viettel
Theo cách của bạn

TREE DATA STRUCTURE

Presenter: Trinh Cao Cuong

4/2024

TABLE OF CONTENTS

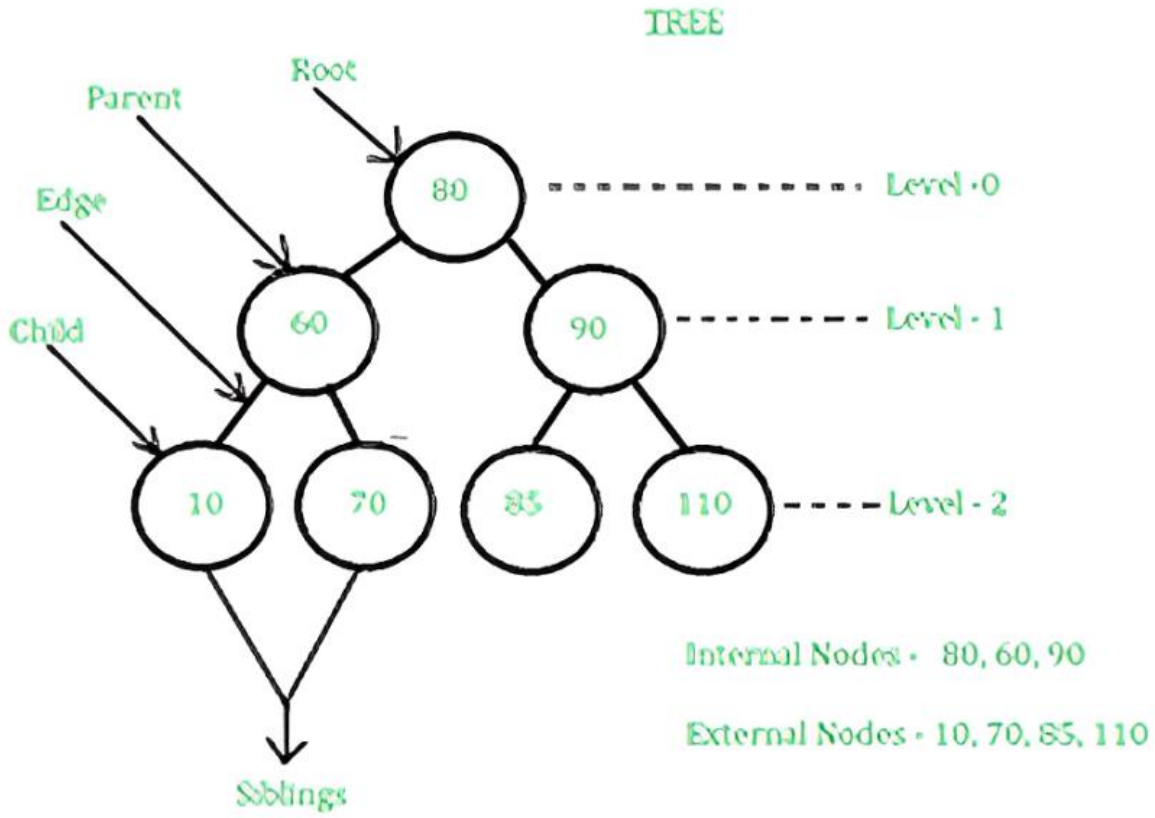
viettel
Theo cách của bạn

1. DEFINITION

viettel
Theo cách của bạn

DEFINITIONS : WHAT IS TREE DATA STRUCTURE ?

Tree is a non-linear data structure. It consists of nodes and edges. A tree **represents data in a hierarchical organization**. It is a special type of connected graph without any cycle or circuit.



DEFINITIONS : ADVANTAGES

- **Efficient searching:** efficient for searching and retrieving data. The time complexity of searching in a tree is typically O(log n)
- **Flexible size:** can grow or shrink dynamically depending on the number of nodes that are added or removed.
- **Easy to traverse:** Traversing a tree is a simple operation, easy to retrieve and process data from a tree structure.
- **Easy to maintain**: Trees are easy to maintain because they enforce a strict hierarchy and relationship between nodes. This makes it easy to add, remove, or modify nodes without affecting the rest of the tree structure.
- **Natural organization:** natural hierarchical organization useful for representing things like file systems, organizational structures, and taxonomies.
- **Fast insertion and deletion:** Inserting and deleting nodes in a tree can be done in O(log n) time

DEFINITIONS : DISADVANTAGES

- **Memory overhead:** require a significant amount of memory to store, especially if they are very large.
- **Imbalanced trees:** non balanced tree can result in uneven search times
- **Complexity:** can be difficult to understand and implement correctly.
- **Limited flexibility:** While trees are flexible in terms of size and structure, they are not as flexible as other data structures like hash tables.
- **Inefficient for certain operations:** inefficient for other operations like sorting or grouping.

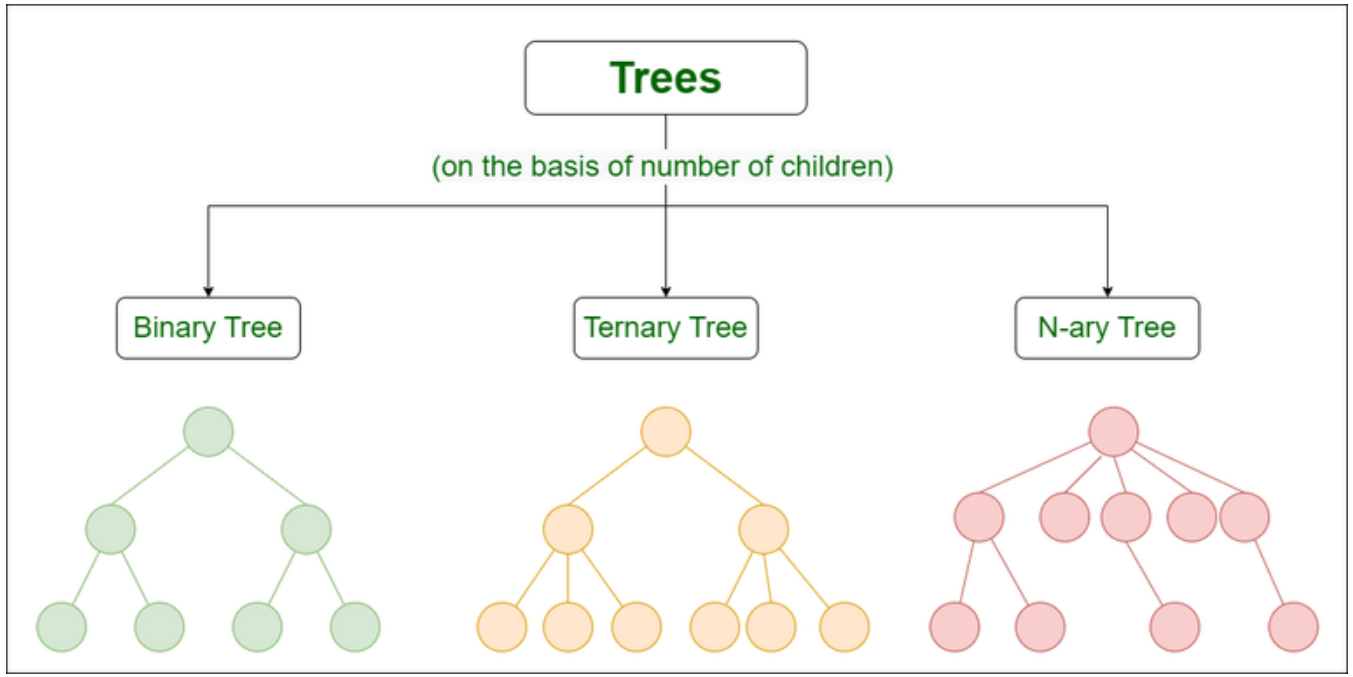
DEFINITIONS : DISADVANTAGES

- **Memory overhead:** require a significant amount of memory to store, especially if they are very large.
- **Imbalanced trees:** non balanced tree can result in uneven search times
- **Complexity:** can be difficult to understand and implement correctly.
- **Limited flexibility:** While trees are flexible in terms of size and structure, they are not as flexible as other data structures like hash tables.
- **Inefficient for certain operations:** inefficient for other operations like sorting or grouping.

DEFINITIONS : TYPES OF TREE

Types of Trees based on the number of children:

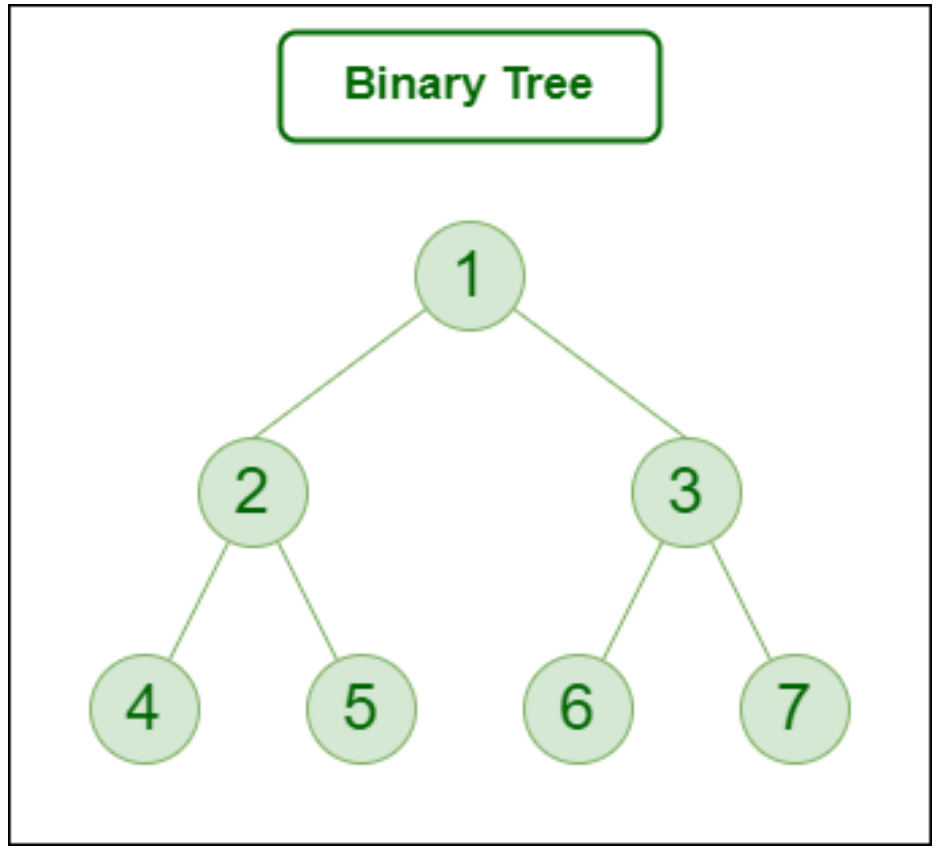
- **Binary tree:** each node of this tree has only 2 children
- **Ternary tree:** each node has at most three child nodes, usually distinguished as “left”, “mid” and “right”.
- **N-Array tree (Generic tree):** a collection of nodes where each node is a data structure that consists of records and a list of references to its children (not duplicate references).



DEFINITIONS : TYPES OF TREE

Types of Trees based on the node's values:

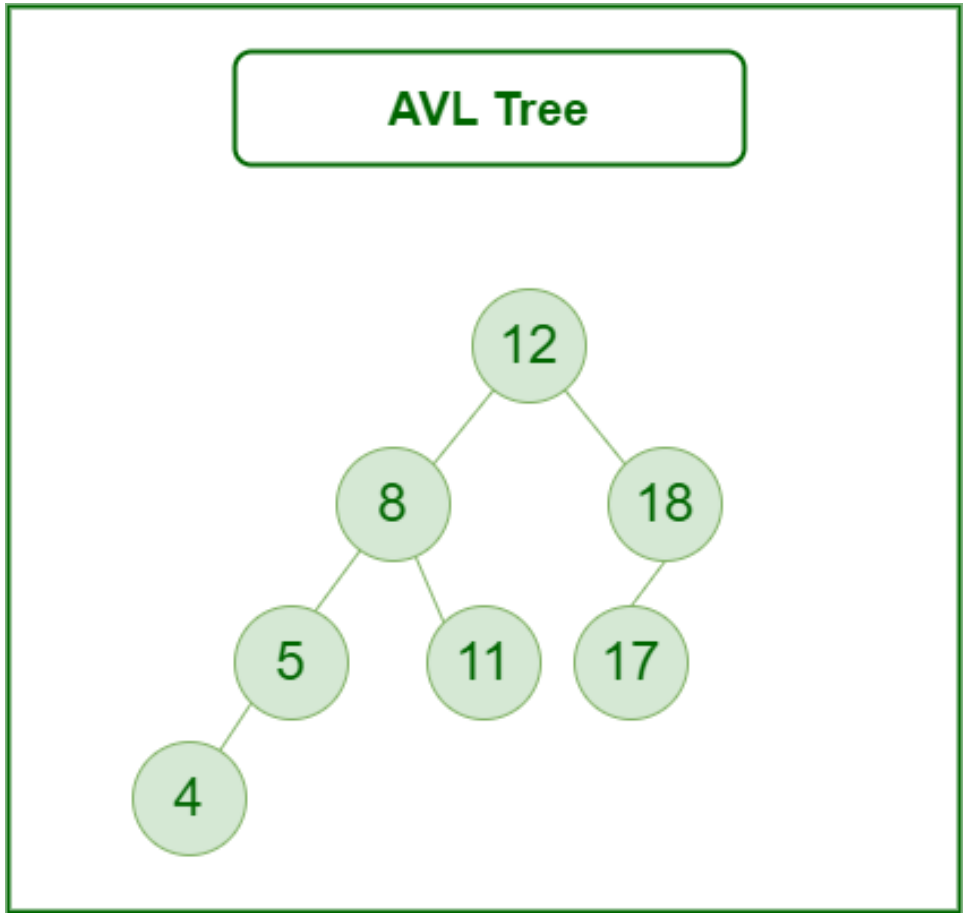
- **Binary search tree:** a node-based binary tree data structure that:
  + The left subtree of a node contains only nodes with keys lesser than the node's key.
  + The right subtree of a node contains only nodes with keys greater than the node's key.
  + The left and right subtree each must also be a binary search tree.
- **AVL tree**
- **Red-Black tree**
- **B and B+ tree**
- **Segment tree**



DEFINITIONS : TYPES OF TREE

Types of Trees based on the node's values:

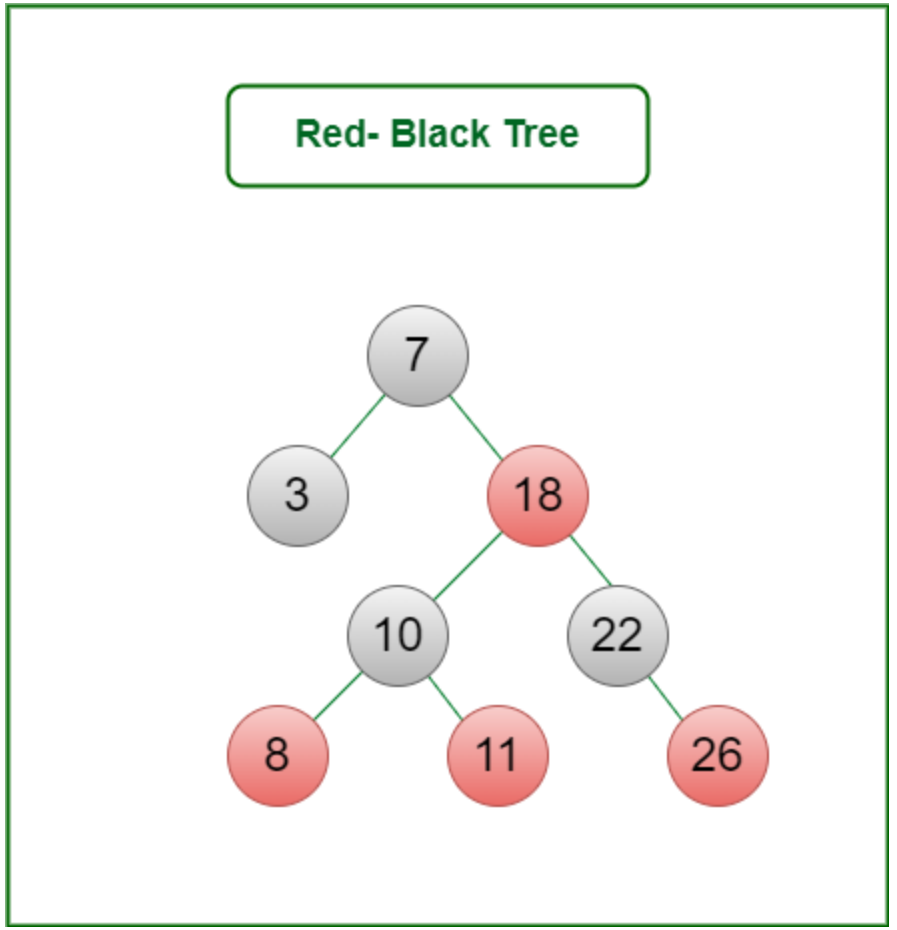
- **Binary search tree**
- **AVL tree:** a self-balancing Binary Search Tree where the difference between heights of left and right subtrees for any node cannot be more than one.
- **Red-Black tree**
- **B and B+ tree**
- **Segment tree**



DEFINITIONS : TYPES OF TREE

Types of Trees based on the node's values:

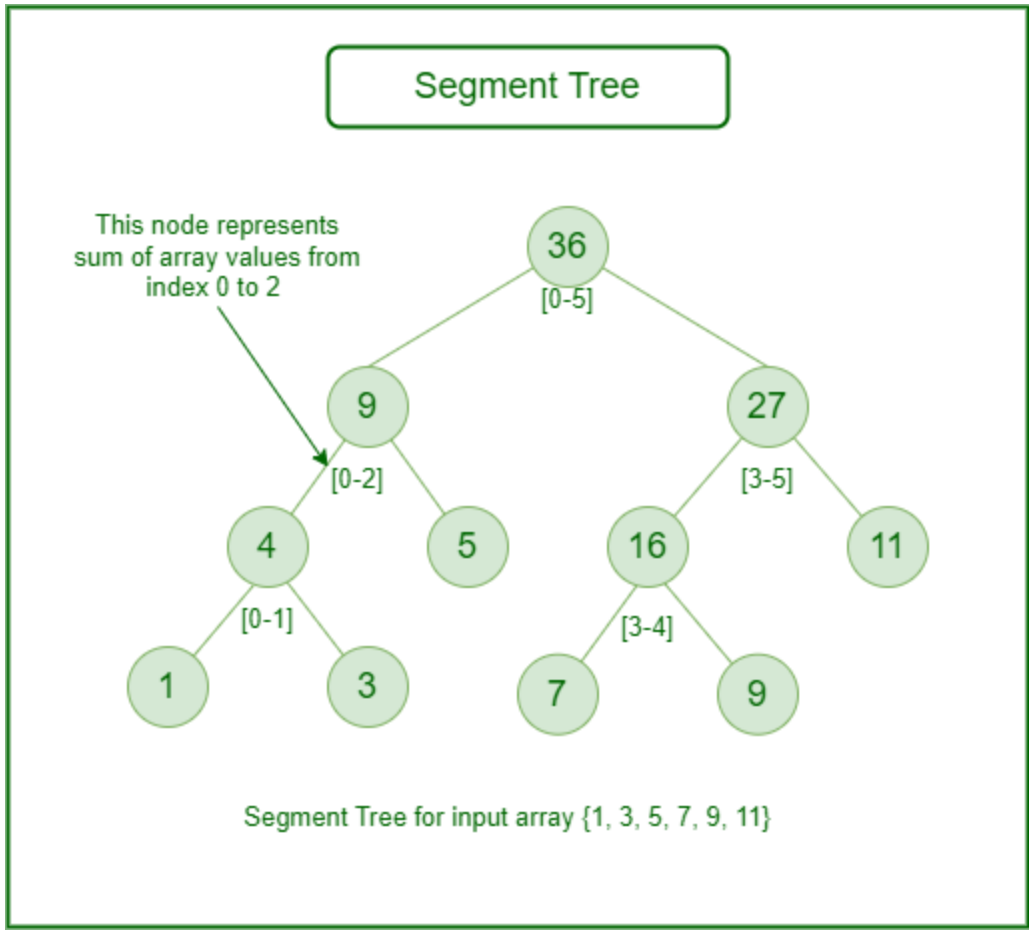
- **Binary search tree**
- **AVL tree**
- **Red-Black tree:** a kind of self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the color (red or black). These colors are used to ensure that the tree remains balanced during insertions and deletions.
- **B and B+ tree**
- **Segment tree**



DEFINITIONS : TYPES OF TREE

Types of Trees based on the node's values:

- **Binary search tree**
- **AVL tree**
- **Red-Black tree**
- **B and B+ tree**
- **Segment tree:** known as a statistic tree, is a tree data structure used for storing information about intervals, or segments. It allows querying which of the stored segments contain a given point. It is a static structure that cannot be modified once it's built.



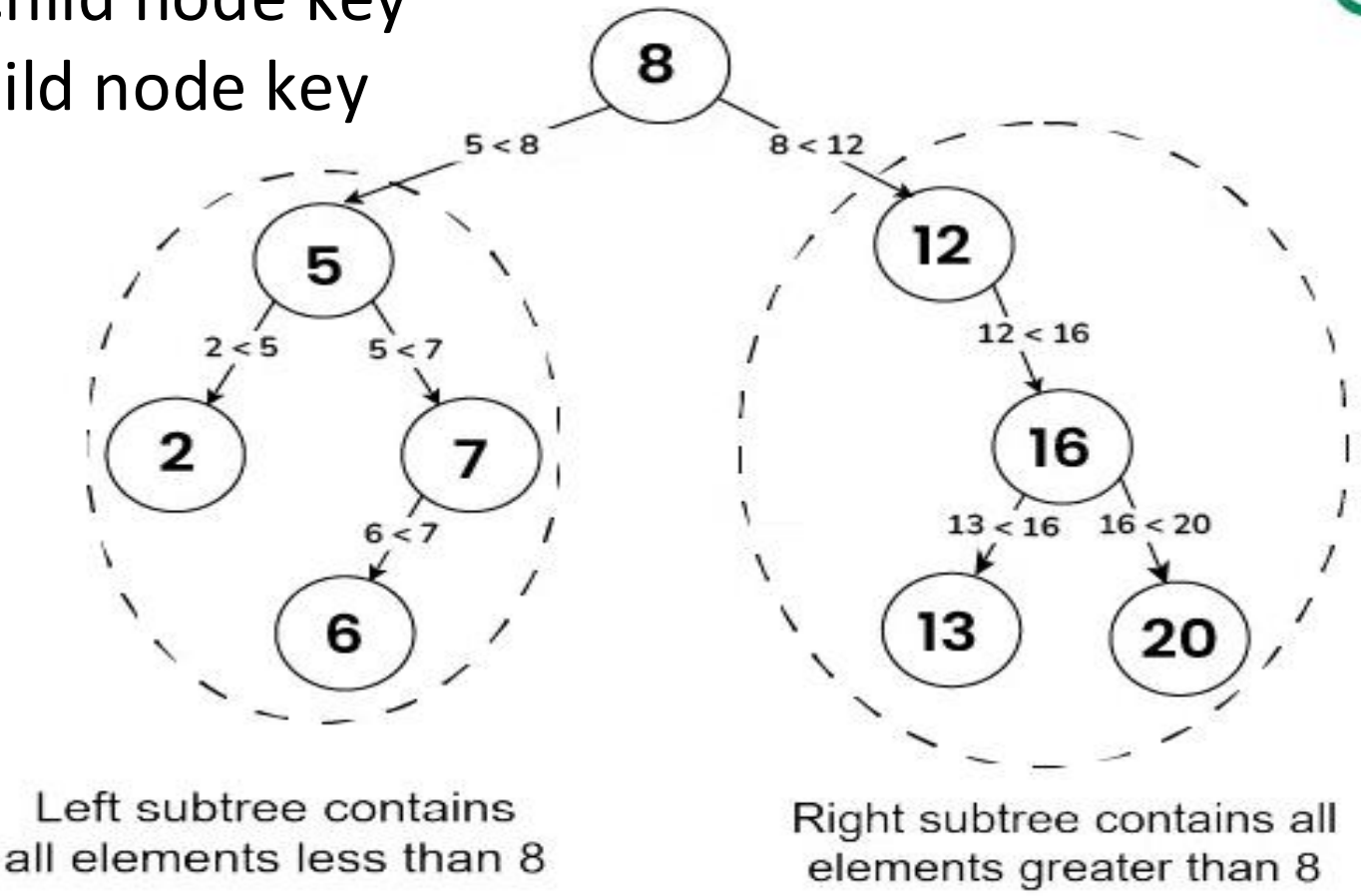
2. BINARY SEARCH TREE

BINARY SEARCH TREE : DEFINITION

A Binary Search Tree is a hierarchical data structure with:

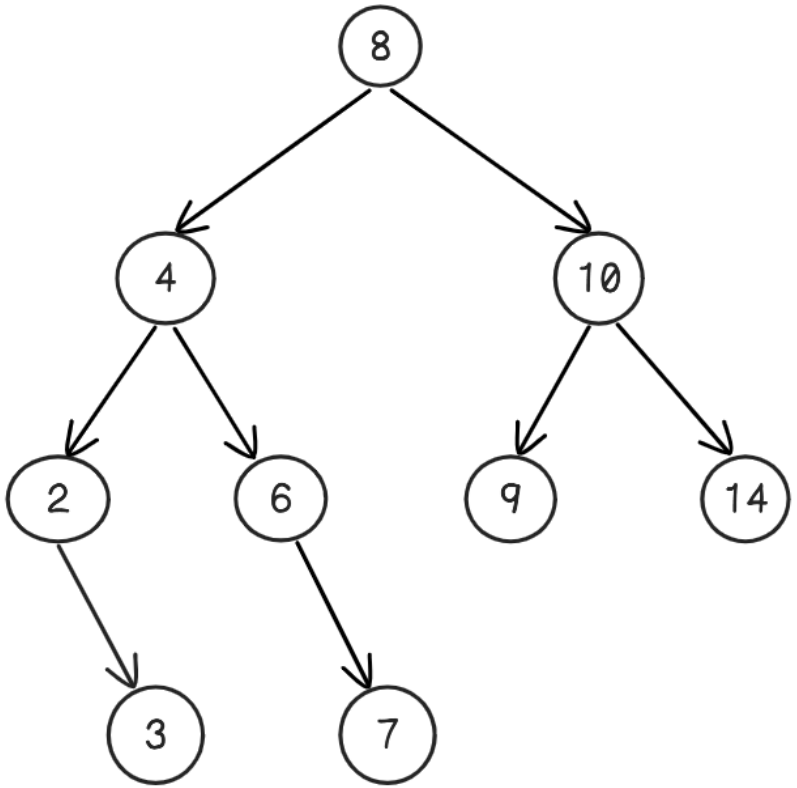
- Each node has at most 2 child (Left child & Right child)
- Each node has key and data
- Right child node key > Parent child node key
- Left child node key < Parent child node key

Handle same value node:
Left or right or reject but be consistent



BINARY SEARCH TREE : DEFINITION

```
290  int main()
291  {
292      /* Create tree */
293      struct tree *tree = (struct tree *)calloc(1, sizeof(struct tree));
294      /* Let us create following BST
295                 8
296               /     \
297              4       10
298             /  \    /  \
299            2    6  9   14
300             \    \
301              3    7
302      */
303      node_insert(tree, 8);
304      node_insert(tree, 4);
305      node_insert(tree, 10);
306      node_insert(tree, 2);
307      node_insert(tree, 3);
308      node_insert(tree, 6);
309      node_insert(tree, 7);
310      node_insert(tree, 9);
311      node_insert(tree, 14);
312
313      /* Print tree inorder */
314      tree_print_inorder(tree);
315
316      node_delete(tree, 4);
317      tree_print_inorder(tree);
318
319      return 0;
320  }
321
```



BINARY SEARCH TREE : BASIC OPERATION

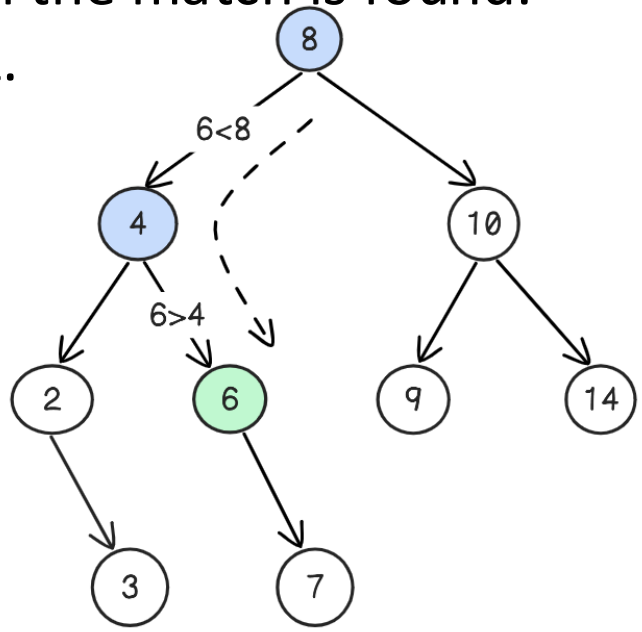
Basic operation:

- Seach node
- Insert node
- Delete node
- Tree traversal

Searching in BST means to locate a specific node in the data structure.
The steps of searching a node in Binary Search tree are:

1. Compare the element to be searched with the root node.
   - If matched then return the node's location.
   - If not matched, then check it is smaller than the root node then move to the left subtree.
   - If it is larger than the root node then move to the right subtree.
2. Repeat the above procedure recursively until the match is found.
3. If the element is not found then return NULL.

Complexity: Olog(n)

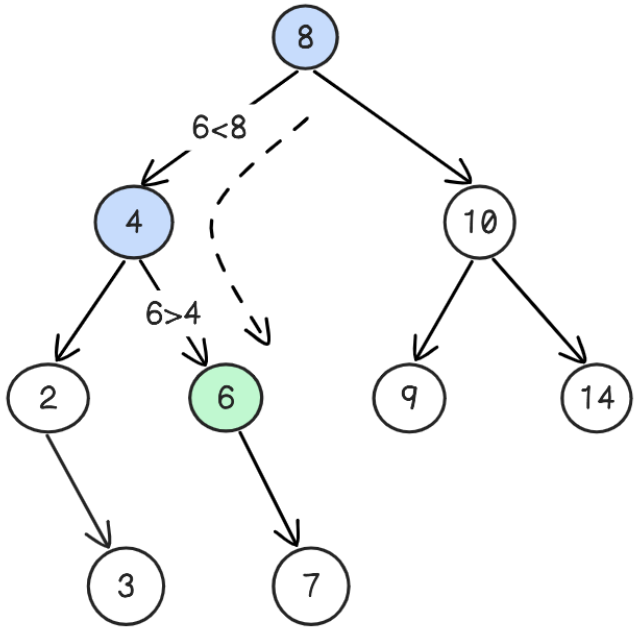


BINARY SEARCH TREE : BASIC OPERATION

```c
/* Insert new node to the tree */
struct node *node_search(struct tree *tree, int key)
{
    if (NULL == tree)
        return NULL;

    struct node *node = tree->top;

    /* If the tree is empty, couldn't find */
    if (NULL == node)
        return NULL;

    /* Otherwise, recur down the tree */
    while (node)
    {
        if (key > node->key)
            node = node->right;
        else if (key < node->key)
            node = node->left;
        else if (key == node->key)
            return node;
    }

    /* Find nothing or something wrong */
    return NULL;
}
```

```c
struct node *node = node_search(tree, 4);
printf("Node %d, Parent: %d, Left: %d, Right: %d",  node->key,
                                                    node->parent->key,
                                                    node->left->key,
                                                    node->right->key);
```

```
PS C:\Users\QuyetDH\Documents\data_struct> gcc .\main.c -o main;.\
main
Node 4, Parent: 8, Left: 2, Right: 6
```

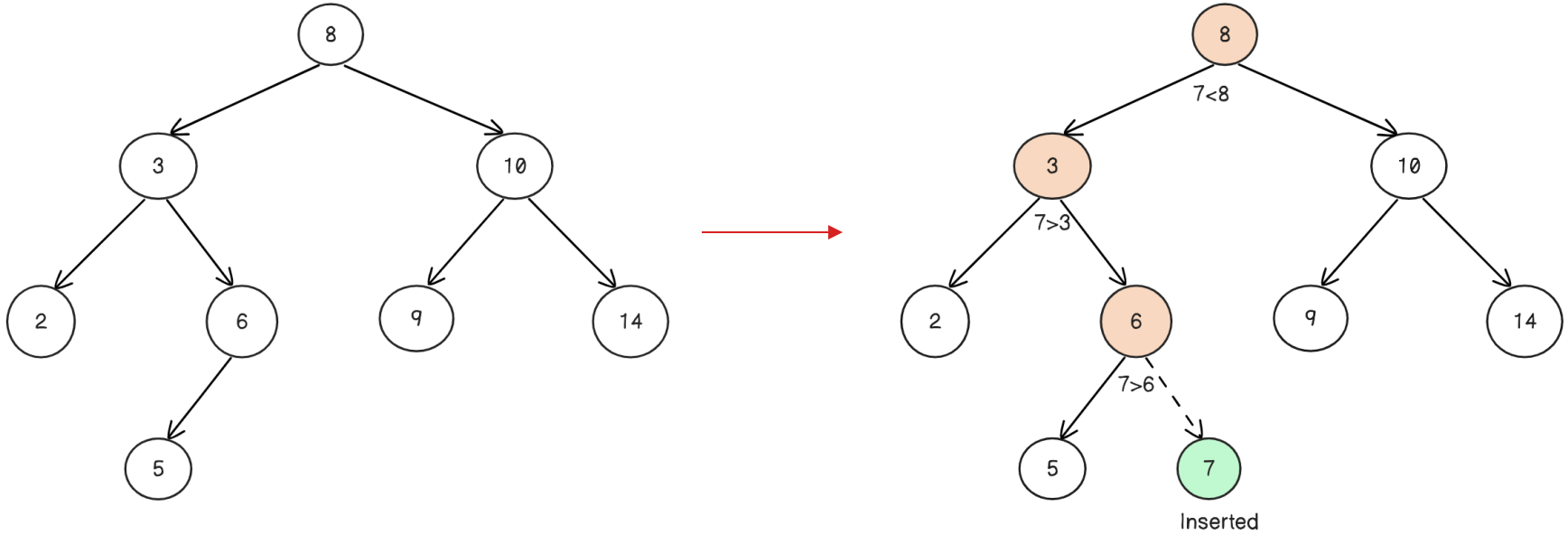


BINARY SEARCH TREE : BASIC OPERATION

Basic operation:

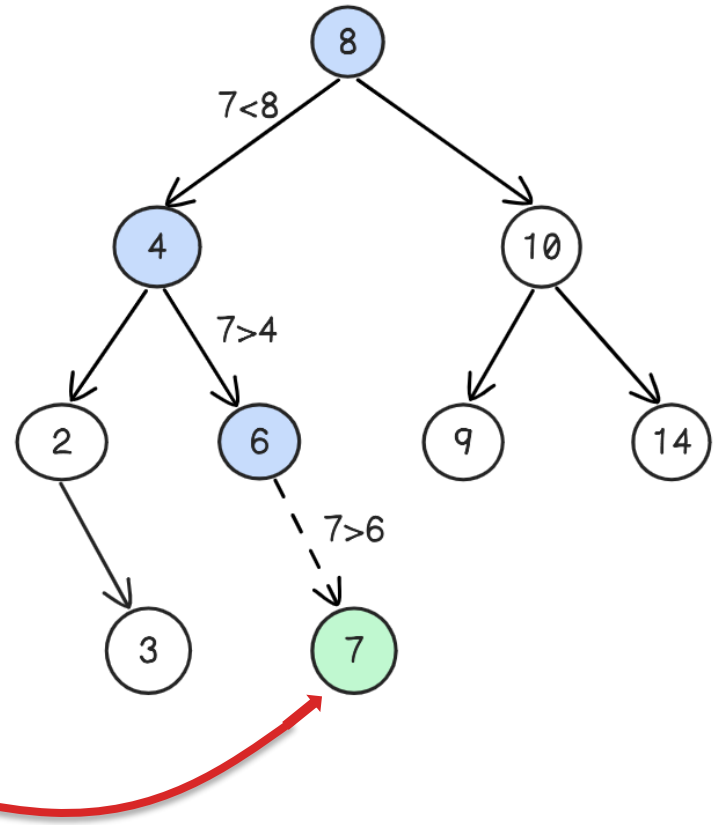
- Seach node
- Insert node
- Delete node
- Tree traversal

A new key is always inserted at the leaf.
Start searching a key from the root till a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node.



BINARY SEARCH TREE : BASIC OPERATION

```
284     /* Let us create following BST
285                 8
286               /   \
287              4     10
288            /  \   /  \
289           2    6 9   14
290            \    \
291             3    7
292     */
293     node_insert(tree, 8);
294     node_insert(tree, 4);
295     node_insert(tree, 10);
296     node_insert(tree, 2);
297     node_insert(tree, 3);
298     node_insert(tree, 6);
299     node_insert(tree, 9);
300     node_insert(tree, 14);
301     node_insert(tree, 7);
302
303     tree_print_inorder(tree);
```




```
PS C:\Users\QuyetDH\Documents\data_struct> gcc .\main.c -o main;.\ma
in
2 3 4 6 7 8 9 10 14
```

```
36  /* Insert new node to the tree */
37  struct node *node_insert(struct tree *tree, int key)
38  {
39      if (NULL == tree)
40          return NULL;
41
42      struct node *node = tree->top;
43      struct node *new = node_create(key);
44
45      /* If the tree is empty, assign to top node */
46      if (NULL == node)
47      {
48          tree->top = new;
49          return new;
50      }
51
52      /* Otherwise, recur down the tree */
53      while (node && node->key != new->key)
54      {
55          if (node->key < new->key)
56          {
57              if (NULL == node->right)
58              {
59                  node->right = new;
60                  new->parent = node;
61                  return new;
62              }
63              else
64                  node = node->right;
65          }
66          else if (NULL == node->left)
67          {
68              node->left = new;
69              new->parent = node;
70              return new;
71          }
72          else
73              node = node->left;
74      }
75
76      /* New has already exist or something wrong */
77      free(new);
78      return node;
79  }
```

BINARY SEARCH TREE : BASIC OPERATION

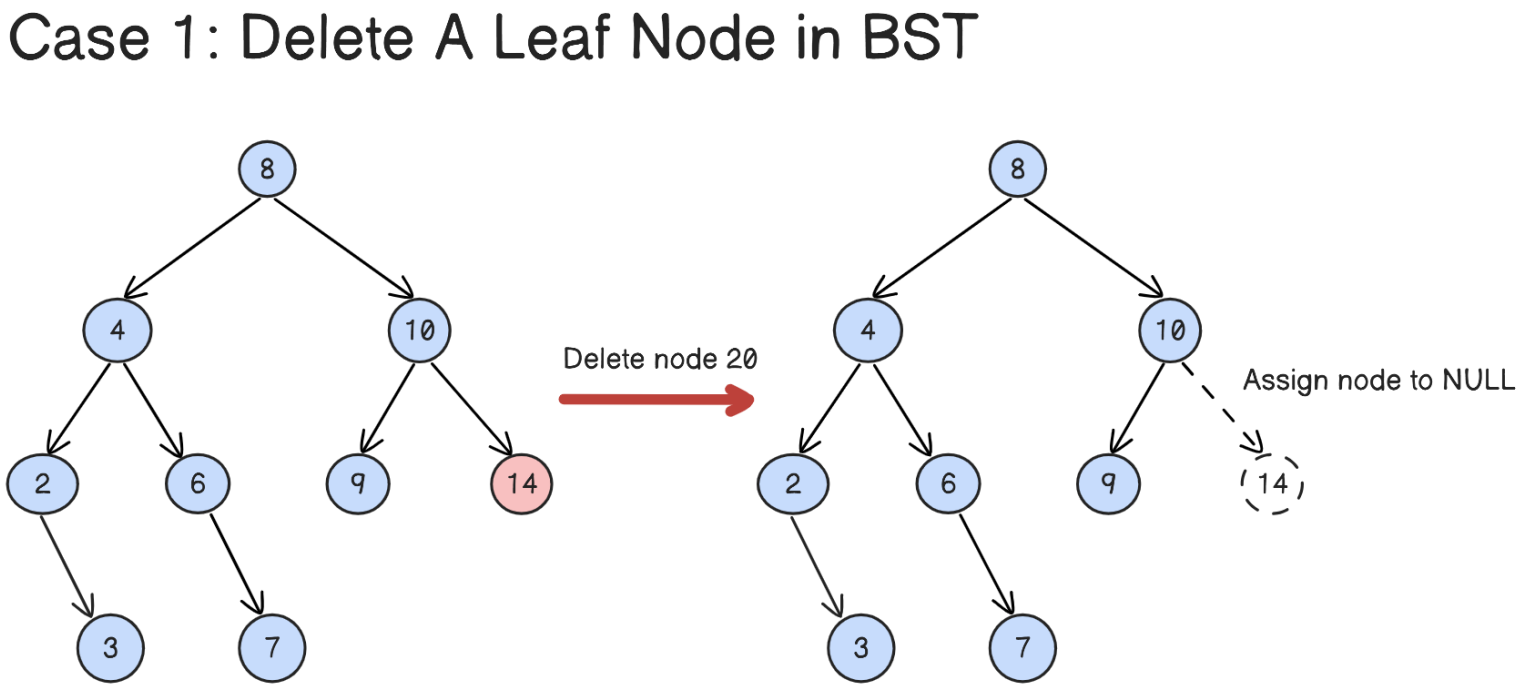
Basic operation:

- Seach node
- Insert node
- Delete node
- Tree traversal

It is used to delete a node with specific key from the BST and return the new BST.

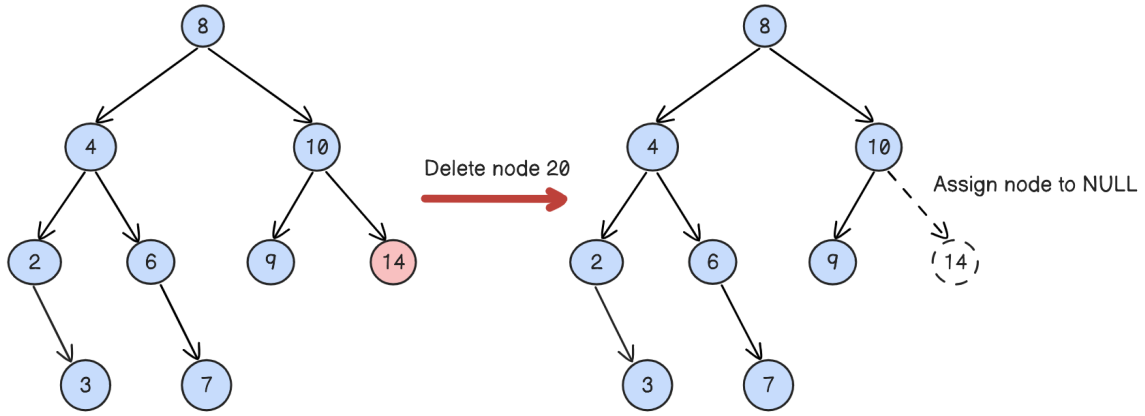
Different scenarios for deleting the node:

Case 1: Just simply remove the leaf node



BINARY SEARCH TREE : BASIC OPERATION

Case 1: Delete A Leaf Node in BST




```
303     /* Print tree inorder */
304     tree_print_inorder(tree);
305
306     node_delete(tree, 14);
307     struct node *node = node_search(tree, 10);
308     printf("Node %d, Parent %d, Left %d, Right %d\n", node->key,
309                                                       node->parent? node->parent->key: -1,
310                                                       node->left? node->left->key: -1,
311                                                       node->right? node->right->key: -1);
312
313     /* Print tree inorder */
314     tree_print_inorder(tree);
```

```
PS C:\Users\QuyetDH\Documents\data_struct> gcc .\main.c -o main;.\ma
in
2 3 4 6 7 8 9 10 14
Node 10, Parent 8, Left 9, Right -1
2 3 4 6 7 8 9 10
```

```
162  /* Delete node by key */
163  struct tree *node_delete(struct tree *tree, int key)
164  {
165      if (NULL == tree || NULL == tree->top)
166          return NULL;
167
168      /* Find node to delete */
169      struct node *node = node_search(tree, key);
170      if (NULL == node)
171          return NULL;
172
173      /* Delete node has no child */
174      if (NULL == node->left && NULL == node->right)
175      {
176          if (node == tree->top)
177              tree->top = NULL;
178
179          if (node == node->parent->left)
180              node->parent->left = NULL;
181
182          if (node == node->parent->right)
183              node->parent->right = NULL;
184
185          free(node);
186          return tree;
187      }
188
189      /* Delete node with one child */
190  >   if (NULL == node->left || NULL == node->right)…
232
233      /* Delete node with 2 child */
234  >   if (node->left && node->right)…
273
274      return NULL;
```

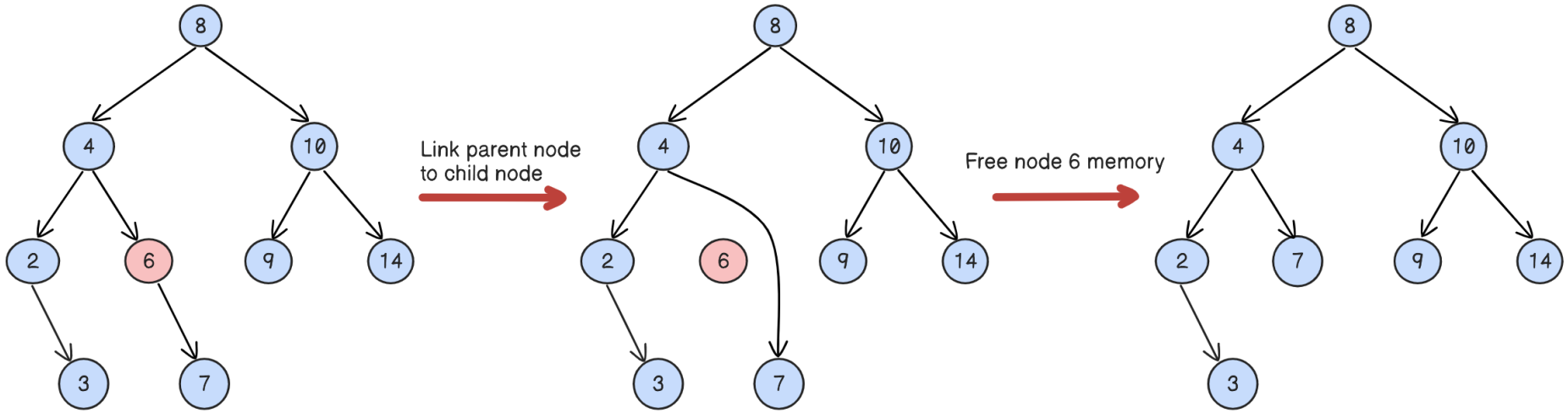
BINARY SEARCH TREE : BASIC OPERATION

Basic operation:

- Seach node
- Insert node
- Delete node
- Tree traversal

Different scenarios for deleting the node:

Case 2: Link child node to parent node then remove current node



BINARY SEARCH TREE : BASIC OPERATION

```
303     /* Print tree inorder */
304     tree_print_inorder(tree);
305
306     node_delete(tree, 6);
307     struct node *node = node_search(tree, 7);
308     printf("Node %d, Parent %d, Left %d, Right %d\n", node->key,
309                                                       node->parent? node->parent->key: -1,
310                                                       node->left? node->left->key: -1,
311                                                       node->right? node->right->key: -1);
312
313     /* Print tree inorder */
314     tree_print_inorder(tree);
```

```
PS C:\Users\QuyetDH\Documents\data_struct> gcc .\main.c -o main;.\ma
in
2 3 4 6 7 8 9 10 14
Node 7, Parent 4, Left -1, Right -1
2 3 4 7 8 9 10 14
```

```
162 /* Delete node by key */
163 struct tree *node_delete(struct tree *tree, int key)
164 {
165     if (NULL == tree || NULL == tree->top)
166         return NULL;
167
168     /* Find node to delete */
169     struct node *node = node_search(tree, key);
170     if (NULL == node)
171         return NULL;
172
173     /* Delete node has no child */
174 >   if (NULL == node->left && NULL == node->right) ···
188
```

```
189     /* Delete node with one child */
190     if (NULL == node->left || NULL == node->right)
191     {
192         if (node->left)
193         {
194             if (node == tree->top)
195             {
196                 tree->top = node->left;
197                 node->left->parent = NULL;
198             }
199             else
200                 node->left->parent = node->parent;
201
202             if (node->parent && node == node->parent->left)
203                 node->parent->left = node->left;
204
205             if (node->parent && node == node->parent->right)
206                 node->parent->right = node->left;
207
208             free(node);
209             return tree;
210         }
211
212         if (node->right)
213         {
214             if (node == tree->top)
215             {
216                 tree->top = node->right;
217                 node->right->parent = NULL;
218             }
219             else
220                 node->right->parent = node->parent;
221
222             if (node == node->parent->left)
223                 node->parent->left = node->right;
224
225             if (node == node->parent->right)
226                 node->parent->right = node->right;
227
228             free(node);
229             return tree;
230         }
231     }
232
233     /* Delete node with 2 child */
234 >   if (node->left && node->right) ···
273
274     return NULL;
275 }
```

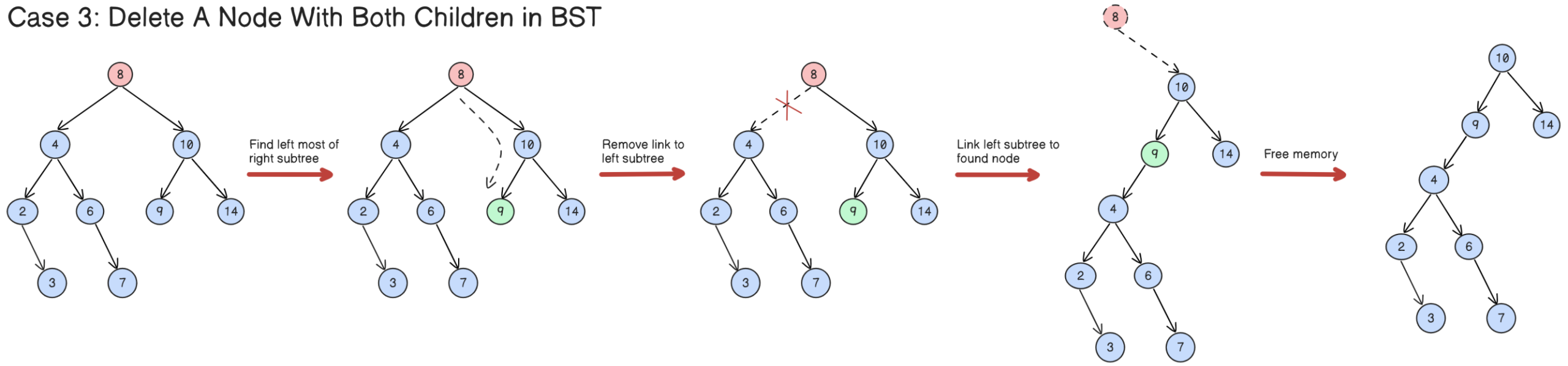
BINARY SEARCH TREE : BASIC OPERATION

Basic operation:

- Seach node
- Insert node
- Delete node
- Tree traversal

Different scenarios for deleting the node:

Case 3: Using minimum node in right subtree to hold the left sub tree



BINARY SEARCH TREE : BASIC OPERATION

```
303     /* Print tree inorder */
304     tree_print_inorder(tree);
305
306     node_delete(tree, 8);
307     struct node *node = node_search(tree, 9);
308     printf("Node %d, Parent %d, Left %d, Right %d\n", node->key,
309                                                       node->parent? node->parent->key: -1,
310                                                       node->left? node->left->key: -1,
311                                                       node->right? node->right->key: -1);
312
313     /* Print tree inorder */
314     tree_print_inorder(tree);
```

```
PS C:\Users\QuyetDH\Documents\data_struct> gcc .\main.c -o main;.\ma
in
2 3 4 6 7 8 9 10 14
Node 9, Parent 10, Left 4, Right -1
2 3 4 6 7 9 10 14
```

```
162 /* Delete node by key */
163 struct tree *node_delete(struct tree *tree, int key)
164 {
165     if (NULL == tree || NULL == tree->top)
166         return NULL;
167
168     /* Find node to delete */
169     struct node *node = node_search(tree, key);
170     if (NULL == node)
171         return NULL;
172
173     /* Delete node has no child */
174 >   if (NULL == node->left && NULL == node->right) ···
188
189     /* Delete node with one child */
190 >   if (NULL == node->left || NULL == node->right) ···
232
```

```
233     /* Delete node with 2 child */
234     if (node->left && node->right)
235     {
236         struct node *subtree = node->right;
237         struct node *successor_node = node->right;
238
239         while (successor_node->left)
240         {
241             successor_node = successor_node->left;
242         }
243
244         successor_node->left = node->left;
245         node->left->parent = successor_node;
246
247
248         if (NULL == subtree)
249             return NULL;
250
251         if (node == tree->top)
252         {
253             tree->top = subtree;
254             subtree->parent = NULL;
255             free(node);
256             return tree;
257         }
258         else if (node == node->parent->left)
259         {
260             node->parent->left = subtree;
261             subtree->parent = node->parent;
262             free(node);
263             return tree;
264         }
265         else if (node == node->parent->right)
266         {
267             node->parent->right = subtree;
268             subtree->parent = node->parent;
269             free(node);
270             return tree;
271         }
272     }
273
274     return NULL;
275 }
```

BINARY SEARCH TREE : BASIC OPERATION

Basic operation:

- Seach node
- Insert node
- Delete node
- Tree traversal

To travel through all node of the tree (for print, etc), it have three way traversal:

- Inorder traversal
- Preorder traversal
- Postorder traversal

BINARY SEARCH TREE : BASIC OPERATION

Basic operation:

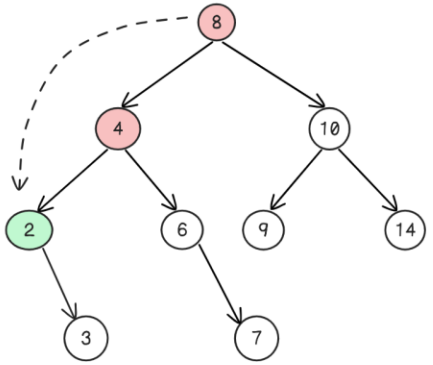
- Seach node
- Insert node
- Delete node
- Tree traversal

Inorder traversal:

- At first traverse left subtree then visit the root and then traverse the right subtree.
- Gives the values of the nodes in sorted order.

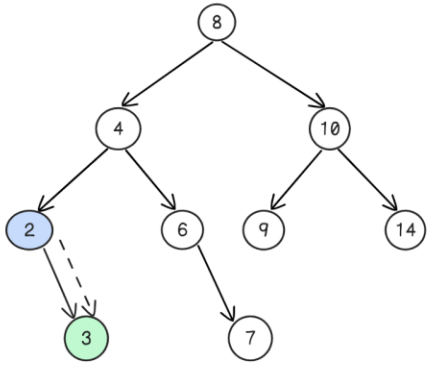
BINARY SEARCH TREE : BASIC OPERATION

Top node is node 8
Node 8 has left child, go to node 4
Node 4 has left child, go to node 2
Node 2 has no left child, return 2



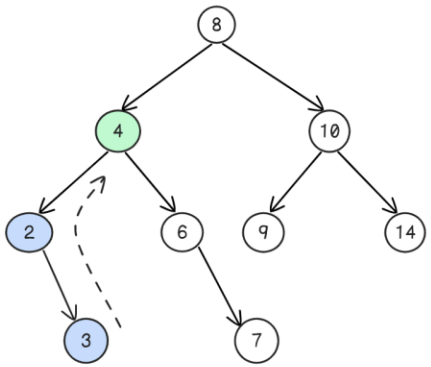
2

Node 2 has right child, go to node 3
Node 3 has no left child, return 3



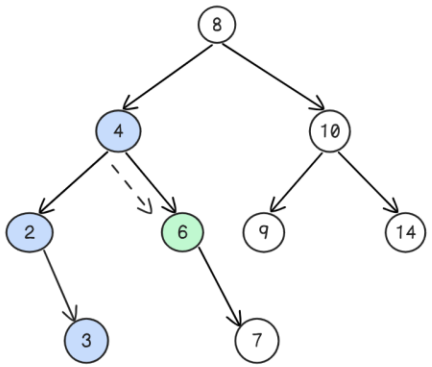
2,3

Node 3 has no child, back to node 2
Node 2 back from right child, back to node 4
Node 4 back from left child, return 4



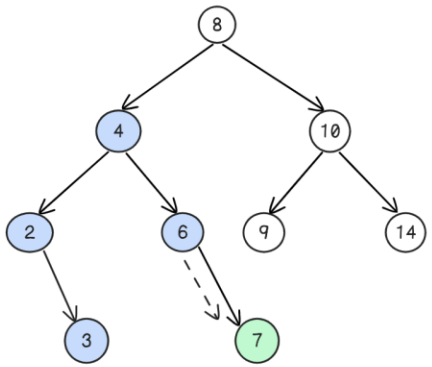
2,3,4

Node 4 has right child, go to node 6
Node 6 has no left child, return 6



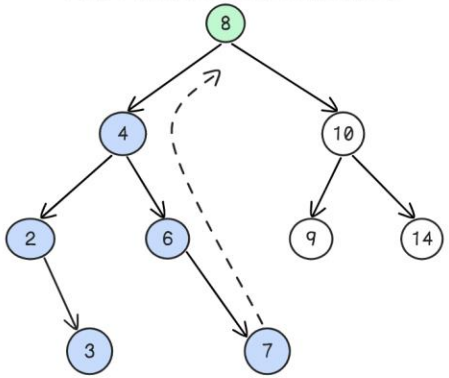
2,3,4,6

Node 6 has right child, go to node 7
Node 7 has no left child, return 7



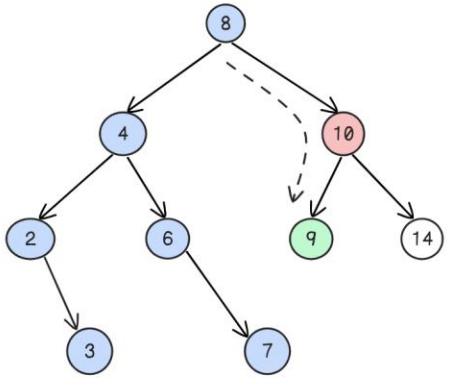
2,3,4,6,7

Node 7 has no right child, back to node 6
Node 6 is back from right child, back to node 4
Node 4 is back from right child, back to node 8
Node 8 is back from left child, return 8



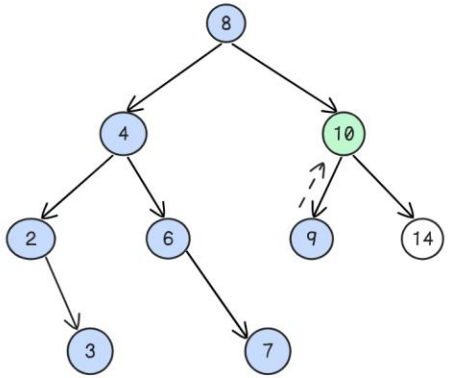
2,3,4,6,7,8

Node 8 has right child, go to node 10
Node 10 has left child, go to node 9
Node 9 has no left child, return 9



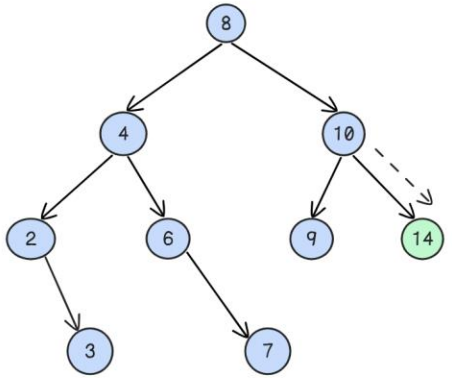
2,3,4,6,7,8,9

Node 9 has no right child, back to node 10
Node 10 is back from left child, return 10



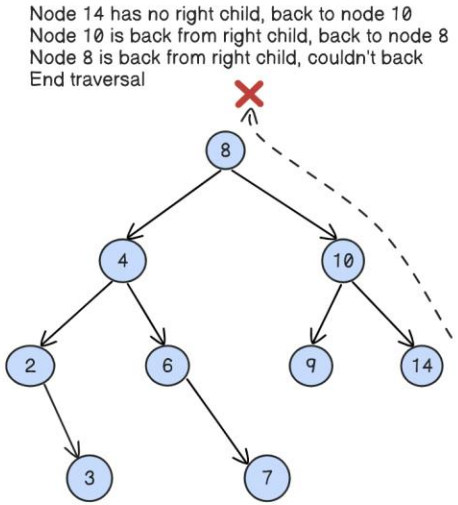
2,3,4,6,7,8,9,10

Node 10 has right child, go to node 14
Node 14 has no left child, return 14



2,3,4,6,7,8,9,10,14

Node 14 has no right child, back to node 10
Node 10 is back from right child, back to node 8
Node 8 is back from right child, couldn't back
End traversal



2,3,4,6,7,8,9,10,14

BINARY SEARCH TREE : BASIC OPERATION

```c
/* Traversal node inorder - left node right */
struct node *node_inorder_next(struct node *node)
{
    if (NULL == node)
        return NULL;

    /* Return left most of right child tree */
    if (node->right)
    {
        node = node->right;

        if (node->left)
        {
            while (node->left)
                node = node->left;

            return node;
        }
        return node;
    }

    /* Node has no right child, come back to parent */
    while (node->parent)
    {
        if (node->parent->left && node == node->parent->left)
            return node->parent;

        node = node->parent;
    }

    /* End tree traversal */
    return NULL;
}
```

```c
    /* Print tree inorder */
    tree_print_inorder(tree);
```

```
PS C:\Users\QuyetDH\Documents\data_struct> gcc .\main.c -o main;.\ma
in
2 3 4 6 7 8 9 10 14
```

```c
/* Print tree inorder traversal */
void tree_print_inorder(struct tree *tree)
{
    if (NULL == tree)
        printf("Non of tree");

    struct node *node = NULL;
    node = tree->top;

    while (node->left)
        node = node->left;

    while (node)
    {
        printf("%d ", node->key);
        node = node_inorder_next(node);
    }
    printf("\n");
}
```

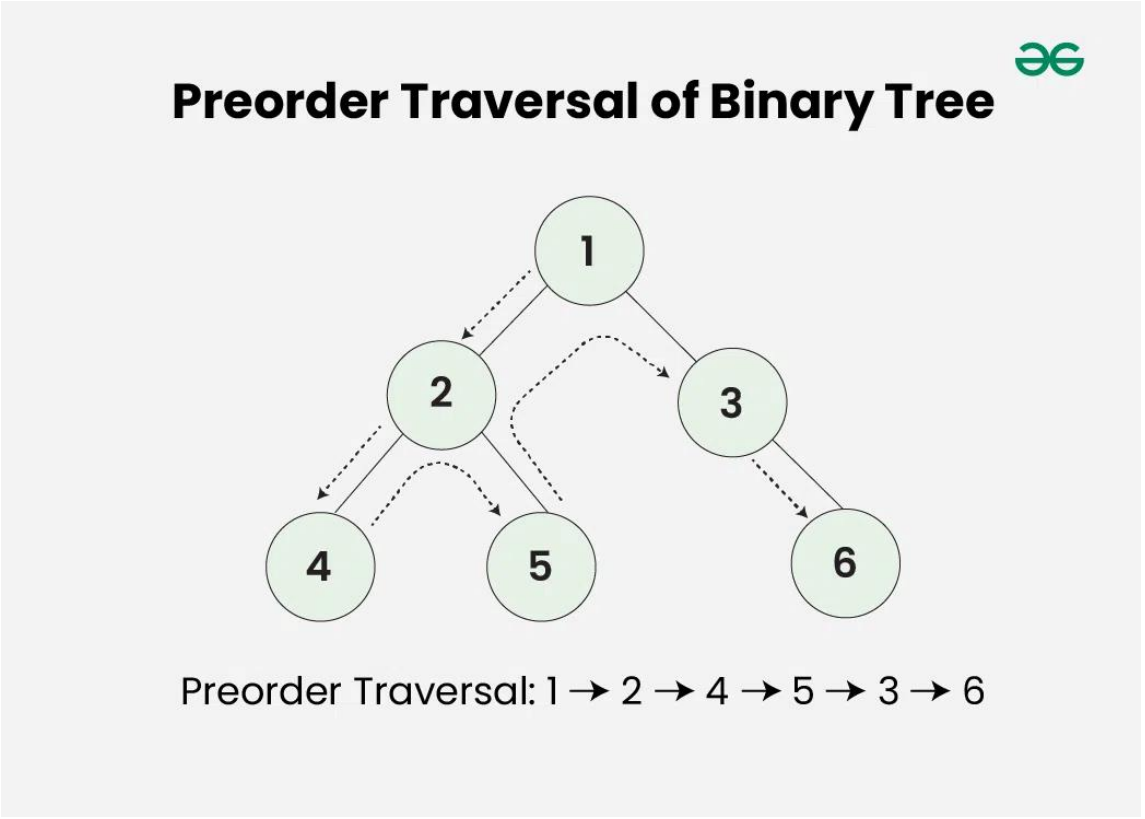
BINARY SEARCH TREE : BASIC OPERATION

Basic operation:

- Seach node
- Insert node
- Delete node
- Tree traversal

Preorder traversal:

- Used to create a copy of the tree.
- Used to get prefix expressions on an expression tree.



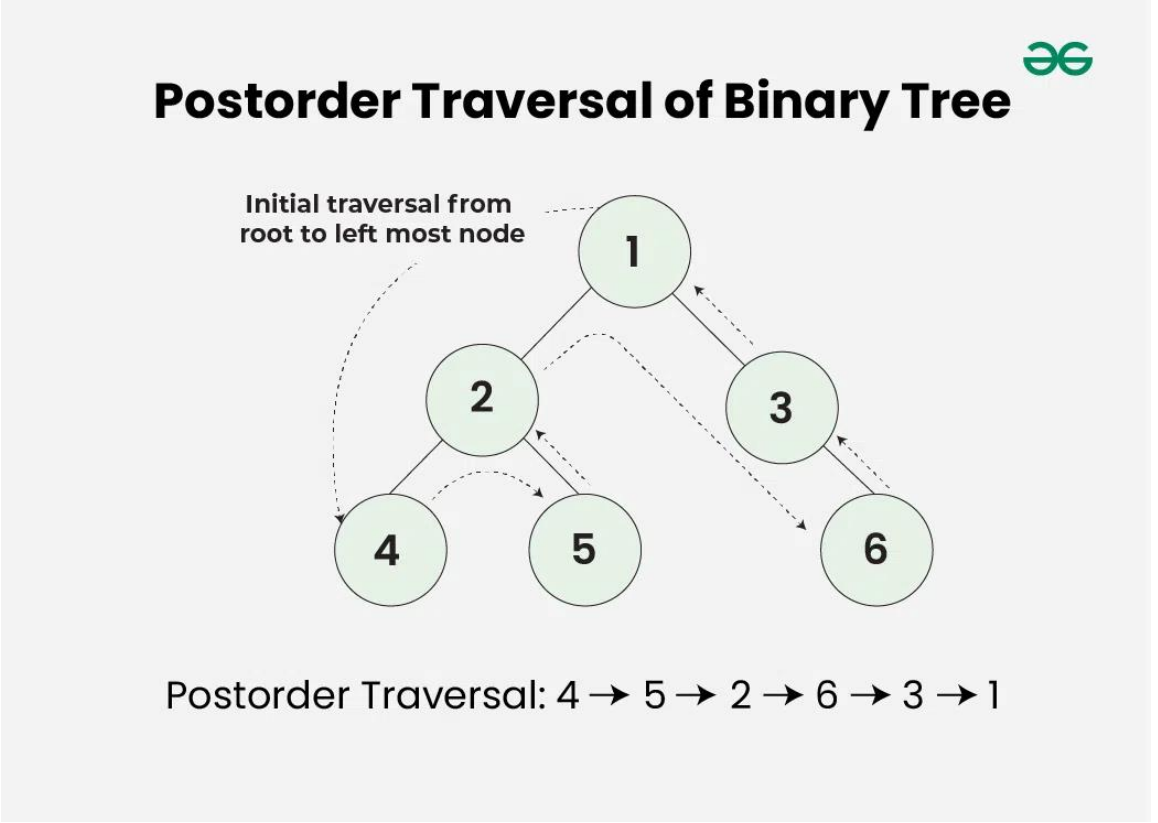
BINARY SEARCH TREE : BASIC OPERATION

Basic operation:

- Seach node
- Insert node
- Delete node
- Tree traversal

Postorder traversal:

- Used to delete the tree.
- Useful to get the postfix expression of an expression tree.



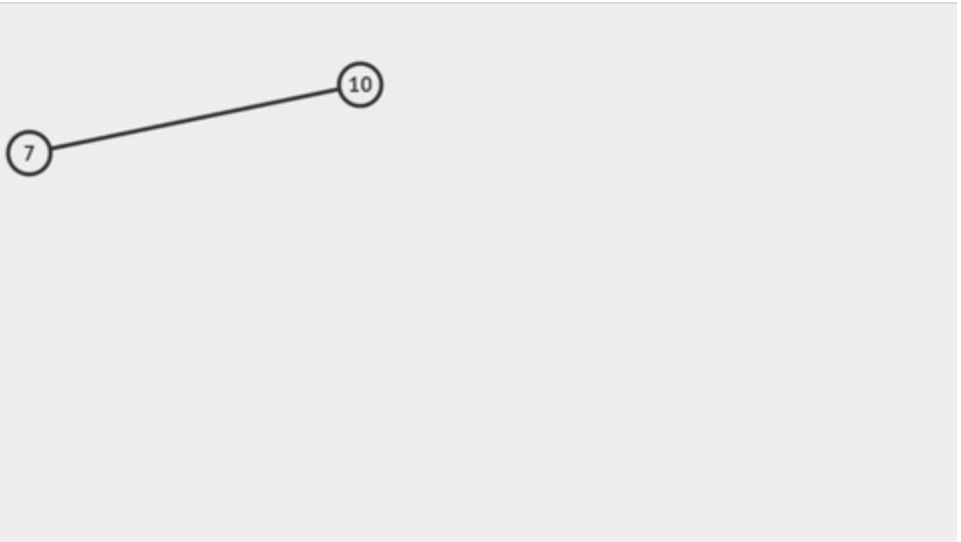
BINARY SEARCH TREE : PROS AND CONS

Advantage:

- Efficient searching: O(log n) time complexity for searching
- Ordered structure: easy to find the next or previous element
- Dynamic insertion and deletion: Elements can be added or removed efficiently

Disadvantage:

- Not self-balancing effect that time complexity in worst-case is linear: O(n)
- Memory overhead: require additional memory to store pointers to child nodes
- Limited functionality: BSTs only support searching, insertion, and deletion operations



BINARY SEARCH TREE : APPLICATION

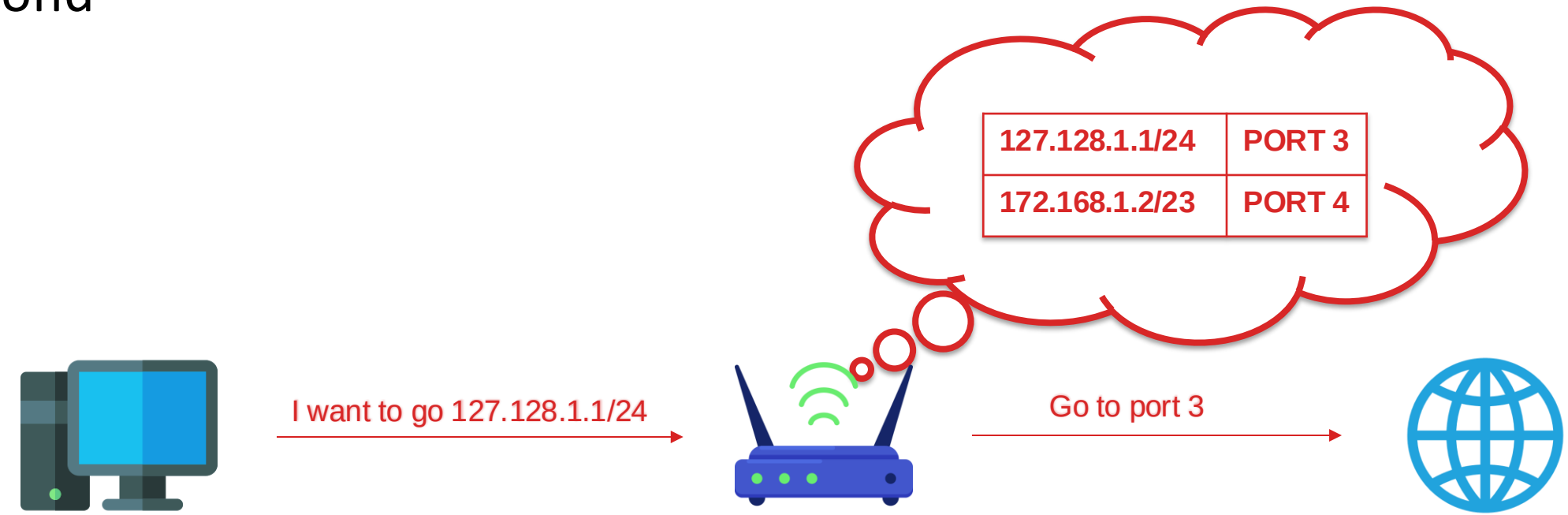
- **Searching**: Finding a specific element in a sorted collection
- **Sorting**: Sorting a collection of elements in ascending or descending order
- **Range queries**: Finding elements within a specified range
- **Data storage**: Storing and retrieving data in a hierarchical manner
- **Databases**: Indexing data for efficient retrieval
- **Computer graphics**: Representing spatial data in a tree structure
- **Artificial intelligence**: Decision trees and rule-based systems

3. BINARY SEARCH TREE APPLICATION: ROUTING TABLE

ROUTING TABLE

A routing table is a set of rules, often viewed in table format, that's used to determine where data packets traveling over an Internet Protocol (IP) network will be directed.

To handle network traffic, a router consults its routing tables millions of times each second



ROUTING TABLE

Routing table require:

- Store IPv4, IPv6 routing
- Searching node as quick as posible

Solution:

- Using binary search tree for quick searching
- Using prefix as key

```
PS C:\Users\QuyetDH\Documents\Simple_Routing_Table> route print
===========================================================================
Interface List
  8...1e b7 0d ca 1c db ......Microsoft Wi-Fi Direct Virtual Adapter
 17...2e b7 0d ca 1c db ......Microsoft Wi-Fi Direct Virtual Adapter #2
 10...00 0e c6 75 82 0d ......ASIX AX88772 USB2.0 to Fast Ethernet Adapter
 11...9c b7 0d ca 1c db ......Qualcomm Atheros AR9485WB-EG Wireless Network Ad
apter
  1...........................Software Loopback Interface 1
===========================================================================

IPv4 Route Table
===========================================================================
Active Routes:
Network Destination        Netmask          Gateway       Interface  Metric
          0.0.0.0          0.0.0.0      172.19.20.1    172.19.20.126     291
          0.0.0.0          0.0.0.0    192.168.101.1   192.168.101.68      55
        127.0.0.0        255.0.0.0         On-link        127.0.0.1     331
        127.0.0.1  255.255.255.255         On-link        127.0.0.1     331
  127.255.255.255  255.255.255.255         On-link        127.0.0.1     331
      172.19.20.0    255.255.255.0         On-link     172.19.20.126     291
    172.19.20.126  255.255.255.255         On-link     172.19.20.126     291
    172.19.20.255  255.255.255.255         On-link     172.19.20.126     291
    192.168.101.0    255.255.255.0         On-link    192.168.101.68     311
   192.168.101.68  255.255.255.255         On-link    192.168.101.68     311
  192.168.101.255  255.255.255.255         On-link    192.168.101.68     311
        224.0.0.0        240.0.0.0         On-link        127.0.0.1     331
        224.0.0.0        240.0.0.0         On-link     172.19.20.126     291
        224.0.0.0        240.0.0.0         On-link    192.168.101.68     311
  255.255.255.255  255.255.255.255         On-link        127.0.0.1     331
  255.255.255.255  255.255.255.255         On-link     172.19.20.126     291
  255.255.255.255  255.255.255.255         On-link    192.168.101.68     311
===========================================================================
Persistent Routes:
  Network Address          Netmask  Gateway Address  Metric
          0.0.0.0          0.0.0.0      172.19.20.1  Default
===========================================================================

IPv6 Route Table
===========================================================================
Active Routes:
 If Metric Network Destination      Gateway
```

ROUTING TABLE – NAVIGATION

ROUTING TABLE – DEFINE DATA TYPE

Routing table data type:

```
114  /* Routing table structure. */
115  struct route_table
116  {
117    struct route_node *top;
118    u_int32_t id;
119  };
120
121  /* Routing node structure. */
122  struct route_node
123  {
124    struct route_node *link[2];
125  #define l_left   link[0]
126  #define l_right  link[1]
127    struct prefix p;
128    struct route_table *table;
129    struct route_node *parent;
130    u_int32_t lock;
131    void *info;
132    void *aggregate;
133  };
```

```
 94  /* Prefix structure IPv4. */
 95  struct prefix
 96  {
 97    u_int8_t family;
 98    u_int8_t prefixlen;
 99    u_int8_t prefix_style;
100    u_int8_t pad1;
101    union
102    {
103      u_int8_t prefix;
104      struct pal_in4_addr prefix4;
105      struct
106      {
107        struct pal_in4_addr id;
108        struct pal_in4_addr adv_router;
109      } lp;
110      u_int8_t val[50];
111    } u;
112  };
```

```
43  /* IPv4 Address Structure */
44  struct pal_in4_addr
45  {
46    union
47    {
48      struct
49      {
50        uint8_t octet_1;
51        uint8_t octet_2;
52        uint8_t octet_3;
53        uint8_t octet_4;
54      } in4_octet;
55      uint32_t in4_addr;
56    } u;
57    uint32_t s_addr;
58  };
```

ROUTING TABLE – DEFINE PROTOTYPE FUNCTION

```
/*=============================================================================
** Tool prototypes
*/

/* Create prefix struct */
struct prefix * create_prefix(u_int8_t family, u_int8_t prefix_style, u_int8_t pad1,
                              u_int8_t octet1, u_int8_t octet2, u_int8_t octet3, u_int8_t octet4,
                              u_int8_t prefixlen);

/* Print prefix in format 0.0.0.0/0 */
void printf_prefix(struct prefix);
```

```
/*=============================================================
** Table prototypes
*/

/* Init routing table */
struct route_table *route_table_init (void);

/* Set table ID */
void route_table_id_set (struct route_table *, u_int32_t);

/* Check node's info in table */
u_char route_table_has_info (struct route_table *);

/* Print routing table info */
void route_table_print_info (struct route_table *);

/* Print routing table */
void route_table_print_table (struct route_table *);

/* Deinit routing table */
void route_table_finish (struct route_table *);
```

```
/*=============================================================================
** Node prototypes
*/

/* Top node in routing table */
struct route_node *route_top (struct route_table *);

/* Check matching prefix node */
struct route_node *route_node_match (struct route_table *, struct prefix *);

/* Insert new node to table */
struct route_node *route_node_get (struct route_table *, struct prefix *);

/* Search node in table */
struct route_node *route_node_lookup (struct route_table *, struct prefix *);

/* Travel routing table */
struct route_node *route_next (struct route_node *);

/* Printf route node. */
void route_node_printf(struct route_node* node);

/* Lock node while using */
struct route_node *route_lock_node (struct route_node *node);

/* Unlock node */
void route_unlock_node (struct route_node *node);

/* Delete node */
void route_node_delete (struct route_node *node);

/* Free node memory */
void route_node_free (struct route_node *node);

#endif
```

ROUTING TABLE – DECLARE TOOL FUNCTION

```
3   /*=====================================================================
4   ** Create prefix struct */
5   struct prefix * create_prefix(
6       u_int8_t family,
7       u_int8_t prefix_style,
8       u_int8_t pad1,
9       u_int8_t octet1,
10      u_int8_t octet2,
11      u_int8_t octet3,
12      u_int8_t octet4,
13      u_int8_t prefixlen
14  )
15  {
16      struct prefix * p = (struct prefix *) malloc(sizeof(struct prefix));
17      p->family = family;
18      p->prefixlen = prefixlen;
19      p->prefix_style = prefix_style;
20      p->pad1 = pad1;
21      p->u.prefix4.u.in4_octet.octet_1 = octet1;
22      p->u.prefix4.u.in4_octet.octet_2 = octet2;
23      p->u.prefix4.u.in4_octet.octet_3 = octet3;
24      p->u.prefix4.u.in4_octet.octet_4 = octet4;
25      return p;
26  }
```

```
28  /*=============================================================
29  ** Printf prefix */
30  void printf_prefix(struct prefix p)
31  {
32    printf("%d.%d.%d.%d/%d",p.u.prefix4.u.in4_octet.octet_1,
33                            p.u.prefix4.u.in4_octet.octet_2,
34                            p.u.prefix4.u.in4_octet.octet_3,
35                            p.u.prefix4.u.in4_octet.octet_4,
36                            p.prefixlen);
37  }
```

ROUTING TABLE – DECLARE TABLE FUNCTION

```c
/*=================================================
** Init routing table */
struct route_table *
route_table_init (void)
{
  struct route_table *rt;

  rt = calloc (1, sizeof (struct route_table));
  if (rt == NULL)
    return NULL;

  return rt;
}
```

```c
/*=================================================================
** Specify an identifier for this route table. */
void
route_table_id_set (struct route_table *table, u_int32_t id)
{
  if (table)
    table->id = id;
}
```

```c
/*=================================================================
** check if the table contains nodes with info set */
u_char
route_table_has_info (struct route_table *table)
{
  struct route_node *node;

  if (table == NULL)
    return 0;

  node = table->top;

  while (node)
    {
      if (node->info)
        return 1;

      if (node->l_left)
        {
          node = node->l_left;
          continue;
        }

      if (node->l_right)
        {
          node = node->l_right;
          continue;
        }

      while (node->parent)
        {
          if (node->parent->l_left == node && node->parent->l_right)
            {
              node = node->parent->l_right;
              break;
            }
          node = node->parent;
        }

      if (node->parent == NULL)
        break;
    }

  return 0;
}
```

ROUTING TABLE – DECLARE TABLE FUNCTION

```
108  /*=========================================================
109  ** Print routing table info */
110  void route_table_print_info (struct route_table * table)
111  {
112      printf("=======================================\n");
113      printf("| Table %5d    address: %10x |\n",table->id , table);
114      printf("| Top node: %10x                |\n", table->top);
115      printf("=======================================\n");
116  }
```

```
118  /*=========================================================
119  ** Print routing table */
120  void route_table_print_table (struct route_table * table)
121  {
122      route_table_print_info(table);
123
124      struct route_node* node;
125      node = table->top;
126
127      while (node)
128      {
129          route_node_printf(node);
130          node = route_next(node);
131      }
132  }
```

ROUTING TABLE – DECLARE TABLE FUNCTION

```
134  /*=========================================
135  ** Free route table. */
136  void
137  route_table_free (struct route_table *rt)
138  {
139    struct route_node *tmp_node;
140    struct route_node *node;
141
142    if (rt == NULL)
143      return;
144
145    node = rt->top;
146
147    while (node)
148      {
149        if (node->l_left)
150          {
151            node = node->l_left;
152            continue;
153          }
154
155        if (node->l_right)
156          {
157            node = node->l_right;
158            continue;
159          }
160
161        tmp_node = node;
162        node = node->parent;
163
164        if (node != NULL)
165          {
166            if (node->l_left == tmp_node)
167              node->l_left = NULL;
168            else
169              node->l_right = NULL;
170
171            route_node_free (tmp_node);
172          }
173        else
174          {
175            route_node_free (tmp_node);
176            break;
177          }
178      }
179
180    free(rt);
181    return;
182  }
```

```
184  /*=========================================
185  ** Deinit routing table */
186  void
187  route_table_finish (struct route_table *rt)
188  {
189    route_table_free (rt);
190  }
```

ROUTING TABLE – DECLARE NODE FUNCTION

```
202  /*==========================================================================
203  ** Get fist node and lock it.  This function is useful when one want
204     to lookup all the node exist in the routing table. */
205  struct route_node *
206  route_top (struct route_table *table)
207  {
208    /* If there is no node in the routing table return NULL. */
209    if (table == NULL || table->top == NULL)
210      return NULL;
211
212    /* Lock the top node and return it. */
213    route_lock_node (table->top);
214    return table->top;
215  }
```

```
217  /*==========================================================================
218  ** Utility mask array. */
219  static const u_char maskbit[] =
220  {
221    0x00, 0x80, 0xc0, 0xe0, 0xf0, 0xf8, 0xfc, 0xfe, 0xff
222  };
223
224  /*==========================================================================
225  ** Macro version of check_bit (). */
226  #define CHECK_BIT(X,P) ((((u_char *)(X))[(P) / 8]) >> (7 - ((P) % 8)) & 1)
227
228  /*==========================================================================
229  ** Check bit of the prefix. */
230  static int
231  check_bit (u_char *prefix, u_char prefixlen)
232  {
233    int offset;
234    int shift;
235    u_char *p = (u_char *)prefix;
236
237    // pal_assert (prefixlen <= 128);
238
239    offset = prefixlen / 8;
240    shift = 7 - (prefixlen % 8);
241
242    return (p[offset] >> shift & 1);
243  }
```

ROUTING TABLE – DECLARE NODE FUNCTION

```
245  /*==========================================================================
246  ** Macro version of set_link (). */
247  #define SET_LINK(X,Y) (X)->link[CHECK_BIT(&(Y)->prefix,(X)->prefixlen)] = (Y);\
248                        (Y)->parent = (X)
249
250  static void
251  set_link (struct route_node *node, struct route_node *new)
252  {
253    int bit;
254
255    bit = check_bit (&new->p.u.prefix, node->p.prefixlen);
256
257    // pal_assert (bit == 0 || bit == 1);
258
259    node->link[bit] = new;
260    new->parent = node;
261  }
```

```
263  /*==========================================================
264  ** If n includes p prefix then return 1 else return 0. */
265  s_int32_t
266  prefix_match (struct prefix *n, struct prefix *p)
267  {
268    s_int32_t offset;
269    s_int32_t shift;
270
271    /* Set both prefix's head pointer. */
272    u_int8_t *np = (u_int8_t *)&n->u.prefix;
273    u_int8_t *pp = (u_int8_t *)&p->u.prefix;
274
275    /* If n's prefix is longer than p's one return 0. */
276    if (n->prefixlen > p->prefixlen)
277      return 0;
278
279    offset = n->prefixlen / PNBBY;
280    shift =  n->prefixlen % PNBBY;
281
282    if (shift)
283      if (maskbit[shift] & (np[offset] ^ pp[offset]))
284        return 0;
285
286    while (offset--)
287      if (np[offset] != pp[offset])
288        return 0;
289    return 1;
290  }
```

ROUTING TABLE – DECLARE NODE FUNCTION

```c
/*==========================================================*
** Copy prefix from src to dest. */
void
prefix_copy (struct prefix *dest, struct prefix *src)
{
  dest->family = src->family;
  dest->prefixlen = src->prefixlen;
  dest->u.prefix4 = src->u.prefix4;
}

/*==========================================================================*
** Allocate new route node. */
struct route_node *
route_node_create ()
{
  return (struct route_node *) calloc(1, sizeof (struct route_node));
}

/*==========================================================================*
** Allocate new route node with prefix set. */
struct route_node *
route_node_set (struct route_table *table, struct prefix *prefix)
{
  struct route_node *node;

  node = calloc(1, sizeof (struct route_node));

  if (node == NULL)
    return NULL;

  prefix_copy (&node->p, prefix);
  node->table = table;

  return node;
}
```

```c
/*==========================================================================*
** Common prefix route genaration. */
static void
route_common (struct prefix *n, struct prefix *p, struct prefix *new)
{
  int i;
  u_char diff;
  u_char mask;

  u_char *np = (u_char *)&n->u.prefix;
  u_char *pp = (u_char *)&p->u.prefix;
  u_char *newp = (u_char *)&new->u.prefix;

  for (i = 0; i < p->prefixlen / 8; i++)
    {
      if (np[i] == pp[i])
        newp[i] = np[i];
      else
        break;
    }

  new->prefixlen = i * 8;

  if (new->prefixlen != p->prefixlen)
    {
      diff = np[i] ^ pp[i];
      mask = 0x80;
      while (new->prefixlen < p->prefixlen && !(mask & diff))
        {
          mask >>= 1;
          new->prefixlen++;
        }
      newp[i] = np[i] & maskbit[new->prefixlen % 8];
    }
}
```

ROUTING TABLE – DECLARE NODE FUNCTION

```
364  /*==========================================================================
365  ** Find matched prefix. */
366  struct route_node *
367  route_node_match (struct route_table *table, struct prefix *p)
368  {
369    struct route_node *node;
370    struct route_node *matched;
371
372    matched = NULL;
373    node = table->top;
374
375    /* Walk down tree.  If there is matched route then store it to
376       matched. */
377    while (node && node->p.prefixlen <= p->prefixlen
378                && prefix_match (&node->p, p))
379      {
380        if (node->info)
381          matched = node;
382        node = node->link[check_bit(&p->u.prefix, node->p.prefixlen)];
383      }
384
385    /* If matched route found, return it. */
386    if (matched)
387      return route_lock_node (matched);
388
389    return NULL;
390  }
```

ROUTING TABLE – DECLARE NODE FUNCTION

```
392  /*==========================================================================
393  ** Add node to routing table. */
394  struct route_node *
395  route_node_get (struct route_table *table, struct prefix *p)
396  {
397    struct route_node *new = NULL;
398    struct route_node *node = NULL;
399    struct route_node *match = NULL;
400
401    match = NULL;
402
403    if (! table)
404      return NULL;
405
406    node = table->top;
407    while (node && node->p.prefixlen <= p->prefixlen &&
408           prefix_match (&node->p, p))
409    {
410      if (node->p.prefixlen == p->prefixlen)
411        {
412          route_lock_node (node);
413          return node;
414        }
415      match = node;
416      node = node->link[check_bit(&p->u.prefix, node->p.prefixlen)];
417    }
418
419    /* If table is empty, setup top route node */
420    if (node == NULL)
421      {
422        new = route_node_set (table, p);
423
424        if (new == NULL)
425          return NULL;
426
427        if (match)
428          set_link (match, new);
429        else
430          table->top = new;
431      }
```

ROUTING TABLE – DECLARE NODE FUNCTION

```
432     /* Else, create a new node */
433     else
434     {
435       new = route_node_create ();
436       if (NULL == new)
437         return NULL;
438
439       /* Create network prefix to new prefix */
440       route_common (&node->p, p, &new->p);
441       new->p.family = p->family;
442       new->info = NULL;
443       new->table = table;
444       set_link (new, node);
445
446       if (match)
447         set_link (match, new);
448       else
449         table->top = new;
450
451       if (new->p.prefixlen != p->prefixlen)
452       {
453         match = new;
454         new = route_node_set (table, p);
455
456         if (NULL == new)
457         {
458           /* Delete the "match" (above created "new") node */
459           route_node_delete (match);
460           return NULL;
461         }
462         set_link (match, new);
463       }
464     }
465     route_lock_node (new);
466
467     return new;
468   }
```

ROUTING TABLE – DECLARE NODE FUNCTION

```
470  /*==========================================================================
471  ** Lookup same prefix node.  Return NULL when we can't find route. */
472  struct route_node *
473  route_node_lookup (struct route_table *table, struct prefix *p)
474  {
475    struct route_node *node;
476
477    node = table->top;
478
479    while (node && node->p.prefixlen <= p->prefixlen &&
480           prefix_match (&node->p, p))
481      {
482        if (node->p.prefixlen == p->prefixlen && node->info)
483          return route_lock_node (node);
484
485        node = node->link[check_bit(&p->u.prefix, node->p.prefixlen)];
486      }
487
488    return NULL;
489  }
```

ROUTING TABLE – DECLARE NODE FUNCTION

```
491  /*===================================================================
492  ** Unlock current node and lock next node then return it. */
493  struct route_node *
494  route_next (struct route_node *node)
495  {
496    struct route_node *next;
497    struct route_node *start;
498
499    /* Node may be deleted from route_unlock_node so we have to preserve
500       next node's pointer. */
501
502    if (node->l_left)
503      {
504        next = node->l_left;
505        route_lock_node (next);
506        route_unlock_node (node);
507        return next;
508      }
509    if (node->l_right)
510      {
511        next = node->l_right;
512        route_lock_node (next);
513        route_unlock_node (node);
514        return next;
515      }
516
517    start = node;
518    while (node->parent)
519      {
520        if (node->parent->l_left == node && node->parent->l_right)
521          {
522            next = node->parent->l_right;
523            route_lock_node (next);
524            route_unlock_node (start);
525            return next;
526          }
527        node = node->parent;
528      }
529    route_unlock_node (start);
530    return NULL;
531  }
```

ROUTING TABLE – DECLARE NODE FUNCTION

```
533  /*==========================================================================
534  ** Printf route node. */
535  void
536  route_node_printf(struct route_node* node)
537  {
538    printf("\n======================================================================");
539    if (node->parent)
540    {
541      printf("\n                  Parent %x : ", node->parent);
542      printf_prefix(node->parent->p);
543      printf("\n                                    |");
544    }
545
546    printf("\n                  Node %x : ", node);
547    printf_prefix(node->p);
548
549    if (node->link[0] || node->link[1])
550    {
551      printf("\n                    /                         \\");
552
553      if (node->link[0])
554      {
555        printf("\nLeft %x : ", node->link[0]);
556        printf_prefix(node->link[0]->p);
557      }
558      else
559        printf("\n                          ");
560
561      if (node->link[1])
562      {
563        printf("        Right %x : ", node->link[1]);
564        printf_prefix(node->link[1]->p);
565      }
566    }
567    if (node->info)
568    {
569      printf("\nInfo: %s", node->info);
570    } else
571    {
572      printf("\n");
573    }
574  }
```

ROUTING TABLE – DECLARE NODE FUNCTION

```
576  /*==================================================
577  ** Lock node. */
578  struct route_node *
579  route_lock_node (struct route_node *node)
580  {
581    node->lock++;
582    return node;
583  }
584
585  /*==================================================
586  ** Unlock node. */
587  void
588  route_unlock_node (struct route_node *node)
589  {
590    node->lock--;
591
592    if (node->lock == 0)
593      route_node_delete (node);
594  }
```

ROUTING TABLE – DECLARE NODE FUNCTION

```
596  /*================================================
597  ** Delete node from the routing table. */
598  void
599  route_node_delete (struct route_node *node)
600  {
601    struct route_node *child;
602    struct route_node *parent;
603
604    // pal_assert (node->lock == 0);
605    // pal_assert (node->info == NULL);
606
607    if (node->l_left && node->l_right)
608      return;
609
610    if (node->l_left)
611      child = node->l_left;
612    else
613      child = node->l_right;
614
615    parent = node->parent;
616
617    if (child)
618      child->parent = parent;
619
620    if (parent)
621      {
622        if (parent->l_left == node)
623          parent->l_left = child;
624        else
625          parent->l_right = child;
626      }
627    else
628      node->table->top = child;
629
630    route_node_free (node);
631
632    /* If parent node is stub then delete it also. */
633    if (parent && parent->lock == 0)
634      route_node_delete (parent);
635  }
636
637  /*================================================
638  ** Free route node. */
639  void
640  route_node_free (struct route_node *node)
641  {
642    free(node);
643  }
```

ROUTING TABLE – INIT TABLE

```
4   void main(){
5       struct route_table * table = route_table_init();
6       route_table_id_set(table, 1);
7
8       route_table_print_table(table);
9   }
```

```
PS C:\Users\QuyetDH\Documents\Simple_Routing_Table> gcc .\main.c .\table.c -o main;.\
ain
=====================================
| Table     1    address:    306e1450 |
| Top node:          0               |
=====================================
```

main.c

table.c

ROUTING TABLE – CREATING A NODE – TOP NODE

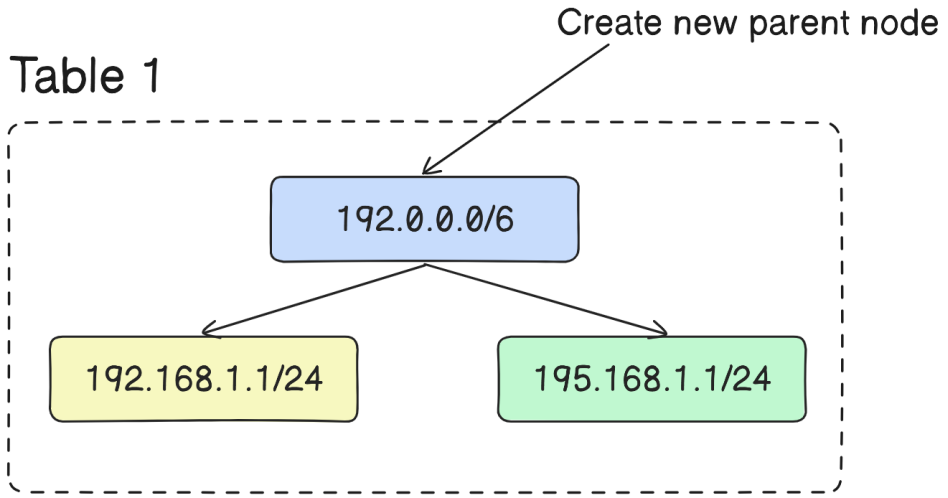
```
4   void main(){
5       struct route_table * table = route_table_init();
6       route_table_id_set(table, 1);
7
8       struct route_node * node;
9       /*======== create 192.168.1.1/24  =======*/
10      node = route_node_get(table, create_prefix(4,1,0,192,168,1,1,24));
11
12      route_table_print_table(table);
13  }
```

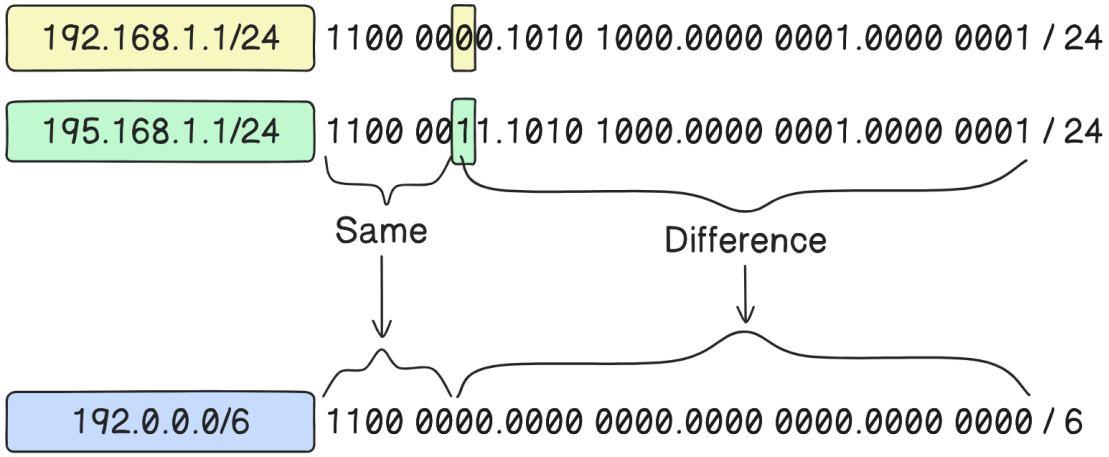
```
PS C:\Users\QuyetDH\Documents\Simple_Routing_Table> gcc main.c table.
c -o main.exe; .\main.exe
=====================================
| Table     1    address:   384f1450 |
| Top node:   384f14b0               |
=====================================

=====================================================================
                    Node 384f14b0 : 192.168.1.1/24
```

main.c

table.c

ROUTING TABLE – CREATING A NODE





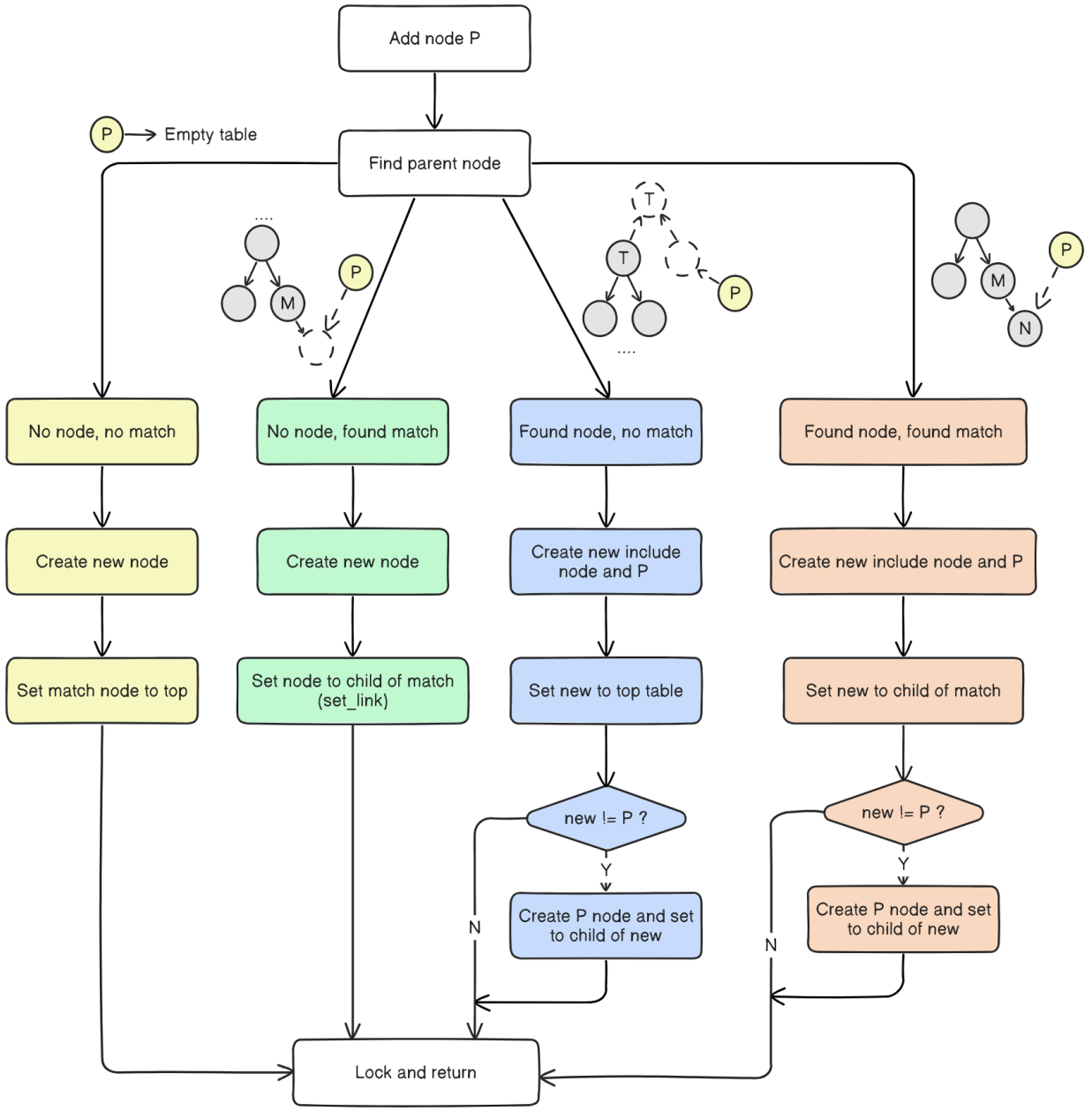

```c
#include <stdio.h>
#include "table.h"

void main(){
    struct route_table * table = route_table_init();
    route_table_id_set(table, 1);

    struct route_node * node;
    /*================= create 192.168.1.1/24  =============
      ======== 1100 0000.1010 1000.0000 0001.0000 0001 / 24 */
    node = route_node_get(table, create_prefix(4,1,0,192,168,1,1,24));

    /*================= create 195.168.1.1/24  =============
      ======== 1100 0011.1010 1000.0000 0001.0000 0001 / 24 */
    node = route_node_get(table, create_prefix(4,1,0,195,168,1,1,24));

    route_table_print_table(table);
}
```

```
PS C:\Users\QuyetDH\Documents\Simple_Routing_Table> gcc main.c table.
c -o main.exe; .\main.exe
====================================
| Table     1    address:   9fc21450 |
| Top node:   9fc21570               |
====================================

============================================================
                  Node 9fc21570 : 192.0.0.0/6
                  /                         \
Left 9fc214b0 : 192.168.1.1/24        Right 9fc215f0 : 195.168.1.1/24

============================================================
                Parent 9fc21570 : 192.0.0.0/6
                                |
                  Node 9fc214b0 : 192.168.1.1/24

============================================================
                Parent 9fc21570 : 192.0.0.0/6
                                |
                  Node 9fc215f0 : 195.168.1.1/24
```

main.c

table.c

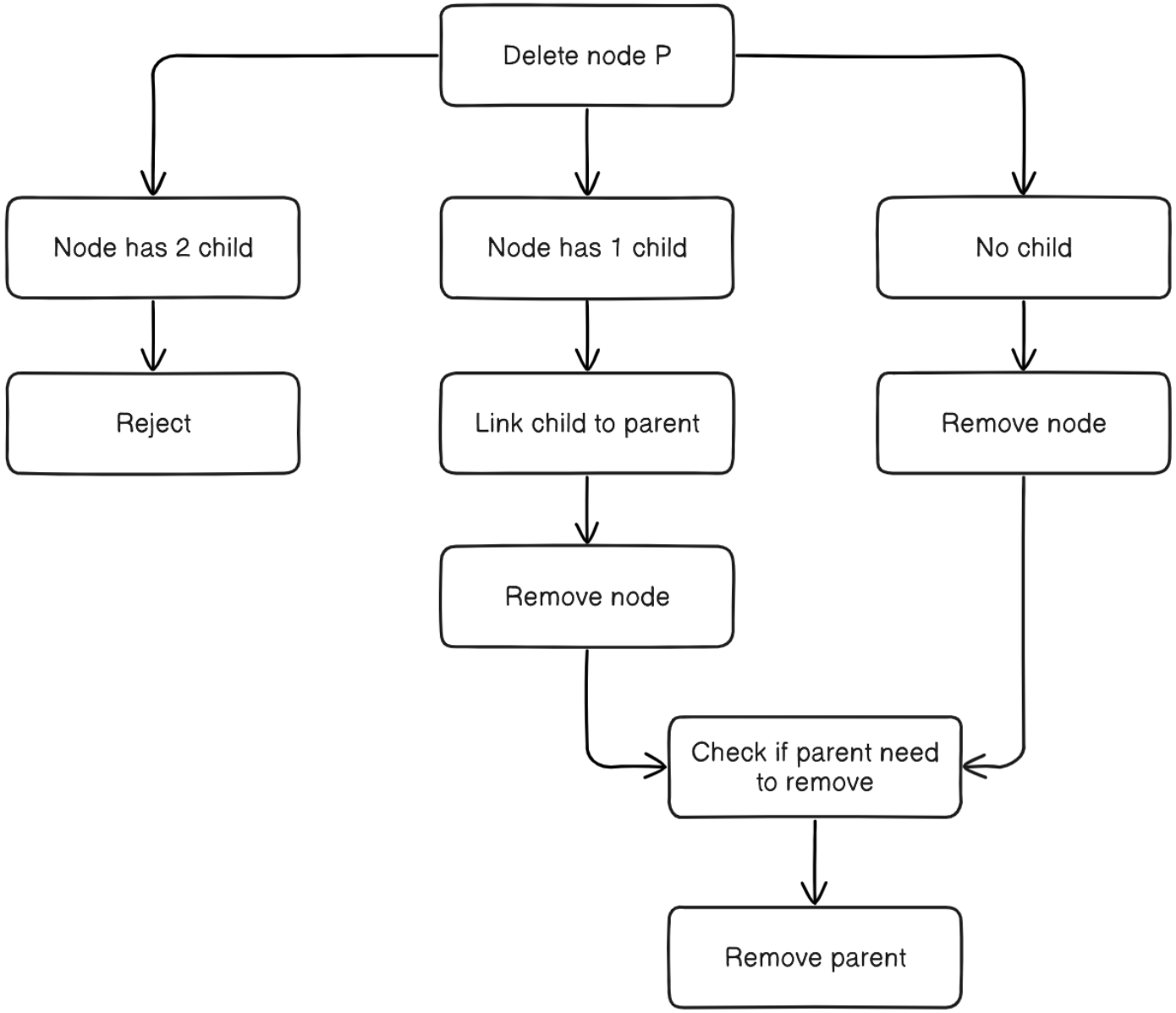
ROUTING TABLE – CREATING A NODE



main.c

table.c

ROUTING TABLE – DELETING A NODE



main.c

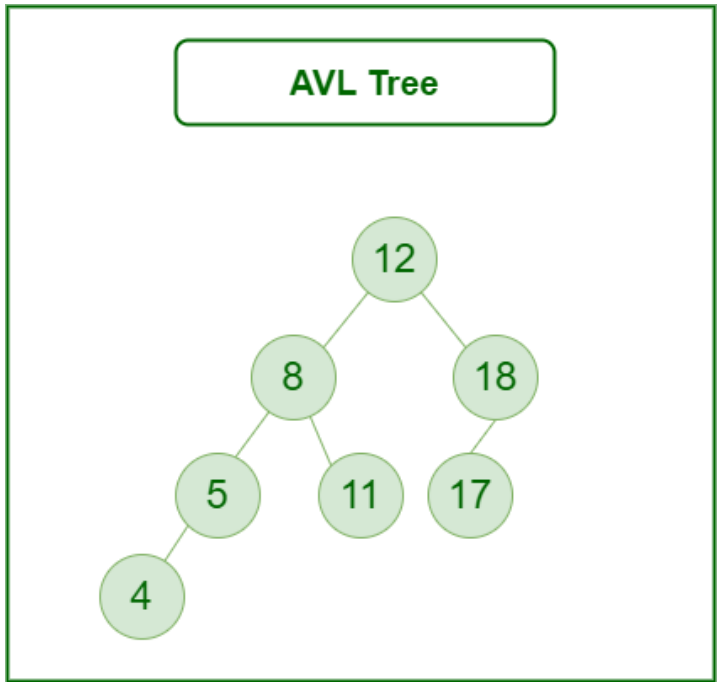
table.c

4. AVL TREE

AVL TREE : DEFINITION

An AVL tree **defined as a self-balancing Binary Search Tree (BST)** where the difference between height of left and right subtrees for any node cannot be more than one:

- Each node has at most 2 child (Left child & Right child)
- Each node has key and data
- Right child node key > Parent child node key
- Left child node key < Parent child node key
- Heights of left and right subtrees <= 1



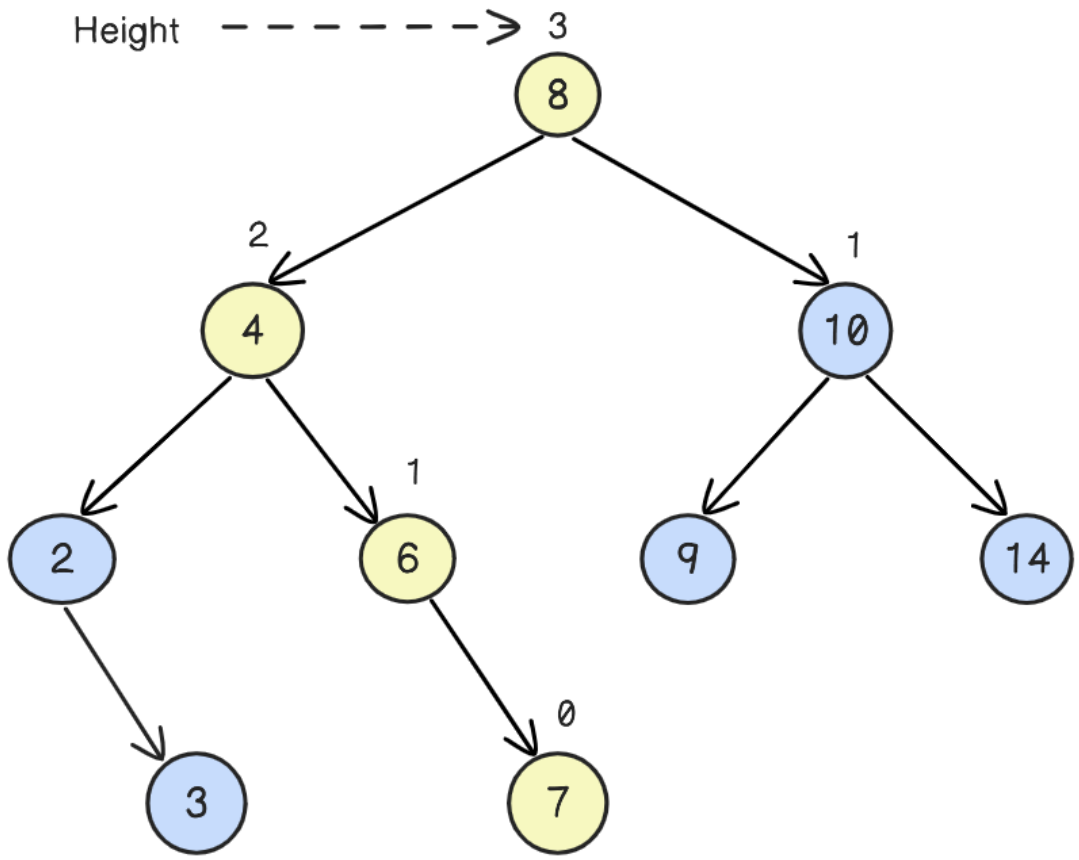
Handle same value node:

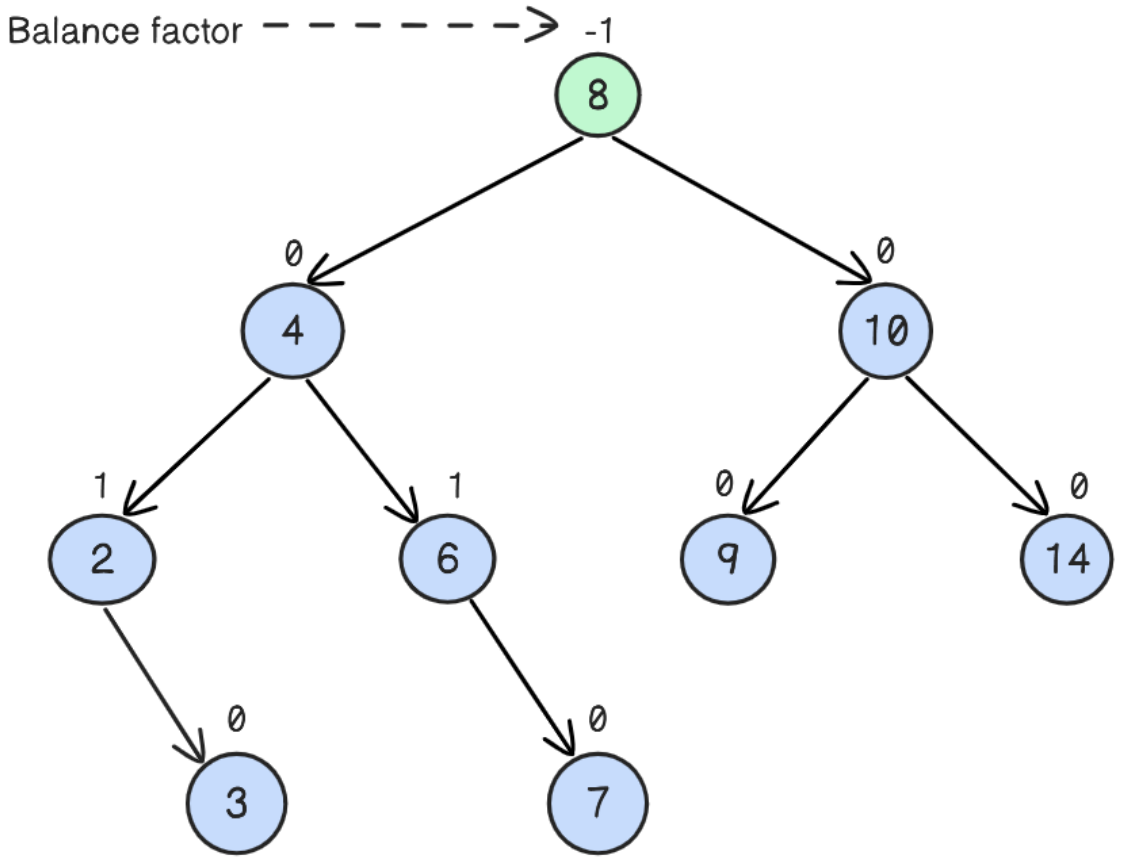
Left or right or reject but be consistent

AVL TREE : DEFINITION

Height: is the longest road from this node to null node

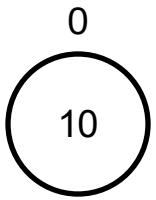
Balance factor: difference between height of left subtree and right subtree





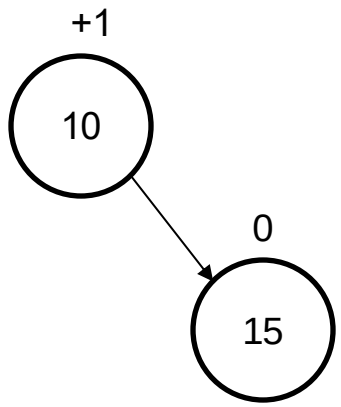
AVL TREE : DEMO

INSERT 10



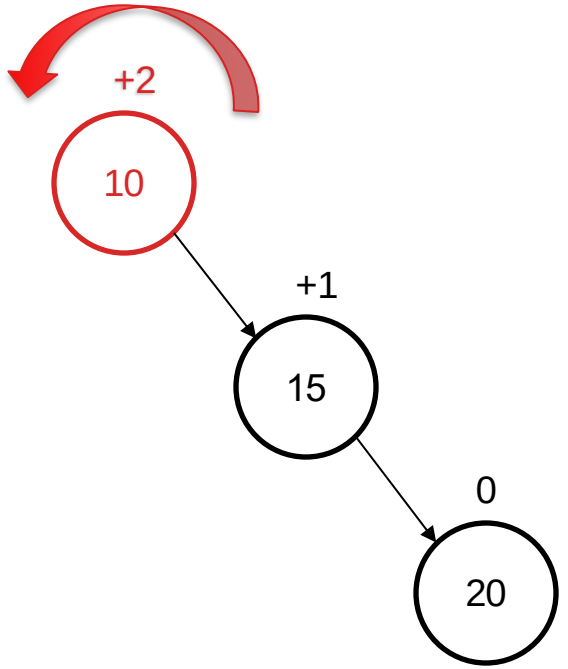
INSERT 15

AVL TREE : DEMO



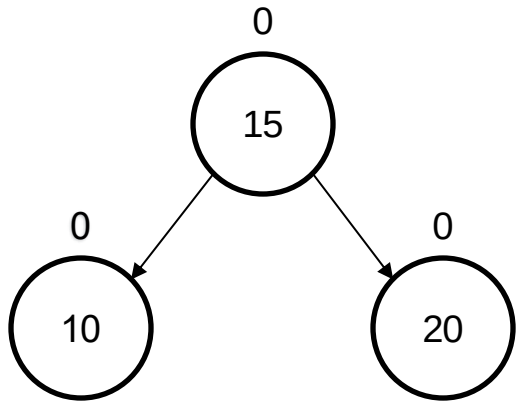
INSERT 20

AVL TREE : DEMO



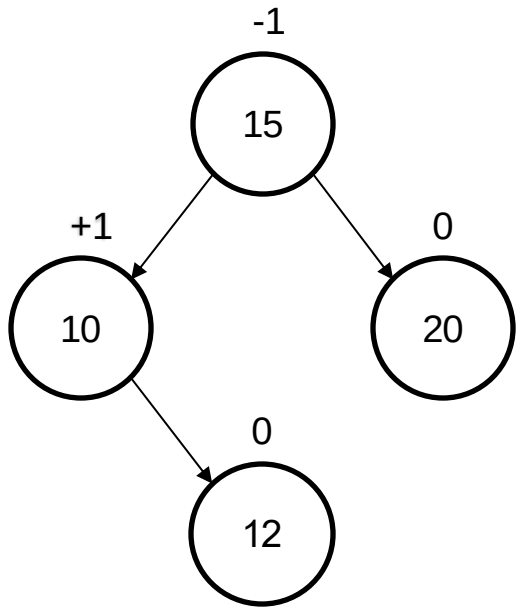
ROTATE LEFT

AVL TREE : DEMO



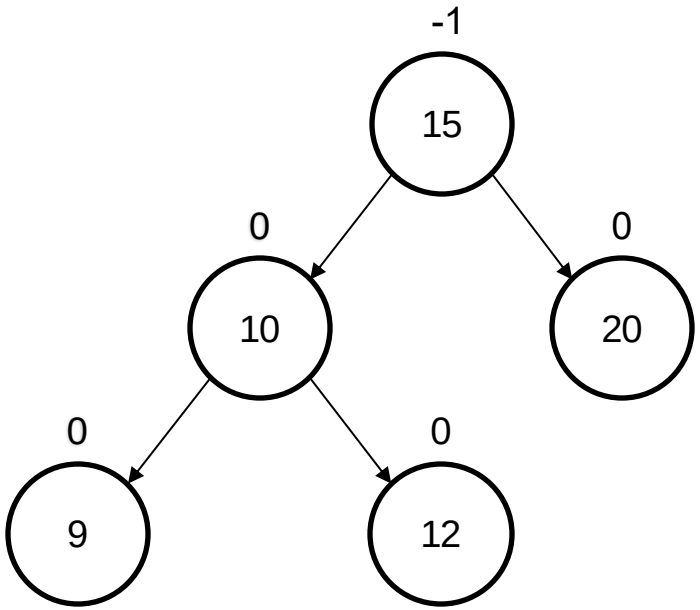
INSERT 12

AVL TREE : DEMO



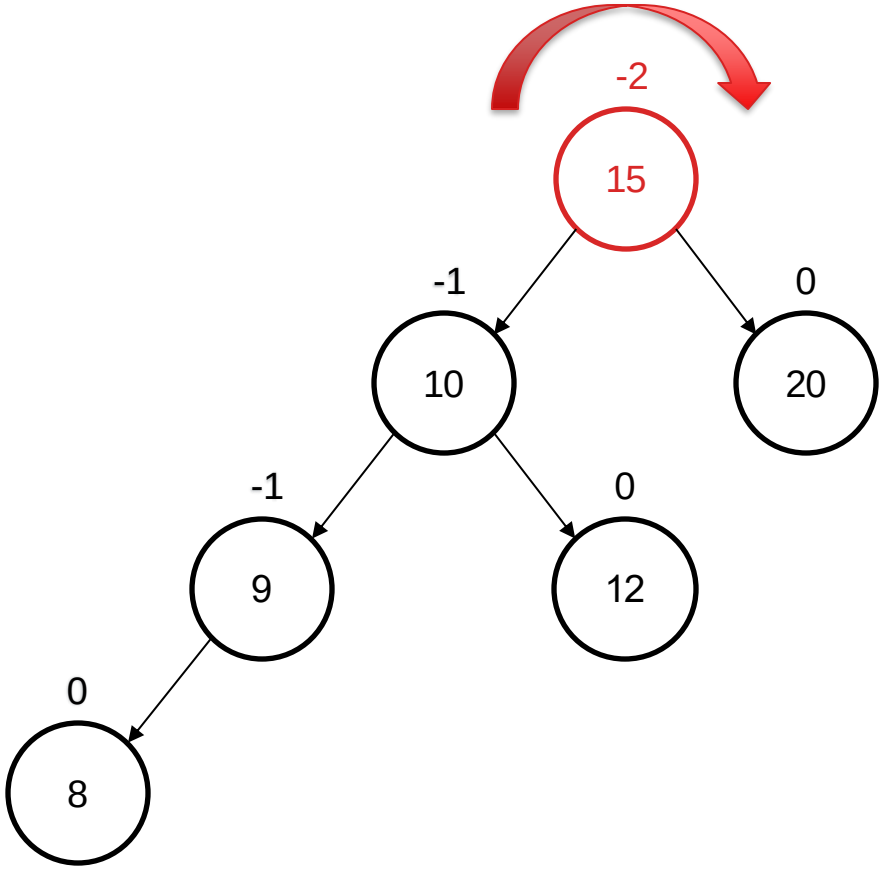
INSERT 9

AVL TREE : DEMO



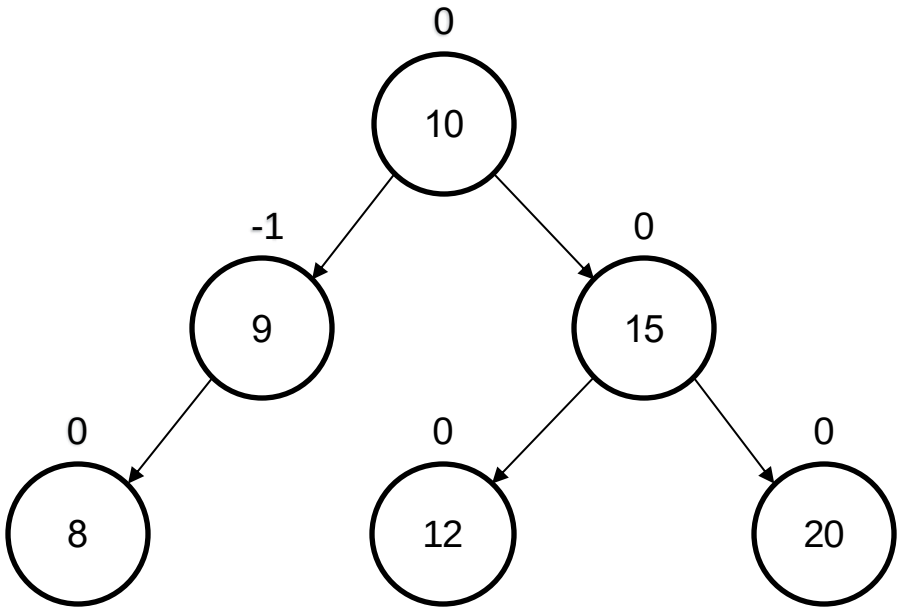
INSERT 8

AVL TREE : DEMO



ROTATE RIGHT

AVL TREE : DEMO



AVL TREE : INIT CODE

```c
/*================================================
 * INCLUDE HEADER
 */

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

/*================================================
 * DEFINE STRUCTURE
 */

struct avl_node
{
  struct avl_node *parent;
  struct avl_node *left;
  struct avl_node *right;
  int key;
  int height;
};

struct avl_tree
{
  struct avl_node *top;
};
```

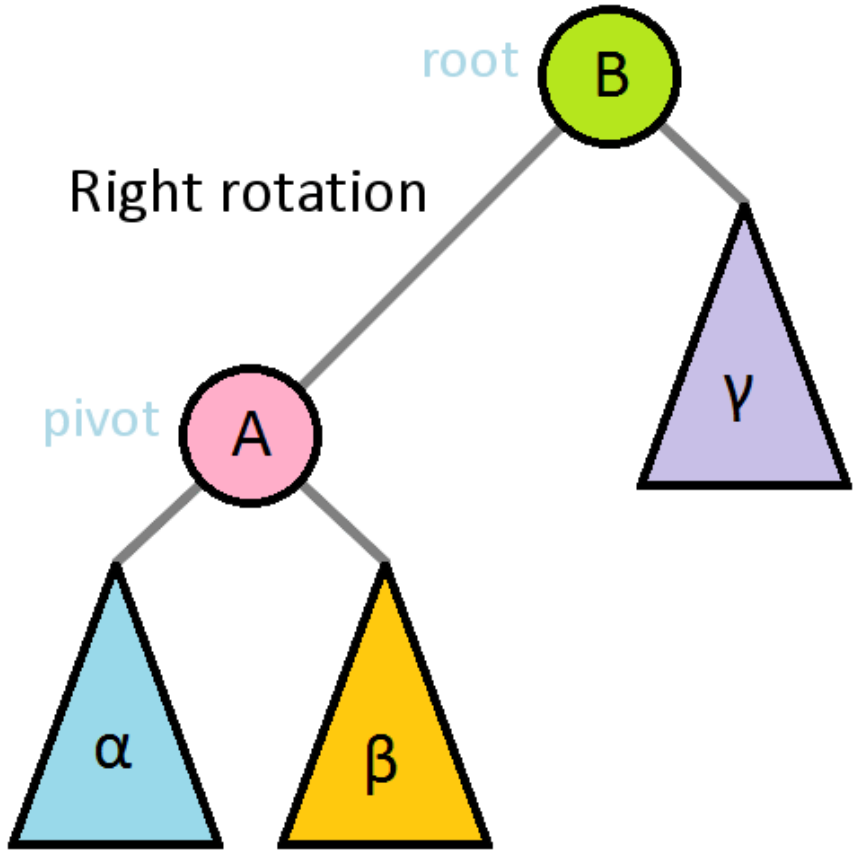
```c
/*================================================
 * DEFINE PROTOTYPE
 */

/* Max of a and b */
int
max(int a, int b)
{
  return (a>b ? a : b);
}

/* Get height of avl node */
int
height(struct avl_node *node) {
  if (NULL == node)
    return 0;
  return node->height;
}

/* Get the balance factor */
int
avl_node_balance_factor(struct avl_node *node)
{
  if (NULL == node)
    return 0;
  return height(node->right) - height(node->left);
}
```

AVL TREE : PRINT NODE

```c
/* Print node info */
void
avl_node_print(struct avl_node *node)
{
  if (NULL == node)
    return;

  printf("\n     Node: %d - Height: %d", node->key, node->height);
  if (node->left && node->right)
  {
    printf("\n    /        \\ ");
    printf("\n L:%d        R:%d", node->left->key, node->right->key);
  }
  else if (node->left)
  {
    printf("\n    /         ");
    printf("\n L:%d        ", node->left->key);
  }
  else if (node->right)
  {
    printf("\n            \\ ");
    printf("\n             R:%d", node->right->key);
  }
  printf("\n=========================");
}
```

AVL TREE : ROTATION

Left rotation and right rotation



AVL TREE : RIGHT ROTATE

```c
/* Right rotate */
struct avl_node *
avl_right_rotate(struct avl_tree *tree, struct avl_node *node)
{
  /*
          p_node                      p_node
            |                           |
          node                        l_node
          /   \                       /    \
      l_node   X        -->         X      node
      /    \                               /   \
     X      temp                          temp   X
  */
  /* Valid param check */
  if (NULL == node || NULL == tree || NULL == node->left)
    return NULL;

  struct avl_node *p_node = node->parent;
  struct avl_node *l_node = node->left;
  struct avl_node *temp = l_node->right;

  /* If node is top node, set top node */
  if (tree->top == node && node->parent == NULL)
    tree->top = l_node;

  /* Else set child node to parent node */
  if (p_node && p_node->left == node)
    p_node->left = l_node;
  else if (p_node && p_node->right == node)
    p_node->right = l_node;

  /* Rotate l_node */
  l_node->parent = p_node;
  l_node->right = node;

  /* Rotate node */
  node->parent = l_node;
  node->left = temp;

  /* If temp is exist, set temp parent */
  if (NULL != temp)
    temp->parent = node;

  /* Reset height of node and l_node */
  node->height = max(height(node->left), height(node->right)) + 1;
  l_node->height = max(height(l_node->left), height(l_node->right)) + 1;

  return l_node;
}
```

AVL TREE : LEFT ROTATE

```c
/* Left rotate */
struct avl_node *
avl_left_rotate(struct avl_tree *tree, struct avl_node *node)
{
  /*
        p_node                      p_node
          |                           |
        node                        r_node
        /   \                       /    \
       X     r_node     -->      node      X
             /    \              /   \
          temp     X            X    temp
  */
 /* Valid param check */
  if (NULL == tree || NULL == node || NULL == node->right)
    return NULL;

  struct avl_node *p_node = node->parent;
  struct avl_node *r_node = node->right;
  struct avl_node *temp = r_node->left;

  /* If node is top node, set top node */
  if (tree->top == node && node->parent == NULL)
    tree->top = r_node;

  /* Else set child node to parent node */
  if (p_node && p_node->left == node)
    p_node->left = r_node;
  else if (p_node && p_node->right == node)
    p_node->right = r_node;

  /* Rotate r_node */
  r_node->parent = p_node;
  r_node->left = node;

  /* Rotate node */
  node->parent = r_node;
  node->right = temp;

  /* If temp is exist, set temp parent */
  if (NULL != temp)
    temp->parent = node;

  /* Reset height of node and r_node */
  node->height = max(height(node->left), height(node->right)) + 1;
  r_node->height = max(height(r_node->left), height(r_node->right)) + 1;

  return r_node;
}
```

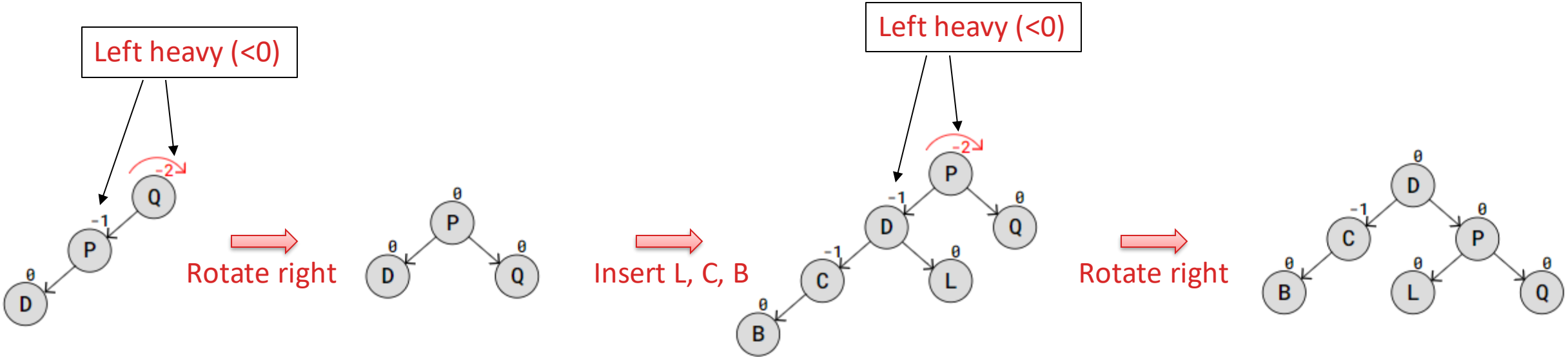
AVL TREE : INSERTION

The Four "out-of-balance" Cases

Case	Description	Rotation to Restore Balance
Left-Left (LL)	The unbalanced node and its left child node are both left-heavy.	A single right rotation.
Right-Right (RR)	The unbalanced node and its right child node are both right-heavy.	A single left rotation.
Left-Right (LR)	The unbalanced node is left heavy, and its left child node is right heavy.	First do a left rotation on the left child node, then do a right rotation on the unbalanced node.
Right-Left (RL)	The unbalanced node is right heavy, and its right child node is left heavy.	First do a right rotation on the right child node, then do a left rotation on the unbalanced node.

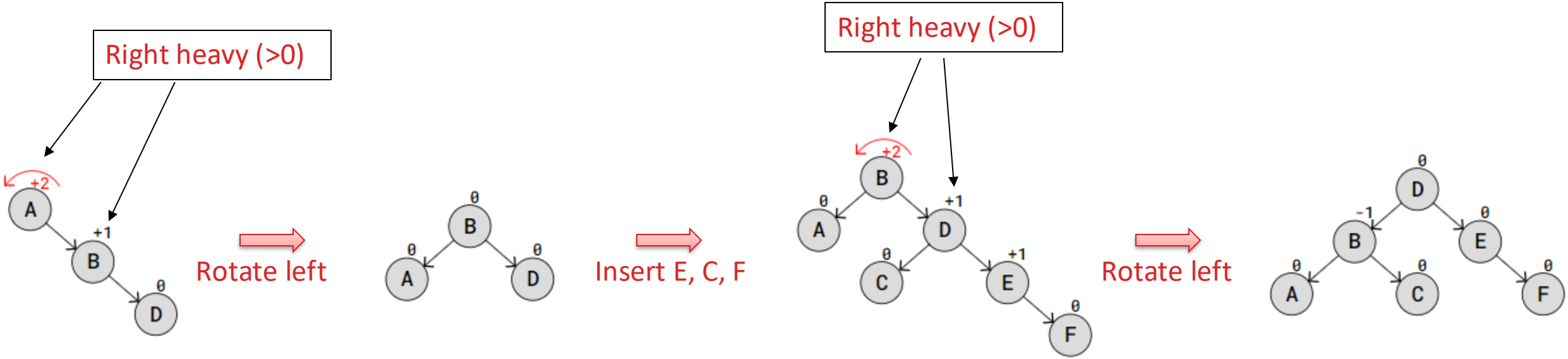
AVL TREE : INSERTION

The left-left case:



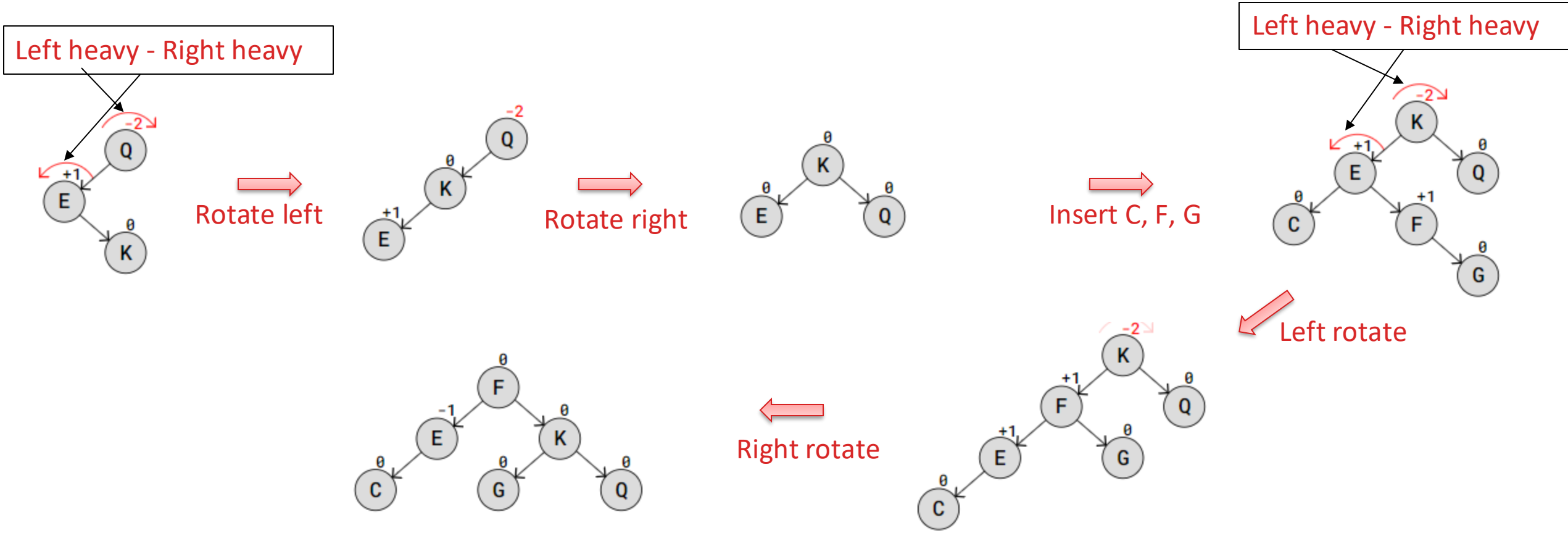
AVL TREE : INSERTION

The right-right case:



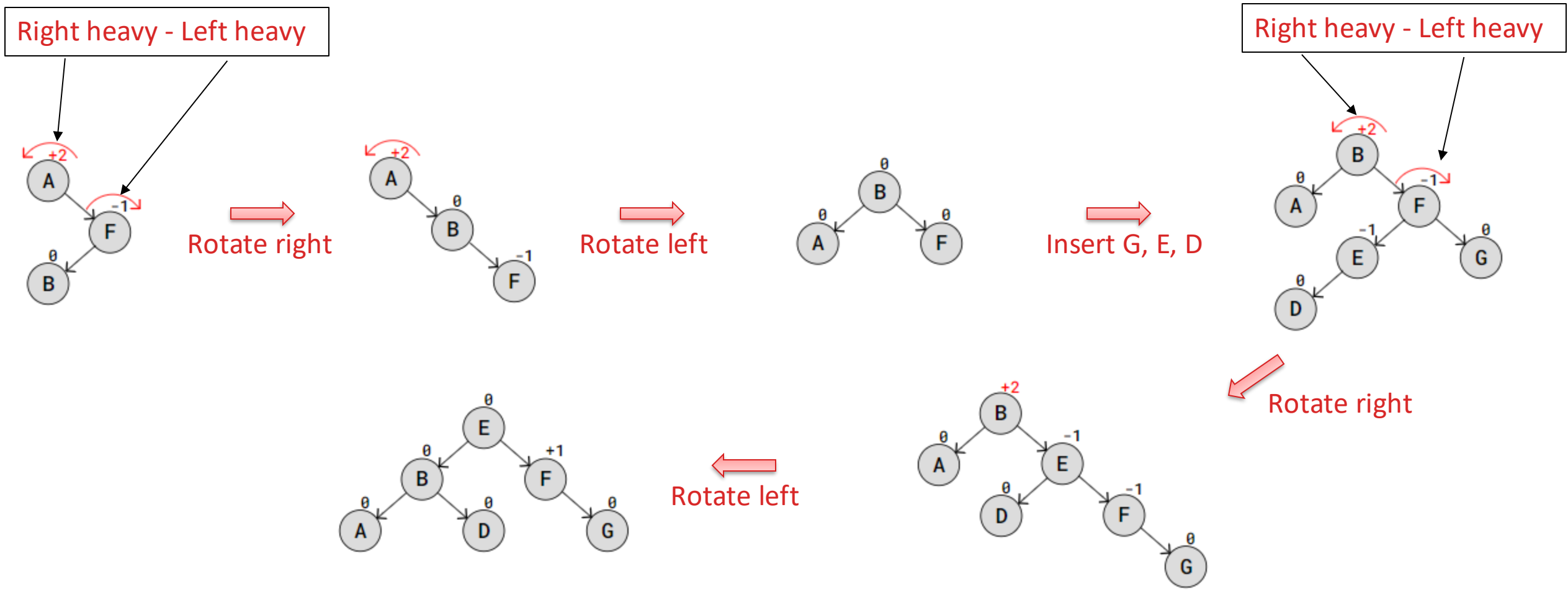
AVL TREE : INSERTION

The left-right case:



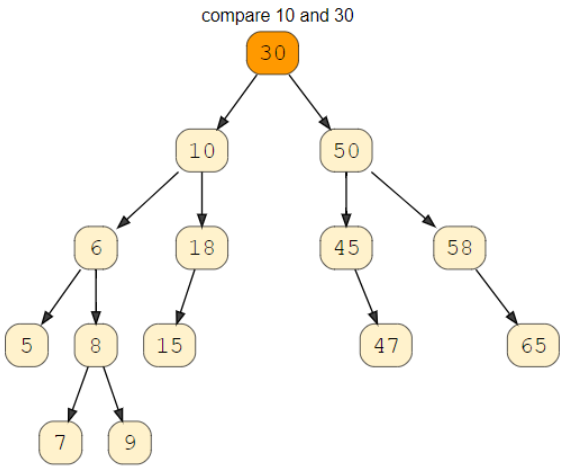
AVL TREE : INSERTION

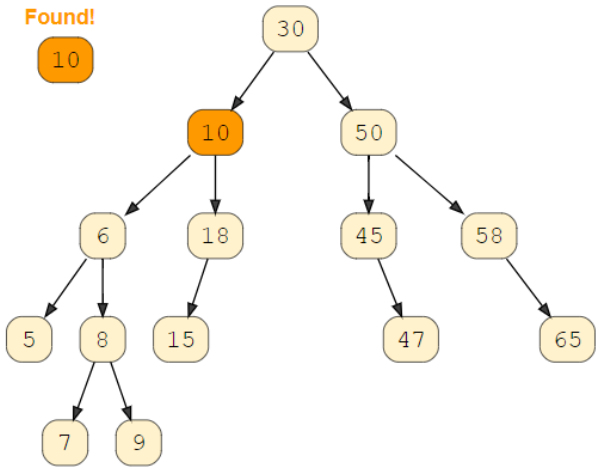
The right-left case:

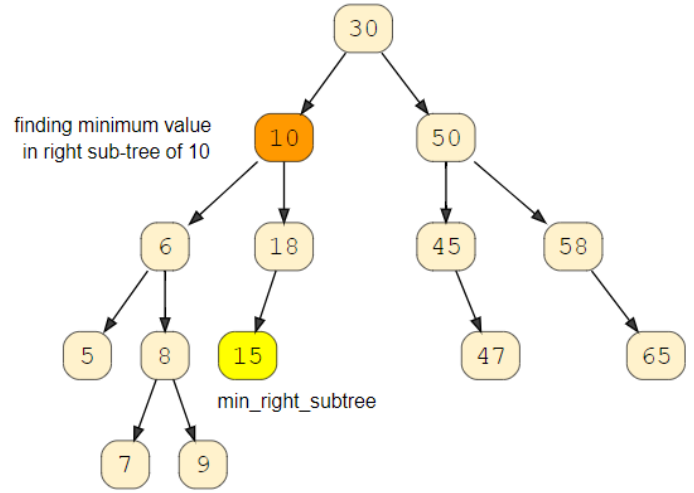


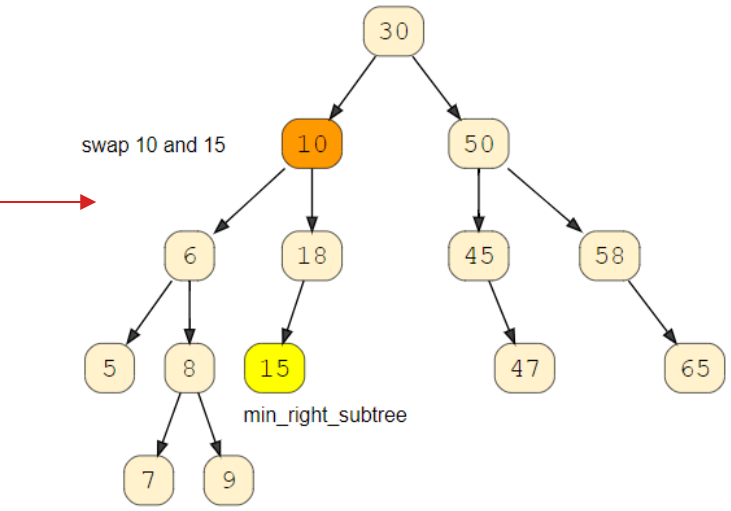
AVL TREE : DELETION

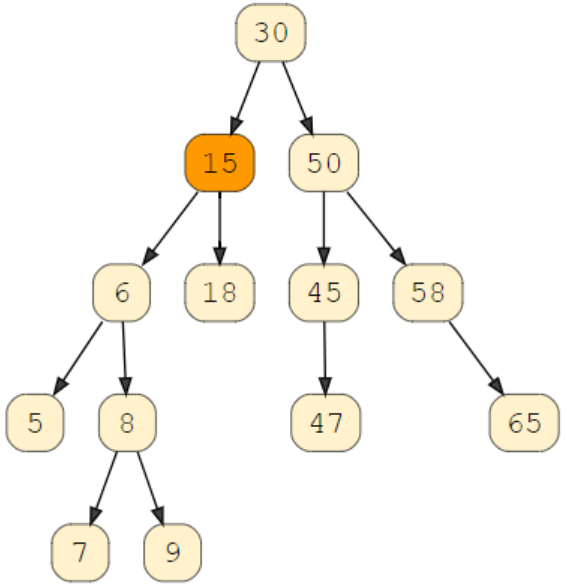
2 child case:

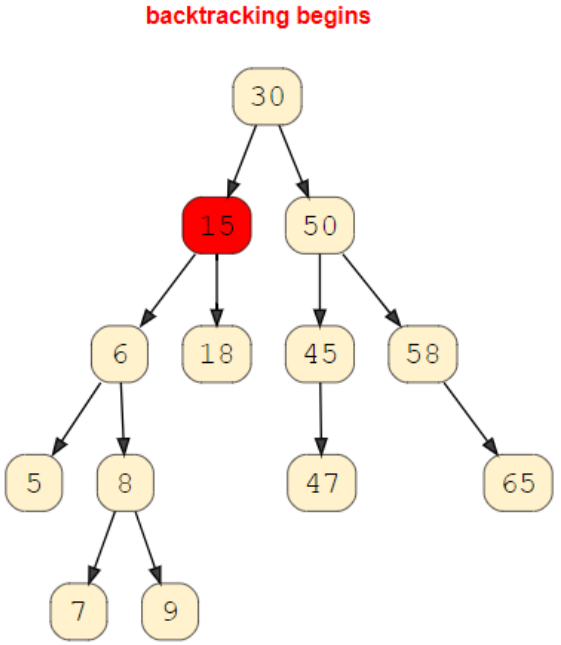






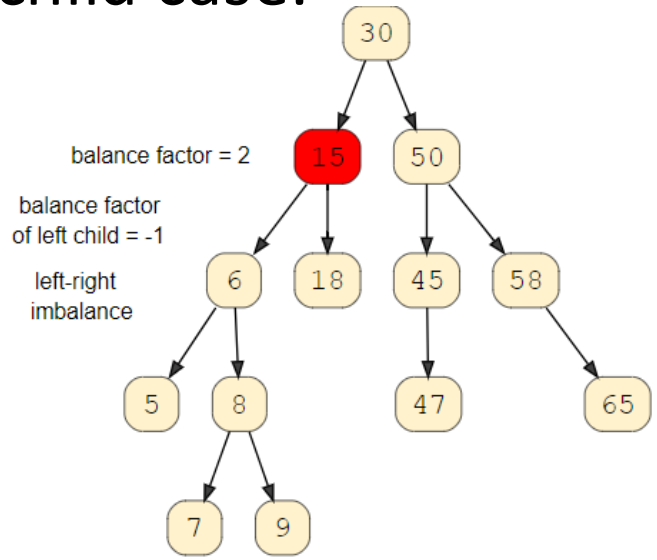




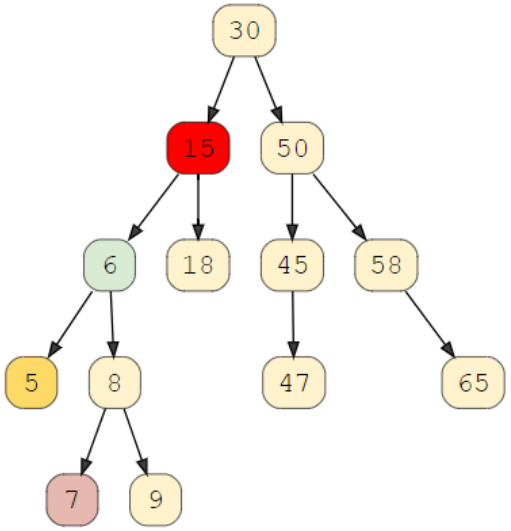


AVL TREE : DELETION

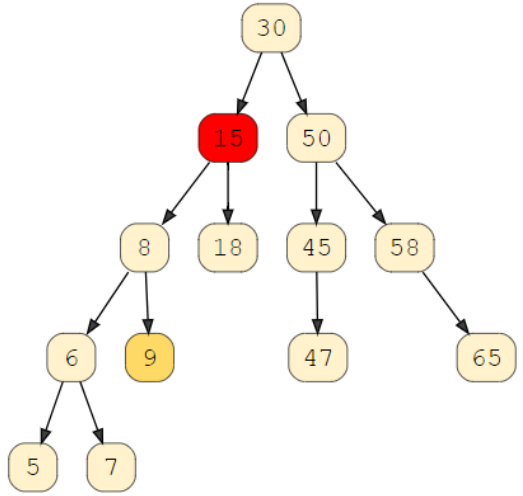
2 child case:

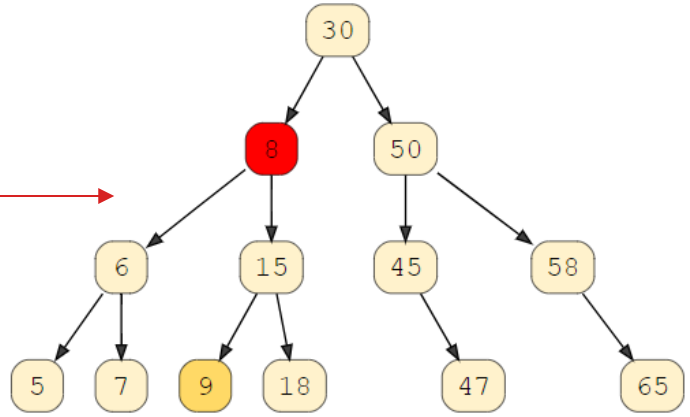


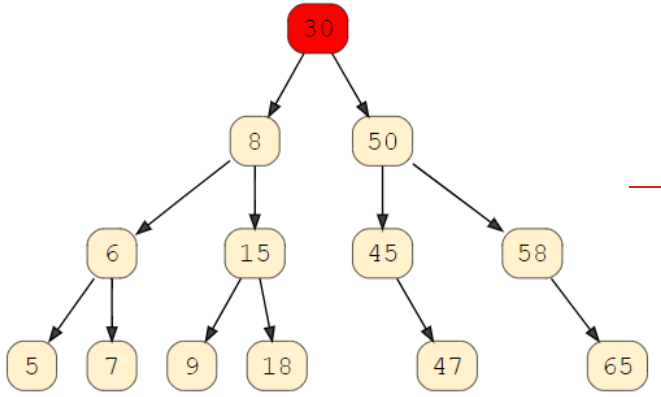
i. rotateLeft(6)



ii. rotateRight(15)

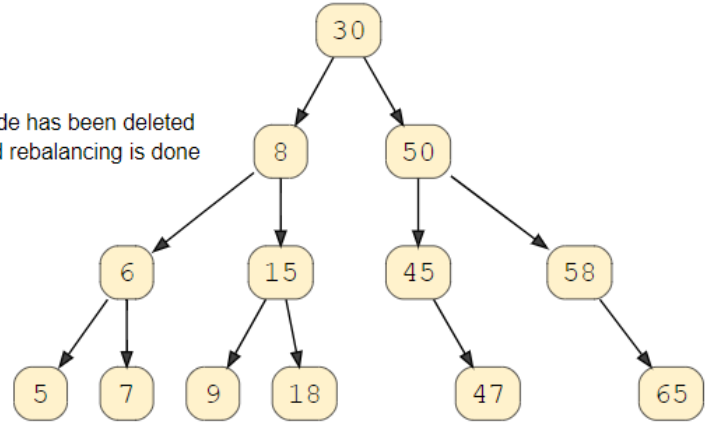






bakctracking finished

node has been deleted
and rebalancing is done



AVL TREE : CREATE NODE & SEARCH NODE

```c
/* Create new avl_node */
struct avl_node *
avl_node_create(int key)
{
  struct avl_node *node = (struct avl_node *)calloc(1, sizeof(struct avl_node));
  if (NULL == node)
    return NULL;

  node->key = key;
  node->height = 1;
  return node;
}
```

```c
struct avl_node *
avl_node_search(struct avl_tree *tree, int key)
{
  if (NULL == tree || NULL == tree->top)
    return NULL;

  struct avl_node *node = tree->top;
  while (node)
  {
    if (key == node->key)
      return node;

    if (key > node->key)
      node = node->right;
    else node = node->left;
  }
  return NULL;
}
```

AVL TREE : INSERT

```
206   /* Insert new node */
207   struct avl_node *
208   avl_node_insert(struct avl_tree *tree, int key)
209   {
210     if (NULL == tree)
211       return NULL;
212
213     struct avl_node *node = tree->top;
214     struct avl_node *new = avl_node_create(key);
215
216     /* Check if empty tree */
217     if (NULL == node)
218     {
219       tree->top = new;
220       return new;
221     }
```

```
223     /* Otherwise, recur down the tree */
224     while (node && node->key != new->key)
225     {
226       if (new->key > node->key )
227       {
228         if (NULL == node->right)
229         {
230           node->right = new;
231           new->parent = node;
232           break;
233         }
234         else
235           node = node->right;
236       }
237       else if (new->key < node->key)
238       {
239         if (NULL == node->left)
240         {
241           node->left = new;
242           new->parent = node;
243           break;
244         }
245         else
246           node = node->left;
247       }
248     }
```

AVL TREE : INSERT

```
250     node = new;
251     /* Update node height and re-balance tree */
252     while (node)
253     {
254       node->height = 1 + max(height(node->left), height(node->right));
255       int this_balance = avl_node_balance_factor(node);
256       int left_balance = avl_node_balance_factor(node->left);
257       int right_balance = avl_node_balance_factor(node->right);
258
259       /* Left - left case */
260       if (this_balance < -1 && left_balance < 0)
261         avl_right_rotate(tree, node);
262
263       /* Right - right case */
264       if (this_balance > 1 && right_balance > 0)
265         avl_left_rotate(tree, node);
266
267       /* Left - right case */
268       if (this_balance < -1 && left_balance > 0)
269       {
270         node->left = avl_left_rotate(tree, node->left);
271         avl_right_rotate(tree, node);
272       }
273
274       /* Right - left case */
275       if (this_balance > 1 && right_balance < 0)
276       {
277         node->right = avl_right_rotate(tree, node->right);
278         avl_left_rotate(tree, node);
279       }
280       node = node->parent;
281     }
282
283     return new;
284   }
```

AVL TREE : MAIN CODE

```c
/* Print the tree pre-order*/
void
avl_tree_print_pre_order(struct avl_node *node)
{
  if (node != NULL)
  {
    avl_node_print(node);
    avl_tree_print_pre_order(node->left);
    avl_tree_print_pre_order(node->right);
  }
}

/*================================================
 * MAIN FUNCTION
 */
int main()
{
  struct avl_tree *tree = calloc(1, sizeof(struct avl_tree));
  avl_node_insert(tree, 1);
  avl_node_insert(tree, 2);
  avl_node_insert(tree, 3);
  avl_node_insert(tree, 4);
  avl_node_insert(tree, 5);
  avl_node_insert(tree, 6);
  avl_node_insert(tree, 7);
  avl_node_insert(tree, 8);
  avl_node_insert(tree, 9);
  avl_node_insert(tree, 10);
  avl_node_insert(tree, 11);
  avl_tree_print_pre_order(tree->top);
}
```

```
PS C:\Users\QuyetDH\Documents\avl_tree-master> gcc .\main.c -o main;./main

    Node: 8 - Height: 4
   /       \
 L:4       R:10
 L:4       R:10
========================
    Node: 4 - Height: 3
   /       \
 L:2       R:6
========================
    Node: 2 - Height: 2
   /       \
 L:1       R:3
========================
    Node: 1 - Height: 1
========================
    Node: 3 - Height: 1
========================
    Node: 6 - Height: 2
   /       \
 L:5       R:7
========================
    Node: 5 - Height: 1
========================
    Node: 7 - Height: 1
========================
    Node: 10 - Height: 3
   /       \
 L:9       R:11
========================
    Node: 9 - Height: 1
========================
    Node: 11 - Height: 2
          \
           R:12
========================
    Node: 12 - Height: 1
========================
```

AVL TREE : REFERENCE

https://www.w3schools.com/dsa/dsa_data_avltrees.php

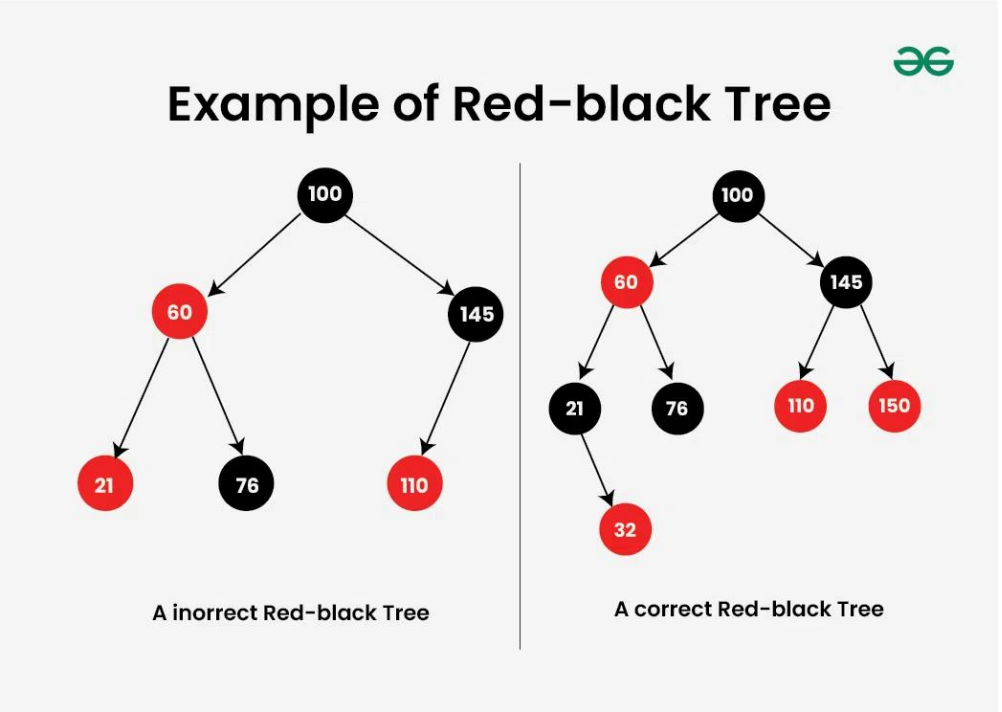
https://www.cs.usfca.edu/~galles/visualization/AVLtree.html

5. RED-BLACK TREE

RED-BLACK TREE : DEFINITION

A Red-Black Tree is a **self-balancing binary search tree** where each node has an additional attribute: a color, which can be either red or black.

- **Node Color**: Each node is either red or black.
- **Root Property**: The root of the tree is always black.
- **Red Property**: Red nodes cannot have red children (no two consecutive red nodes on any path).
- **Black Property**: Every path from a node to its descendant null nodes (leaves) has the same number of black nodes.
- **Leaf Property**: All leaves (NIL nodes) are black.



RED-BLACK TREE : VISUALIZE

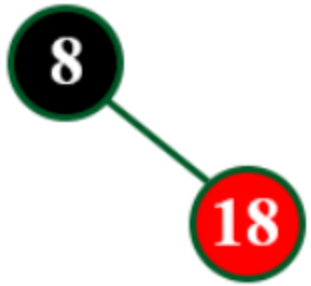
Create a RED BLACK Tree by inserting following sequence of number
8, 18, 5, 15, 17, 25, 40 & 80.

insert (8)

Tree is Empty. So insert newNode as Root node with black color.

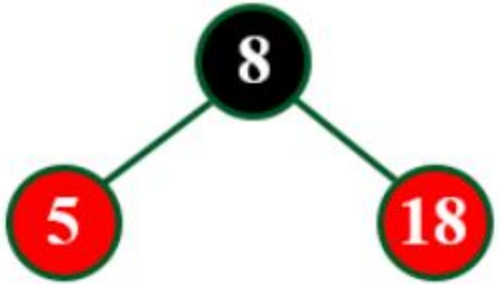
insert (18)

Tree is not Empty. So insert newNode with red color.



insert (5)

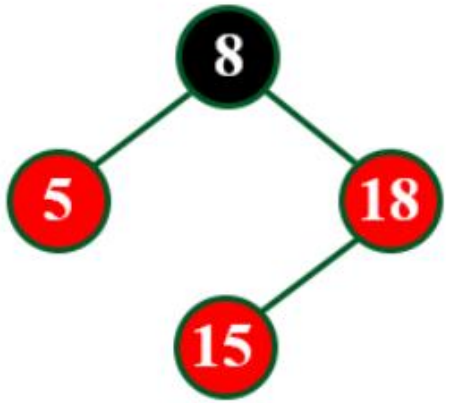
Tree is not Empty. So insert newNode with red color.



RED-BLACK TREE : VISUALIZE

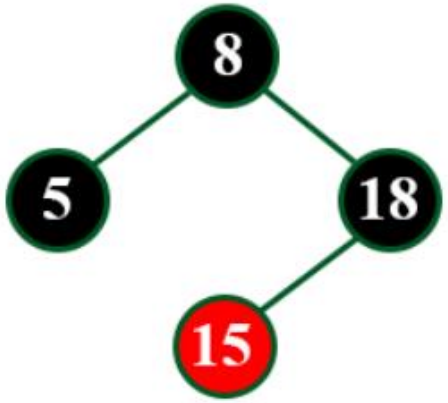
insert (15)

Tree is not Empty. So insert newNode with red color.



Here there are two consecutive Red nodes (18 & 15).
The newnode's parent sibling color is Red
and parent's parent is root node.
So we use RECOLOR to make it Red Black Tree.

After RECOLOR

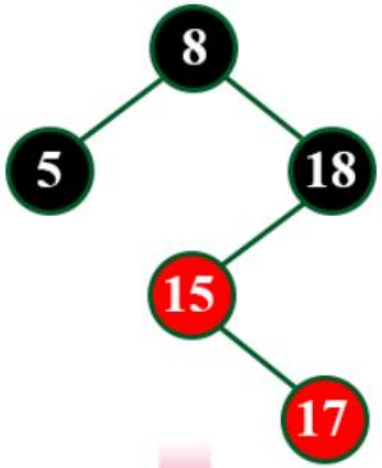


After Recolor operation, the tree is satisfying all Red Black Tree properties.

RED-BLACK TREE : VISUALIZE

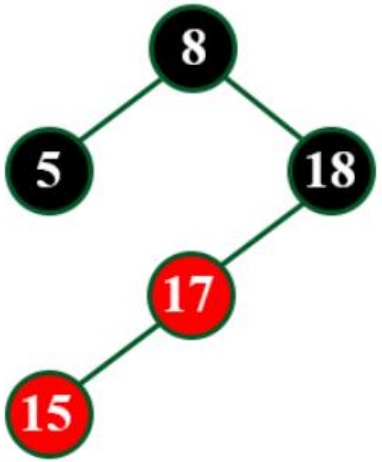
insert (17)

Tree is not Empty. So insert newNode with red color.

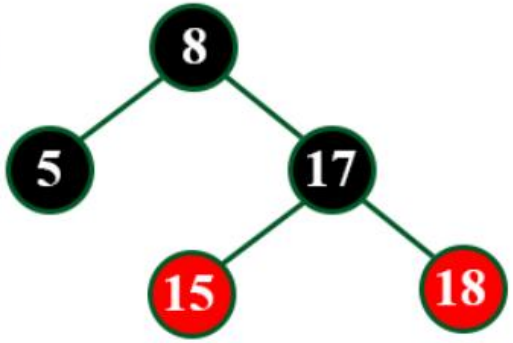


Here there are two consecutive Red nodes (15 & 17).
The newnode's parent sibling is NULL. So we need rotation.
Here, we need LR Rotation & Recolor.

After Left Rotation



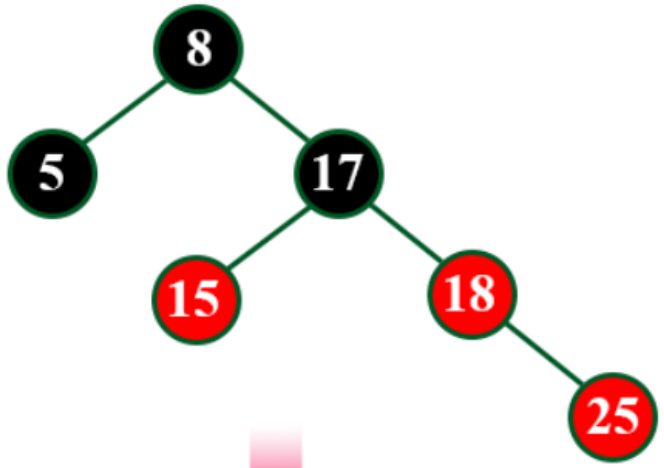
After Right Rotation & Recolor



RED-BLACK TREE : VISUALIZE

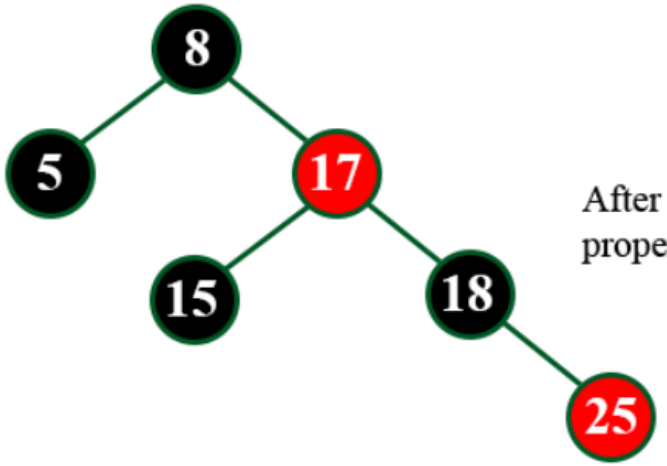
insert (25)

Tree is not Empty. So insert newNode with red color.



Here there are two consecutive Red nodes (18 & 25).
The newnode's parent sibling color is Red
and parent's parent is not root node.
So we use RECOLOR and Recheck.

After Recolor

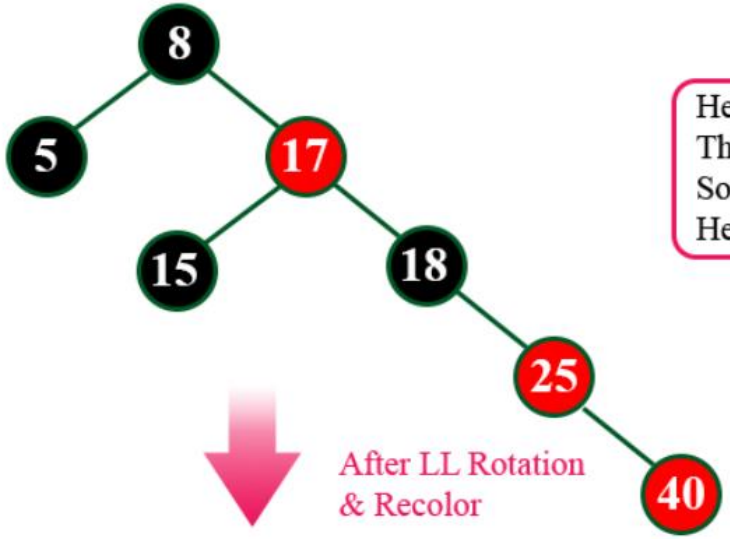


After Recolor operation, the tree is satisfying all Red Black Tree properties.

RED-BLACK TREE : VISUALIZE

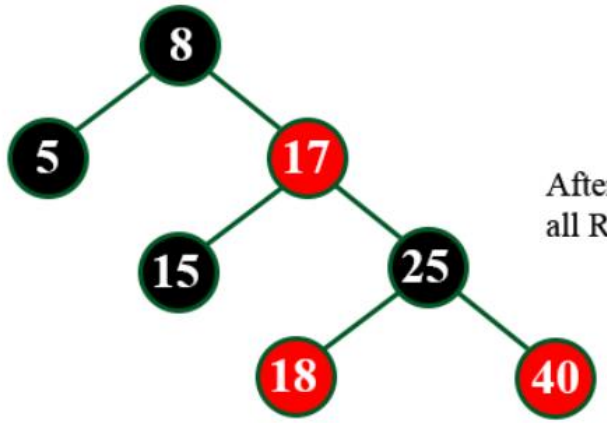
insert (40)

Tree is not Empty. So insert newNode with red color.



Here there are two consecutive Red nodes (25 & 40).
The newnode's parent sibling is NULL
So we need a Rotation & Recolor.
Here, we use LL Rotation and Recheck.

After LL Rotation
& Recolor

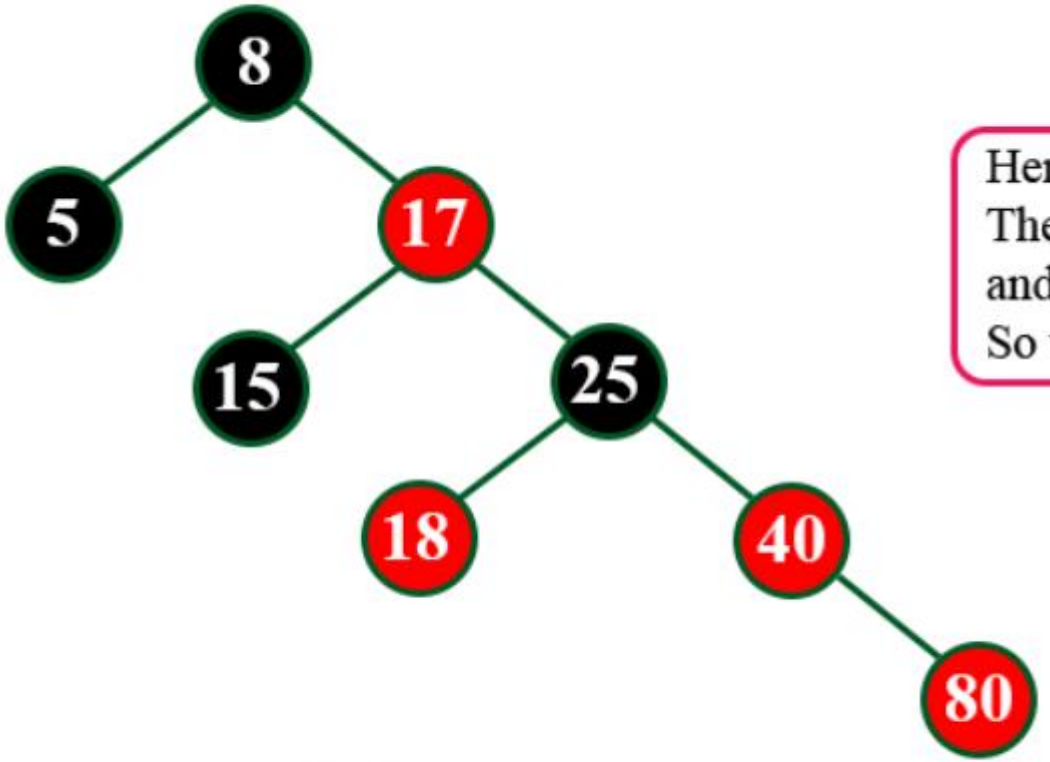


After LL Rotation & Recolor operation, the tree is satisfying all Red Black Tree properties.

RED-BLACK TREE : VISUALIZE

insert (80)

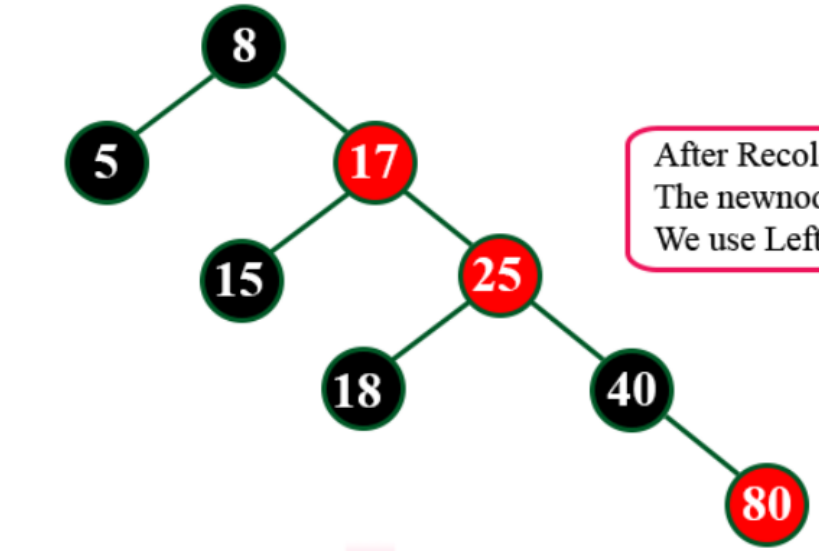
Tree is not Empty. So insert newNode with red color.



Here there are two consecutive Red nodes (40 & 80).
The newnode's parent sibling color is Red
and parent's parent is not root node.
So we use RECOLOR and Recheck.

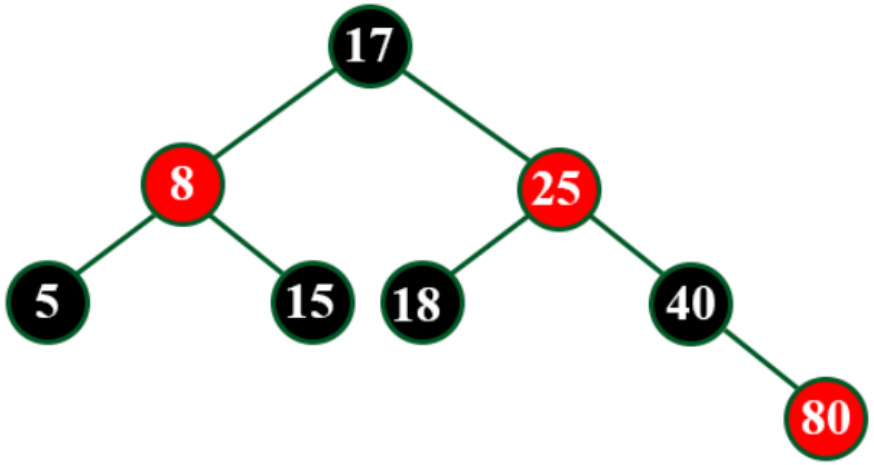
After Recolor

RED-BLACK TREE : VISUALIZE

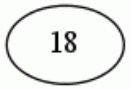


After Recolor again there are two consecutive Red nodes (17 & 25).
The newnode's parent sibling color is Black. So we need Rotation.
We use Left Rotation & Recolor.

After Left Rotation
& Recolor



RED-BLACK TREE : VISUALIZE



https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

viettel
Theo cách của bạn

THANK YOU

Presenter: Trinh Cao Cuong

4/2024