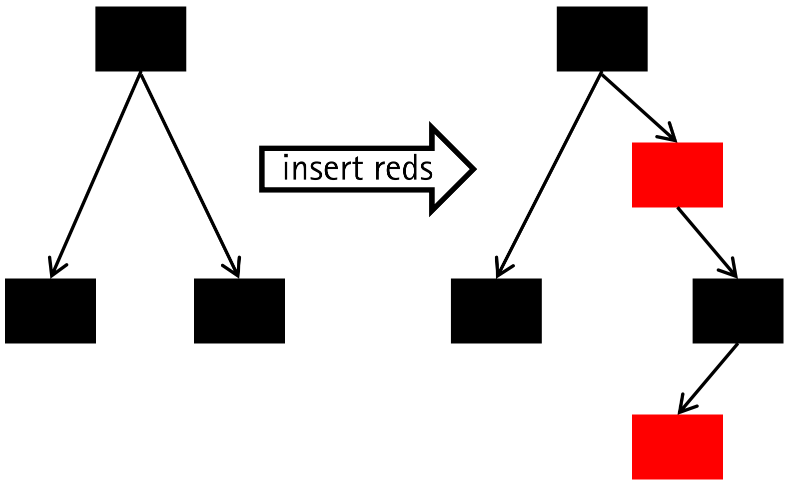


Red–Black Trees

- Ordered binary tree with these additional properties
  1. A node is either red or black
  2. The root is black
  3. No red node has a red parent
  4. Every path from the root to a leaf has the same number of black nodes

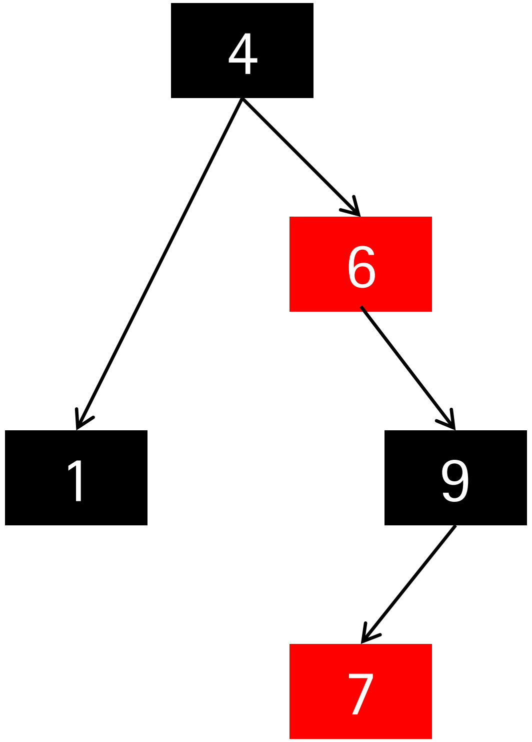


- These properties ensure that red-black trees are (sufficiently) balanced. Why?
- Longest path can at most be twice as long as shortest path. Why?



Red-Black Trees in C: Data Definition

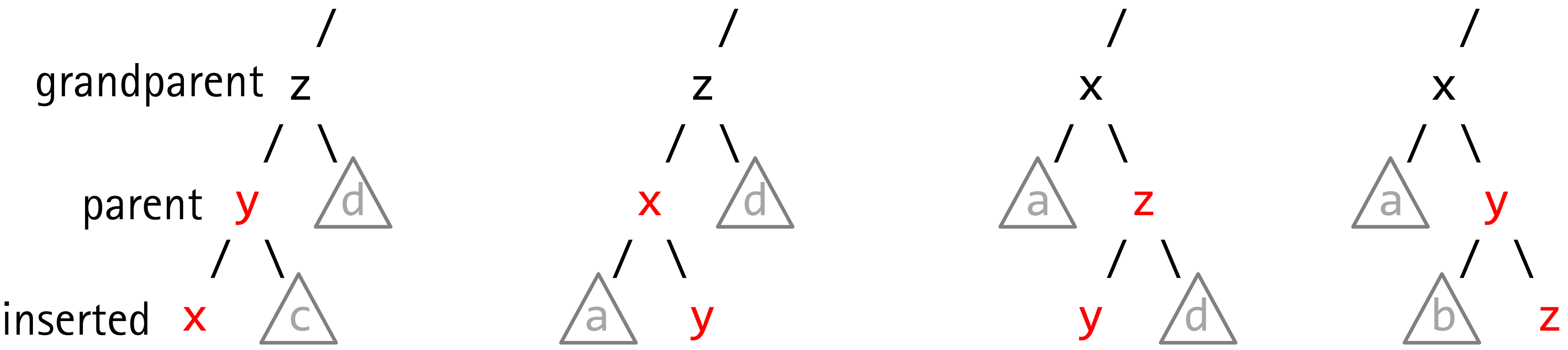
```
// The color of a node may be black or red
typedef enum { B, R } Color;

// Represents a single tree node.
typedef struct RBNode {
    Color color;
    int value;
    struct RBNode* left; // self-reference
    struct RBNode* right; // self-reference
} RBNode;
```



Insertion in Red-Black Trees

- Find insertion point (as with ordinary search tree)
- Insert a red leaf node at the insertion point
- Does not change number of black nodes (no violation of rule 4)
- Inserted node may have a red parent (violation of rule 3)
- Consider parent and grandparent to fix the violation

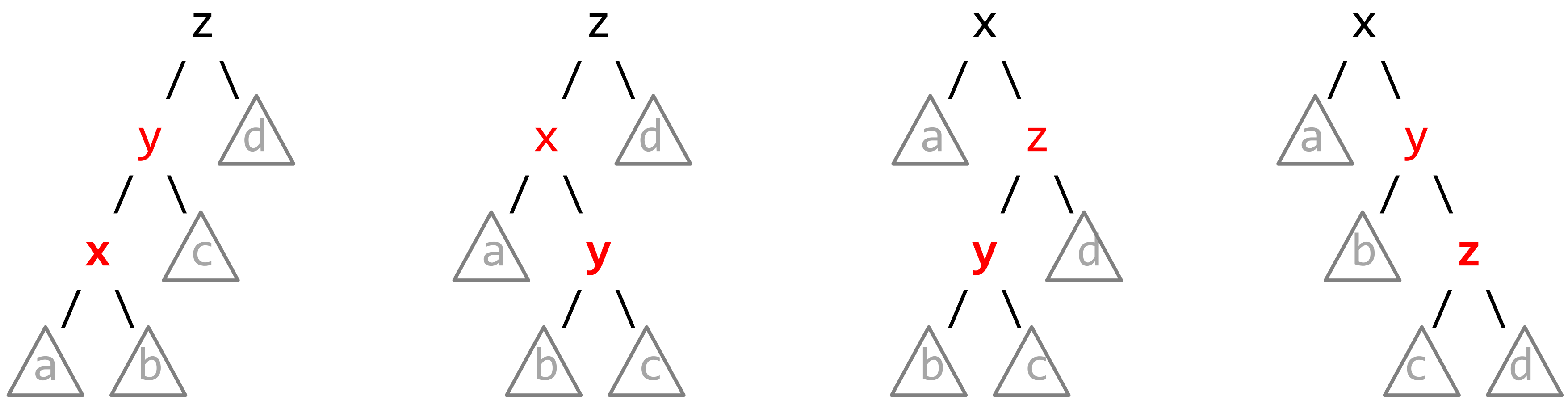
1. A node is either red or black
2. The root is black
3. No red node has a red parent
4. Every path from the root to a leaf has the same number of black nodes



are subtrees, possibly empty

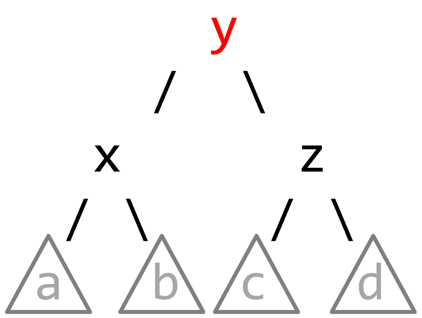
Sort order: a<x<b<y<c<z<d

Possible Violations through Insertion



1. A node is either red or black
2. The root is black
3. No red node has a red parent
4. Every path from the root to a leaf has the same number of black nodes

Restoring rule 3 (no red node has a red parent) without violating rule 4 (every path from root to a leaf has the same number of black nodes):



Sort order: a<x<b<y<c<z<d

Problem: Red y may violate rule 3 in next level up
→ Fixing recursively

Red-Black Trees in C: Add an Element

```
RBNode* add_rbtree(RBNode* tree, int x) {
    if (tree == NULL) {
        return new_leaf(B, x);
    } else {
        add_rbtree_rec(tree, x);
        tree->color = B;   // make the root black (increases by 1 the
        return tree;       // number of black nodes on all paths to a leaf)
    }
}
```

1. A node is either red or black
2. The root is black
3. No red node has a red parent
4. Every path from the root to a leaf has the same number of black nodes

Red-Black Trees in C: Add an Element

```c
RBNode* add_rbtree_rec(RBNode* tree, int x) {
    if (x < tree->value) {
        if (tree->left != NULL) {
            add_rbtree_rec(tree->left, x);
            balance(tree); // may need to rebalance the tree
        } else {
            tree->left = new_leaf(R, x); // found insertion point
        }
    } else if (x > tree->value) {
        if (tree->right != NULL) {
            add_rbtree_rec(tree->right, x);
            balance(tree); // may need to rebalance the tree
        } else {
            tree->right = new_leaf(R, x); // found insertion point
        }
    }
    return tree;
}
```

Red-Black Trees in C: Balancing

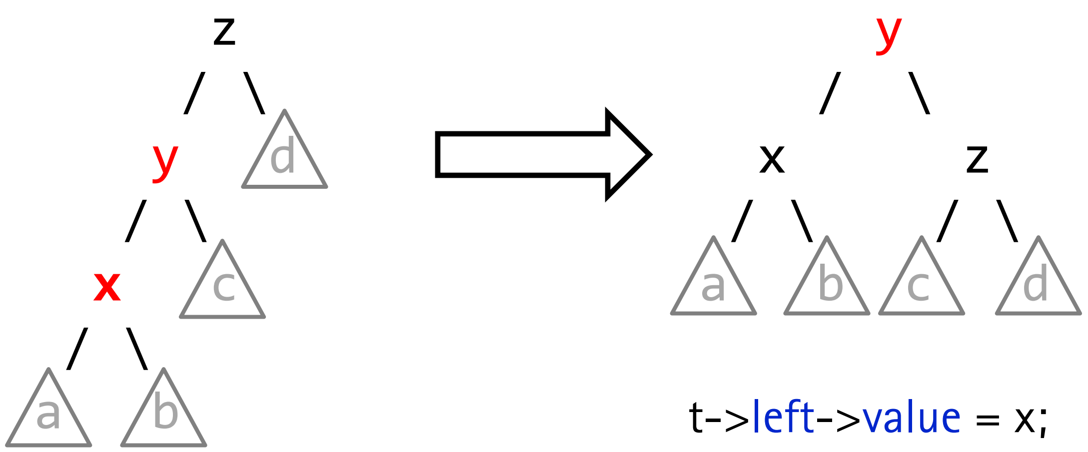
```
void balance(RBNode* t) {
    bool lRed = t->left != NULL && t->left->color == R;     // is left red?
    bool rRed = t->right != NULL && t->right->color == R;   // is right red?
    if (!lRed && !rRed) return;
    RBNode* ll = t->left == NULL ? NULL : t->left->left;
    RBNode* lr = t->left == NULL ? NULL : t->left->right;
    RBNode* rl = t->right == NULL ? NULL : t->right->left;
    RBNode* rr = t->right == NULL ? NULL : t->right->right;
    bool llRed = ll != NULL && ll->color == R;
    bool lrRed = lr != NULL && lr->color == R;
    bool rlRed = rl != NULL && rl->color == R;
    bool rrRed = rr != NULL && rr->color == R;
    RBNode *a, *b, *c, *d;
    int x, y, z;
```

```
       v
     /   \
    l     r
   / \   / \
  ll lr rl rr
```

Determine which nodes are red

Red-Black Trees in C: Balancing (case 1 of 4)

```
if (lRed && llRed) {
        a = ll->left;
        b = ll->right;
        c = lr;
        d = t->right;
        x = ll->value;
        y = t->left->value;
        z = t->value;
        t->right = ll;
} ...
```

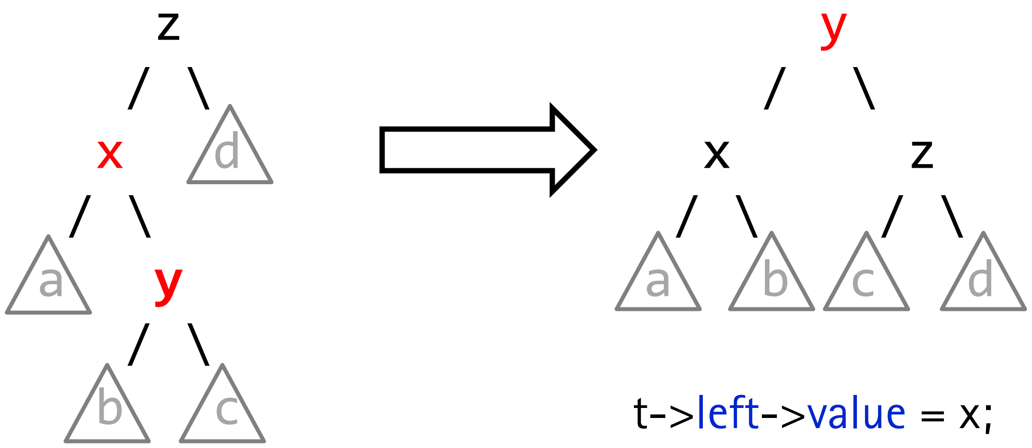


Identify the pieces as a, b, c, d, x, y, z

```
t->left->value = x;
t->value = y;
t->right->value = z;
t->left->left = a;
t->left->right = b;
t->right->left = c;
t->right->right = d;
t->left->color = B;
t->color = R;
t->right->color = B;
```

Red-Black Trees in C: Balancing (case 2 of 4)

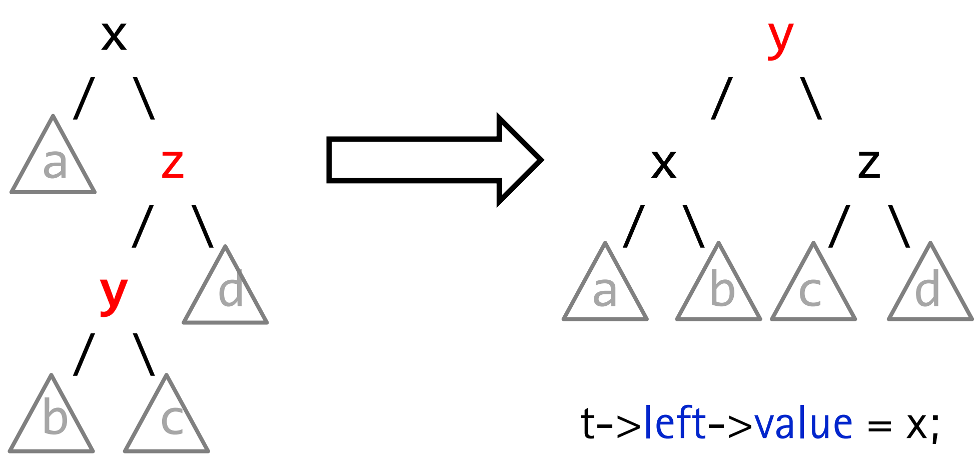
```
} else if (lRed && lrRed) {
        a = ll;
        b = lr->left;
        c = lr->right;
        d = t->right;
        x = t->left->value;
        y = lr->value;
        z = t->value;
        t->right = lr;
    } ...
```




```
t->left->value = x;
t->value = y;
t->right->value = z;
t->left->left = a;
t->left->right = b;
t->right->left = c;
t->right->right = d;
t->left->color = B;
t->color = R;
t->right->color = B;
```

Red-Black Trees in C: Balancing (case 3 of 4)

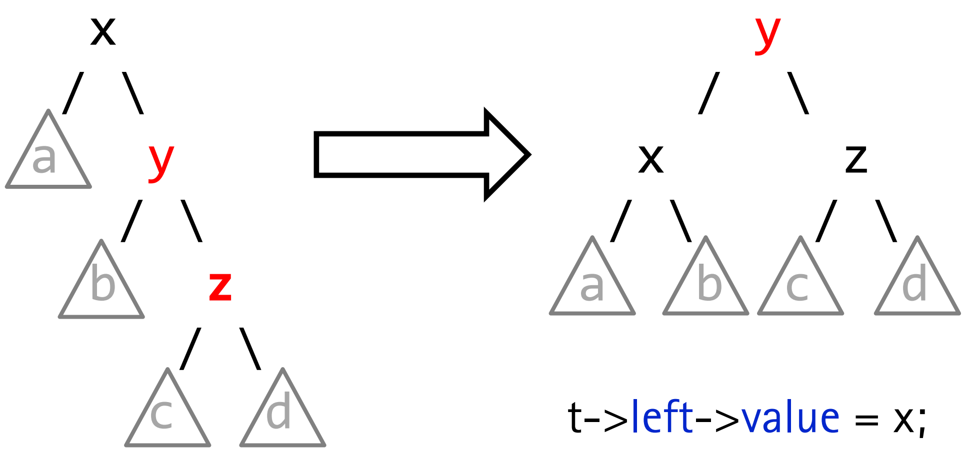
```
} else if (rRed && rlRed) {
        a = t->left;
        b = rl->left;
        c = rl->right;
        d = rr;
        x = t->value;
        y = rl->value;
        z = t->right->value;
        t->left = rl;
} …
```




```
t->left->value = x;
t->value = y;
t->right->value = z;
t->left->left = a;
t->left->right = b;
t->right->left = c;
t->right->right = d;
t->left->color = B;
t->color = R;
t->right->color = B;
```

Red-Black Trees in C: Balancing (case 4 of 4)

```
} else if (rRed && rrRed) {
        a = t->left;
        b = rl;
        c = rr->left;
        d = rr->right;
        x = t->value;
        y = t->right->value;
        z = rr->value;
        t->left = rr;
} else {
        return; // no need to rebalance
}
...
```




```
t->left->value = x;
t->value = y;
t->right->value = z;
t->left->left = a;
t->left->right = b;
t->right->left = c;
t->right->right = d;
t->left->color = B;
t->color = R;
t->right->color = B;
```

Red-Black Trees in C: Balancing

```
    t->left->value = x;
    t->value = y;
    t->right->value = z;
    t->left->left = a;
    t->left->right = b;
    t->right->left = c;
    t->right->right = d;
    t->left->color = B;
    t->color = R;
    t->right->color = B;
}
```

Why do Java programmers wear glasses?

Because they don't C#.

https://www.reddit.com/r/Jokes/comments/1k0tv1/why_do_java_programmers_wear_glasses/